ЗАТВЕРДЖЕНО

Науково-методичною радою
Запорізького
національного університету

протокол від “27” 06.2023 р.
№ 12

КОНСПЕКТ ЛЕКЦІЙ
з навчальної дисципліни
“Організація баз даних та баз знань: Бази знань, засновані на онтологіях ”

для здобувачів вищої освіти освітнього ступеня “магістр ”

спеціальності 126 “Інформаційні системи та технології ”

освітньо-професійна програма “Інформаційні системи та штучний інтелект ”

Математичний факультет

кафедра комп’ютерних наук

Розробники:

к.т.н., ст. викладач кафедри комп’ютерних наук Добровольський Геннадій Анатолійович

к.т.н., доцент кафедри комп’ютерних наук Кеберле Наталія Геннадіївна

Запоріжжя

2023

УДК 004.91:004.655/658 (076.1)

ББК з973.233-018.2я73+з988-5

К 33

Добровольський Г.А., Кеберле Н. Г. “Організація баз даних та баз знань: бази знань, засновані на онтологіях” / Г.А. Добровольський, Н.Г. Кеберле. Запоріжжя : ЗНУ, 2023. 126 с.

Конспект лекцій “Бази знань, засновані на онтологіях” з навчальної дисципліни “Організація баз даних та баз знань” призначено для здобувачів кваліфікації магістра спеціальності 126 "Інформаційні системи та технології".

Видання описує елементи онтологій, мови специфікації онтологій, зосереджуючись на OWL, RDF та RDFS. Описана мова запитів до онтологій SPARQL, типові процеси створення онтологій. Коротко окреслено технології Semantic Web. Дане видання може бути використано для проведення індивідуальних занять зі студентами, організації самостійної роботи.

Рецензент Єрмолаєв В.А., к.ф.-м.н., професор кафедри комп’ютерних наук Українського католицького університету

Відповідальний за випуск: Шило Г.М., д.т.н, зав. кафедрою комп’ютерних наук Запорізького національного університету

ЗМІСТ

ВСТУП

Метою вивчення дисципліни “Організація баз даних і баз знань” є оволодіння знаннями методів створення інформаційних підсистем, відповідальних за зберігання даних і знань.

Основними завданнями вивчення дисципліни “Організація баз даних і баз знань” є:

- оволодіння базовими знаннями технологій створення інформаційних підсистем на основі баз даних та навичками їх застосування;
- оволодіння базовими знаннями технологій створення інформаційних підсистем на основі баз знань та навичками їх застосування.

Досягнення визначених завдань передбачає використання таких ***інноваційних навчальних технологій***: при проведенні лекцій: майстер-класи; частково-пошуковий; лекція; демонстрація; пояснення. При проведенні практичних занять: лабораторні роботи. Контрольні заходи: захист лабораторних робіт, опитування, тестування.

Досягнення визначених завдань передбачає досягнення ***загальних компетентностей***:

ЗК 1 Здатність до абстрактного мислення, аналізу та синтезу.

Досягнення визначених завдань передбачає досягнення ***спеціальних (фахових) компетентностей:***

СК1 Здатність розробляти та застосувати ІСТ, необхідні для розв’язання стратегічних і поточних задач.

СК10 Здатність застосовувати методи захисту даних в інформаційних системах.

СК11 Здатність проектувати бази даних і бази знань.

Досягнення визначених завдань передбачає досягнення ***програмних результатів навчання***:

РН1 Відшуковувати необхідну інформацію в науковій і технічній літературі, базах даних, інших джерелах, аналізувати та оцінювати цю інформацію.

РН8 Розробляти моделі інформаційних процесів та систем різного класу, використовувати методи моделювання, формалізації, алгоритмізації та реалізації моделей з використанням сучасних комп’ютерних засобів

РН9 Розробляти і використовувати сховища даних, здійснювати аналіз даних для підтримки прийняття рішень.

Онтології. Походження терміну ontology

Parmenides: вчення про буття

Computer science: Tom Gruber, 1993. Formal and explicit specification of a shared conceptualization

People cannot share knowledge if they do not speak common language

Визначення спільної мови:

- **синтаксис** − символи та концепції,
- **семантика** − узгоджене розуміння змісту символів (символи <=> концепції),
- **ієрархія** − класифікація концепцій (клас-підклас),
- **тезаурус** − асоціації та відношення між концептами,
- **онтологія** − правила та знання про те, які з відношень мають сенс.

Символ

В онтології (в контексті інформатики) символ − це абстрактне представлення поняття або сутності, яке використовується для його позначення або ідентифікації. Символи використовуються для представлення понять, категорій або типів речей, таких як об’єкти, події або процеси, які існують у певній області чи галузі знань.

В онтології символи, як правило, визначаються у формальній мові, такій як RDF або OWL, яка використовується для представлення знань і концепцій структурованим і систематичним способом. Символ визначається як термін, який представляє концепцію або сутність і може використовуватися для ствердження тверджень або пропозицій щодо них.

Символи в онтології можуть бути визначені різними способами, залежно від домену та необхідного рівня абстракції. Наприклад, символом може бути слово, фраза або рядок символів, як-от “яблуко” або “фрукт”. Це також може

бути унікальний ідентифікатор, наприклад **URI** ***(уніфікований ідентифікатор ресурсу)***, який використовується для представлення певної концепції чи сутності в графі знань або базі даних.

Концепція

В онтології (в контексті інформатики) концепція відноситься до абстрактної ідеї або категорії, яка використовується для представлення певного типу сутності, події або процесу в межах області знань. Концепцію можна розглядати як розумову конструкцію, яка фіксує істотні характеристики класу об’єктів і може використовуватися для міркування про зв’язки та властивості цих об’єктів.

В онтології поняття зазвичай визначаються за допомогою формальної мови, такої як RDF або OWL, яка забезпечує точний і однозначний спосіб представлення знань і понять. Концепція визначається набором атрибутів або властивостей, які описують її визначальні характеристики та можуть бути використані, щоб відрізнити її від інших концепцій чи сутностей у межах того самого домену.

Наприклад, в онтології тварин поняття “ссавці” можна визначити як клас тварин, які мають волосся, народжують живих дитинчат і виробляють молоко для вигодовування дитинчат. Це поняття можна далі розділити на більш конкретні підпоняття, такі як “примат” або “гризун”, кожне з яких має власний набір визначальних властивостей.

Концепції в онтології використовуються для представлення знань та інформації структурованим і систематичним способом і можуть використовуватися в різних програмах, таких як пошук інформації, управління знаннями та штучний інтелект.

Класифікація онтологій

За рівнем формальності

Контрольований словник в CS ***(Computer Science)*** − це попередньо визначений набір стандартизованих термінів або фраз, які використовуються для представлення концепцій, ідей або сутностей у певній області.

Це тип методу організації інформації, який забезпечує узгодженість і точність під час пошуку й аналізу даних. Контрольована лексика зазвичай використовується в системах пошуку інформації, таких як бази даних, пошукові системи та системи керування вмістом, для підвищення точності та запам’ятовування під час пошуку. Використовуючи контрольований словник, користувачі можуть легко знаходити потрібну інформацію без необхідності переглядати нерелевантні дані чи синоніми.

Логіка першого порядку (FOL), також відома як логіка предикатів, − це формальна мова, яка використовується для точного й однозначного вираження тверджень і пропозицій про об’єкти та їхні зв’язки. Це математична система, яка дозволяє нам міркувати про властивості та зв’язки об’єктів у структурований та систематичний спосіб.

У логіці першого порядку ми використовуємо логічні символи, такі як “і”, “або”, “ні”, “якщо-то” і “для всіх”, щоб висловити логічні пропозиції щодо властивостей і відносин об’єктів. Основні елементи логіки першого порядку включають змінні, константи, предикати і квантори. Змінні використовуються для представлення об’єктів або сутностей, тоді як константи використовуються для представлення конкретних об’єктів. Предикати використовуються для представлення властивостей або зв’язків між об’єктами, а квантори використовуються для визначення обсягу твердження чи пропозиції.

Логіка першого порядку широко використовується в інформатиці, математиці, філософії та лінгвістиці для вираження складних ідей і міркувань про властивості та зв’язки об’єктів у структурований та систематичний спосіб.

За застосуванням

Онології верхнього рівня

В інформатиці **онтології верхнього рівня** − це онтології загального призначення високого рівня, які забезпечують базову структуру для представлення знань у різних областях і програмах. Онтології верхнього рівня охоплюють базові категорії та поняття, які є спільними для всіх або більшості доменів, і забезпечують загальний словник для опису та міркування щодо сутностей, подій і процесів.

Онтології верхнього рівня зазвичай включають невелику кількість дуже загальних понять, таких як час, простір, події та сутності, які можна використовувати для побудови більш конкретних і детальних онтологій для окремих доменів. Вони розроблені таким чином, щоб бути незалежними від домену та надавати загальну структуру для інтеграції знань із різних джерел і доменів.

Приклади онтологій верхнього рівня в інформатиці включають:

- **Базова формальна онтологія (BFO): BFO** − це онтологія верхнього рівня, яка забезпечує формальну структуру для охоплення загальних категорій і концепцій, таких як фізичні об’єкти, якості та процеси. Він розроблений для сумісності з іншими онтологіями та широко використовується в біомедичних дослідженнях.
- **DOLCE: DOLCE** − це базова онтологія, яка охоплює основні категорії та поняття, такі як час, простір і події, і забезпечує формальну структуру для представлення знань у різних областях.

- **OpenCyc: OpenCyc** − це велика онтологія загального призначення, яка надає загальний словник для опису сутностей, подій і процесів у різних областях, таких як біологія, фізика та соціальні науки.

Онтології верхнього рівня важливі в інформатиці, оскільки вони забезпечують спільну мову та структуру для представлення знань, а також полегшують взаємодію та інтеграцію різних систем і програм.

Онтології предметної області

В інформатиці предметно-орієнтовані онтології − це онтології, які призначені для охоплення знань і концепцій у певній області чи області знань. На відміну від онтологій верхнього рівня, які забезпечують загальну структуру для представлення знань у різних доменах, онтології, орієнтовані на домен, пристосовані до конкретних потреб і вимог конкретного домену.

Предметно-спеціальні онтології охоплюють концепції, зв’язки та обмеження, характерні для конкретної області, наприклад медицини, фінансів чи інженерії. Вони, як правило, розробляються експертами предметної області у співпраці з інженерами знань і призначені для забезпечення формального, структурованого представлення знань у межах домену.

Приклади предметно-орієнтованих онтологій в інформатиці включають:

1. **Генна онтологія (GO): GO** − це онтологія, яка фіксує знання про функції генів і зв’язки між різними видами. Він надає стандартний словник для анотування та аналізу генних даних і широко використовується в дослідженнях біоінформатики.
2. **Бізнес-онтологія фінансової індустрії (FIBO): FIBO** − це онтологія, яка фіксує знання про фінансові концепції та відносини, такі як фінансові інструменти, ринки та транзакції. Він створений для надання стандартного словника для представлення фінансових даних і покращення сумісності між фінансовими системами.

3. **RadLex: RadLex** − це онтологія, яка фіксує знання про процедури радіологічної візуалізації та термінологію. Він надає стандартний словник для анотування та обміну радіологічними даними та широко використовується в радіологічних дослідженнях і практиці.

Предметно-орієнтовані онтології важливі в інформатиці, оскільки вони забезпечують формальне, структуроване представлення знань у певній області, яке можна використовувати для покращення інтеграції даних, сумісності та аналізу. Вони також полегшують розробку предметно-спеціальних програм та інструментів, таких як системи підтримки прийняття рішень та експертні системи.

Онтології класу задач

В інформатиці онтології, орієнтовані на завдання, − це онтології, які призначені для охоплення знань і концепцій, які є специфічними для конкретного завдання чи програми. На відміну від онтологій верхнього рівня або предметно-спеціальних онтологій, які забезпечують загальну або предметно-спеціальну структуру для представлення знань, онтології для конкретного завдання призначені для охоплення конкретних знань, необхідних для конкретного завдання чи проблеми.

Онтології, що стосуються конкретного завдання, як правило, розробляються експертами в галузі завдання або прикладної області у співпраці з інженерами знань і призначені для забезпечення структурованого та формального представлення знань, необхідних для завдання. Вони можуть включати концепції, зв’язки та обмеження, які є специфічними для завдання, а також будь-які відповідні базові знання із загальної або предметної онтології.

Приклади специфічних для завдань онтологій в інформатиці включають:

1. **Онтологія медичних процедур (MPO): MPO** − це онтологія, яка фіксує знання про медичні процедури, включаючи етапи, необхідні ресурси та

обладнання, а також очікувані результати. Він призначений для підтримки розробки систем підтримки прийняття рішень для медичних процедур.

2. **Онтологія управління повітряним рухом (ATCO): ATCO** − це онтологія, яка фіксує знання про управління повітряним рухом, включаючи правила та процедури керування повітряними суднами, відповідні правила та стандарти, а також наявні ресурси та обладнання. Він призначений для підтримки розвитку інтелектуальних систем управління повітряним рухом.
3. **Онтологія електронної комерції (ECO): ECO** − це онтологія, яка фіксує знання про електронну комерцію, включаючи типи продуктів і послуг, відповідні бізнес-процеси та транзакції, а також правила й норми, які регулюють електронну комерцію. Він призначений для підтримки розробки програм і систем електронної комерції.

Онтології, орієнтовані на завдання, важливі в інформатиці, тому що вони забезпечують структуроване та формальне представлення знань, необхідних для конкретного завдання чи програми, які можна використовувати для розробки інтелектуальних систем і програм, які підтримують прийняття рішень і розв’язання проблем.

Онтології прикладних задач

Онтології прикладних задач (application ontologies) в інформатиці онтології додатків − це онтології, призначені для підтримки розробки конкретних програмних додатків або систем. Онтології додатків охоплюють знання та концепції, необхідні для конкретного додатка, і забезпечують структуроване та формальне представлення цих знань.

Онтології додатків, як правило, розробляються експертами в області у співпраці з інженерами знань і розробниками програмного забезпечення. Вони призначені для інтеграції в процес розробки програмного забезпечення та

використовуються для підтримки розробки програмних додатків, які можуть міркувати та маніпулювати знаннями, представленими в онтології.

Приклади прикладних онтологій в інформатиці включають:

1. **Semantic Web Rule Language *(SWRL, Мова правил семантичної мережі):*** SWRL − це мова онтології, яка використовується для представлення правил і обмежень, які можна застосувати до даних семантичної мережі. Він використовується для підтримки розробки інтелектуальних систем, які можуть міркувати та маніпулювати даними семантичної мережі.
2. **Common Warehouse Metamodel, *Загальна метамодель сховища (CWM):* CWM** − це онтологія, яка використовується для представлення структур і операцій сховища даних. Він використовується для підтримки розробки додатків і систем зберігання даних.
3. **Educational Modeling Language, Мова освітнього моделювання (EML): EML** − це онтологія, яка використовується для представлення освітніх моделей і ресурсів, таких як цілі навчання, критерії оцінювання та освітня діяльність. Він використовується для підтримки розробки освітніх програм і систем.

Онтології додатків важливі в інформатиці, оскільки вони забезпечують структуроване та формальне представлення знань, необхідних для конкретного додатка, які можуть бути використані для підтримки розробки інтелектуальних систем і додатків, які можуть міркувати про ці знання та маніпулювати ними.

Склад онтології, структура онтології

Класи

Класи в онтологіях відносяться до категорій або типів речей або понять, які мають спільні характеристики або властивості. Класи є будівельними

блоками онтологій і використовуються для організації та класифікації знань і концепцій, представлених в онтології.

Класи в онтологіях визначаються за допомогою комбінації логічних аксіом і семантичних описів. Логічні аксіоми визначають зв’язки між класами, такі як відносини успадкування та підпорядкування, тоді як семантичні описи забезпечують більш природний мовний опис класу та його властивостей.

Логічні аксіоми

Логічні аксіоми − це формальні твердження, які визначають зв’язки між класами, властивостями та індивідами в онтології. Логічні аксіоми використовуються для визначення структури та поведінки онтології та забезпечують спосіб міркувати про знання, представлені в онтології.

Логічні аксіоми в онтологіях зазвичай виражаються логічною мовою, такою як логіка першого порядку або логіка опису, і використовуються для визначення наступних відносин:

1. **Ієрархія класів (Class hierarchy):** Логічні аксіоми можна використовувати для визначення зв’язків підкласів між класами в онтології. Наприклад, аксіома “Собака є підкласом Ссавців” визначає, що клас Собака є більш конкретним підкласом більш загального класу Ссавці.
2. **Обмеження властивостей (Property constraints):** Логічні аксіоми можна використовувати для визначення обмежень на значення, які можна призначити властивостям. Наприклад, аксіома “лише hasHeight (float або integer)” визначає, що властивість hasHeight може мати лише значення типу float або integer.
3. **Обмеження домену(Domain) та діапазону (Range):** Логічні аксіоми можна використовувати для визначення обмежень на класи, які можуть бути пов’язані з властивістю. Наприклад, аксіома “hasChild domain Person” визначає, що властивість hasChild може бути пов’язана лише з екземплярами класу Person.

4. **Обмеження потужності (Cardinality constraints):** Логічні аксіоми можна використовувати для визначення обмежень щодо кількості значень, які можна призначити властивості. Наприклад, аксіома “hasParent max 2” визначає, що екземпляр може мати не більше двох значень властивості hasParent.

Логічні аксіоми важливі в онтологіях, оскільки вони забезпечують формальний і точний спосіб визначення зв’язків між класами, властивостями та індивідами. Вони дозволяють автоматизувати міркування та висновки щодо знань, представлених в онтології, і підтримують розробку інтелектуальних систем і програм, які можуть маніпулювати цими знаннями та міркувати про них.

Семантичні описи

В інформатиці семантичні описи відносяться до описів природною мовою класів, властивостей та індивідів, представлених в онтології. Семантичні описи забезпечують зрозумілий для людини короткий виклад значення та наміру формальних логічних аксіом, які визначають онтологію.

Семантичні описи в онтологіях зазвичай виражаються в стандартному форматі, такому як мова веб-онтології (OWL), і можуть містити різноманітну інформацію, зокрема:

1. **Ім’я:** ім’я класу, властивості чи індивідуума, яке зазвичай є лаконічним і описовим терміном, який відображає суть представленої концепції.
2. **Визначення:** докладний опис значення та наміру концепції, що забезпечує більш повне розуміння концепції, ніж сама назва.
3. **Синоніми:** альтернативні назви або мітки для концепції, які можуть використовуватися для підтримки інтеграції даних і сумісності в різних онтологіях і програмах.

4. **Приклади:** Приклади екземплярів, які належать до класу, які можуть допомогти пояснити значення та обсяг представленого поняття.

Семантичні описи важливі в онтологіях, оскільки вони забезпечують спосіб подолання розриву між формальними логічними аксіомами, які визначають онтологію, і людським розумінням представлених концепцій. Вони допомагають гарантувати, що онтологія є як машиночитаною, так і людиною, і можуть сприяти співпраці та спілкуванню між експертами в області, інженерами знань і розробниками програмного забезпечення.

Наприклад, в онтології про тварин клас “ссавець” може бути визначений як підклас класу “тварина” з додатковими властивостями, такими як “має хутро” і “народжує живого”. Клас “собака” може бути визначений як підклас класу “ссавець” з додатковими властивостями, такими як “має чотири ноги” та “гавкає”.

Класи в онтологіях є важливими, оскільки вони забезпечують спосіб організації та класифікації знань і концепцій у структурований і формальний спосіб. Вони можуть використовуватися для підтримки міркувань і висновків в інтелектуальних системах і програмах, а також можуть полегшити інтеграцію даних і взаємодію, надаючи спільний словниковий запас і структуру для представлення знань.

Атрибути

В онтологіях **(CS) атрибут** − це характеристика або властивість класу чи індивіда. Атрибути використовуються для опису властивостей або особливостей концепції чи сутності, які мають відношення до домену, що моделюється.

Атрибути можуть бути як простими, так і складними. Прості атрибути − це базові типи даних, наприклад рядки, числа або дати, які можна використовувати для представлення таких значень, як зріст, вага або колір.

Складні атрибути складаються з інших класів або властивостей і можуть використовуватися для представлення більш складних зв’язків між сутностями.

Атрибути зазвичай визначаються за допомогою комбінації логічних аксіом і семантичних описів. Логічні аксіоми визначають зв’язки між атрибутами та класами, такими як обмеження домену та діапазону, тоді як семантичні описи забезпечують опис природною мовою атрибута та його значення.

Наприклад, в онтології про автомобілі клас “Автомобіль” може мати такі атрибути, як “марка”, “модель”, “рік” і “колір”. Атрибут “make” може мати обмеження домену “Car” і обмеження діапазону “string”, що вказує на те, що це атрибут класу Car і що його значення є рядковим типом даних. Семантичний опис для атрибута "make" може включати таке визначення, як "Марка автомобіля, із зазначенням виробника або марки".

Атрибути важливі в онтологіях, оскільки вони забезпечують спосіб опису характеристик і властивостей концепцій і сутностей, що моделюються. Вони можуть бути використані для підтримки інтеграції даних і взаємодії, надаючи спільний словник і структуру для представлення знань, а також можуть використовуватися для підтримки міркувань і висновків в інтелектуальних системах і програмах.

Відношення

Відношення стосуються зв’язків або зв’язків між класами, властивостями та індивідами в онтології. Зв’язки використовуються для опису зв’язків і взаємодій між поняттями та сутностями в домені та забезпечують спосіб представлення складних структур знань і залежностей.

Відносини в онтологіях можна класифікувати за різними категоріями на основі їхніх характеристик і властивостей. Деякі поширені типи відносин включають:

1. **Властивість об’єкта, Object property:** властивості об’єкта представляють бінарні зв’язки між індивідами або класами. Наприклад, властивість об’єкта “hasParent” з’єднує дочірню особину з батьківською особою або класом.
2. **Властивість типу даних, Datatype property:** властивості типу даних представляють атрибути або характеристики особи чи класу. Наприклад, властивість типу даних "hasHeight" пов'язує індивіда зі значенням його висоти.
3. **Симетрична властивість, Symmetric property:** симетричні властивості − це властивості об’єктів, які діють в обох напрямках. Наприклад, симетрична властивість "isSiblingOf" з'єднує двох осіб, які мають одного батьківського елемента.
4. **Перехідна властивість, Transitive property.** Перехідні властивості − це властивості об’єктів, які зберігаються між окремими особами ланцюгом. Наприклад, транзитивна властивість “isAncestorOf” пов’язує індивіда з усіма його предками.
5. **Зворотна властивість, Inverse property:** Зворотні властивості − це пари властивостей об’єкта, які пов’язані одна з одною. Наприклад, зворотні властивості "hasChild" і "isChildOf" з'єднують батьківську особину з дочірньою особистістю і навпаки.

Відносини важливі в онтологіях, оскільки вони забезпечують спосіб представлення складних структур знань і залежностей. Вони дозволяють автоматизувати міркування та висновки щодо знань, представлених в онтології, і підтримують розробку інтелектуальних систем і програм, які можуть маніпулювати цими знаннями та міркувати про них.

Обмеження

Обмеження в онтологіях відносяться до обмежень або правил, які застосовуються до понять і зв'язків всередині онтології. Ці обмеження можна

використовувати для забезпечення узгодженості та коректності онтології та її додатків. Деякі поширені типи обмежень в онтологіях включають:

1. **Обмеження кількості, Cardinality restrictions:** вони визначають мінімальну та максимальну кількість зв’язків між поняттями. Наприклад, клас, який представляє особу, може мати обмеження кількості, яке вказує, що кожна особа може мати лише одну дату народження.
2. **Обмеження діапазону, Range restrictions:** вони визначають тип значення, яке можна призначити властивості чи атрибуту. Наприклад, властивість, яка представляє вік людини, може мати обмеження діапазону, яке визначає, що значення має бути цілим числом.
3. **Обмеження домену, Domain restrictions:** вони визначають типи понять, які можуть бути пов’язані одне з одним. Наприклад, властивість, яка представляє роботодавця особи, може мати обмеження домену, яке визначає, що пов’язане поняття має бути компанією.
4. **Обмеження значень, Value restrictions:** вони визначають дозволені значення для властивості або атрибута. Наприклад, властивість, що представляє стать людини, може мати обмеження значення, яке вказує, що значення має бути або “чоловічим”, або “жіночим”.

Загалом, обмеження в онтологіях відіграють важливу роль у забезпеченні цілісності та узгодженості онтології та її додатків.

Структура онтології

В інформатиці **ABox (Assertion Box)** − це компонент онтології, що містить твердження щодо окремих екземплярів понять. Він використовується для представлення даних або фактів про певну область в онтології.

В онтології ****TBox (Terminology Box)**** містить схему або таксономію понять, тоді як ABox містить екземпляри цих понять. ABox складається з окремих екземплярів понять, а також зв’язків між цими екземплярами.

Наприклад, якщо онтологія містить клас, що представляє людину, і клас, що представляє компанію, ABox може містити екземпляри людей і компаній, а також твердження про зв’язки між ними (наприклад, “Джон працює в компанії ABC”).

ABox забезпечує спосіб представлення фактичних даних або фактів, які моделює онтологія. Це дозволяє використовувати онтологію для міркувань і висновків на основі даних, а також дозволяє інтегрувати дані з різних джерел. Крім того, ABox можна використовувати для перевірки узгодженості даних зі схемою, представленою в TBox.

Загалом, ABox є важливим компонентом онтології, оскільки він забезпечує спосіб представлення фактичних даних або фактів про домен у структурований та формалізований спосіб.

Resource Description Framework (RDF)

RDF означає Resource Description Framework, який є стандартом для моделювання та обміну метаданими та даними у Всесвітній павутині. RDF забезпечує гнучкий спосіб опису ресурсів, таких як веб-сторінки, книги та люди, а також їхні зв’язки з іншими ресурсами.

RDF базується на ідеї операторів **`суб’єкт-предикат-об’єкт`**, також відомих як трійки. У трійці RDF суб’єктом є описуваний ресурс, предикатом є властивість або зв’язок суб’єкта, а об’єктом є значення цієї властивості або пов’язаного ресурсу.

Наприклад, вислів “Книгу “Мобі Дік” написав Герман Мелвилл” можна виразити у вигляді трійки RDF:

Subject: Книга “Мобі Дік”

Predicate: має автора

Object: Герман Мелвилл

Subject: Герман Мелвилл

Predicate: написав

Object: Книга “Мобі Дік”

Дані RDF можуть бути представлені в різних форматах, таких як **RDF/XML**, **Turtle** або **JSON-LD**, і запитуватися за допомогою мови запитів **SPARQL**.

RDF широко використовується в семантичній мережі, де він дозволяє машинам розуміти та обробляти дані в Інтернеті. Він також використовується в різних програмах, таких як опис метаданих, представлення знань та інтеграція даних.

Основними елементами RDF ***(Resource Description Framework)*** є:

- **`Resource,`** ***(ресурс)*****: Resource** – це будь-яка ідентифікована сутність, наприклад веб-сторінка, особа або фізичний об’єкт. Кожен ресурс ідентифікується **URI** ***(уніфікованим ідентифікатором ресурсу)*** або порожнім вузлом.
- **`Property`**, ***(властивість)*****: Property** описує характеристику або зв’язок ресурсу. У RDF властивості також називають предикатами. Кожна властивість ідентифікується URI.
- **`Value`**, ***(значення)*****: Value** – це об’єкт або тип даних властивості. Значеннями можуть бути літерали, такі як рядки чи числа, або інші ресурси, визначені URI.
- **`Triple`**, ***(трійка)*****: Triple** є основною одиницею інформації в RDF і складається з суб’єкта, предиката та об’єкта. Це твердження про ресурс, наприклад, “книгу “Мобі Дік” написав Герман Мелвілл”.

- **`Namespace`**, ***(простір імен)*: Простір імен** – це спосіб скорочення URI в RDF. Визначивши префікс простору імен, повний URI можна замінити префіксом у операторах RDF.

Ці базові елементи складають основу RDF і дозволяють створювати багаті взаємопов’язані описи ресурсів в Інтернеті.

URI

URI ***(уніфікований ідентифікатор ресурсу)*** − це рядок символів, який ідентифікує ресурс в Інтернеті. URI використовуються для пошуку та доступу до ресурсів, таких як веб-сторінки, зображення, файли та служби тощо.

URI складається з двох частин: схеми та частини, специфічної для схеми. Схема вказує протокол або схему, яка використовується для доступу до ресурсу, наприклад **HTTP, FTP** або **mailto**. Специфічна частина схеми ідентифікує сам ресурс, наприклад ім’я домену, шлях або рядок запиту.

Наприклад, розглянемо URI *https://www.dbpedia.org/resource/Currency*

Namespace dbr = *https://www.dbpedia.org/resource/*

Скорочений варіант *dbr:Currency*

У цьому URI “https” − це схема, а “*www.dbpedia.org/*” − це частина схеми. Схема "https" вказує, що доступ до ресурсу здійснюється за допомогою протоколу HTTPS, а спеціальна частина схеми "www.dbpedia.org/" ідентифікує доменне ім’я та шлях до ресурсу.

URI, URL, URN

URI ***(уніфікований ідентифікатор ресурсу)*** − це рядок символів, який ідентифікує ресурс в Інтернеті. URI можна використовувати для пошуку та доступу до ресурсів, таких як веб-сторінки, зображення, файли та служби. URI також поділяються на два типи: **URL** ***(уніфіковані покажчики ресурсів)*** і **URN** ***(уніфіковані імена ресурсів)***.

URL-адреса − це певний тип URI, який ідентифікує ресурс за його розташуванням або адресою в Інтернеті. URL-адреса містить схему (наприклад, HTTP, FTP), ім’я домену та шлях до ресурсу.

З іншого боку, **URN − це тип URI**, який ідентифікує ресурс за назвою, а не за місцем розташування. URN мають бути постійними глобально унікальними ідентифікаторами, які залишаються дійсними навіть у разі переміщення ресурсу або зміни його розташування. На відміну від URL, URN не містять інформації про розташування ресурсу.

Наприклад, URL-адреса може бути:

https://www.example.com/images/cat.jpg

тоді як URN може бути:

urn:isbn:0-486-27557-4

У першому прикладі URL-адреса вказує розташування файлу зображення на веб-сайті. У другому прикладі URN визначає книгу за її міжнародним стандартним номером книги (ISBN).

Підсумовуючи, хоча як URI, так і URN використовуються для ідентифікації ресурсів в Інтернеті, URI є загальним терміном, який включає як URL-адреси, так і URN. URL-адреси ідентифікують ресурс за його розташуванням в Інтернеті, тоді як URN ідентифікують ресурс за назвою.

RDF/XML

RDF/XML − це один із форматів, який використовується для представлення даних RDF (Resource Description Framework), а також синтаксис на основі XML для серіалізації трійок RDF у машинозчитувані документи.

У RDF/XML кожна трійка RDF представлена як елемент XML із суб’єктом, предикатом і об’єктом. Тема представлена елементом `rdf:Description` з атрибутом `rdf:about`, який визначає URI описуваного ресурсу. Предикат представлено елементом XML з атрибутом `rdf:resource`,

який визначає URI властивості або зв’язку. Об’єкт представлено або елементом XML з атрибутом `rdf:resource`, який визначає URI пов’язаного ресурсу, або простим текстовим елементом XML із літеральним значенням властивості.

Наприклад, наведений нижче фрагмент RDF/XML представляє заяву “Книгу “Moby Dick” написав Herman Melville” у RDF:

```
<rdf:Description rdf:about="http://example.com/books/mobydick">
    <dc:creator
     rdf:resource="http://example.com/authors/hermanmelville"/>
</rdf:Description>
```

У цьому прикладі темою є книга “Moby Dick”, ідентифікована URI “*http://example.com/books/mobydick*”. Предикатом є властивість "`dc:creator`", яка представляє зв’язок між книгою та її автором. Об’єктом є автор Herman Melville, ідентифікований URI “*http://example.com/authors/hermanmelville*”.

RDF/XML забезпечує стандартний спосіб обміну даними RDF між різними системами, але він може бути складним і важким для читання та запису вручну. У результаті інші формати серіалізації RDF, такі як Turtle і JSON-LD, стали більш популярними.

У RDF/XML зазвичай використовуються такі теги:

- **`rdf:RDF:`** елемент верхнього рівня, який містить усі оператори RDF у документі.
- **`rdf:Description:`** Елемент, який використовується для опису ресурсу в RDF. Він представляє предмет потрійного RDF і може містити атрибути rdf:about, rdf:ID або rdf:nodeID для ідентифікації ресурсу.
- **`rdf:about:`** атрибут, який використовується для визначення URI описуваного ресурсу.

- **`rdf:ID`:** атрибут, який використовується для призначення унікального ідентифікатора ресурсу в документі RDF/XML.
- **`rdf:nodeID`:** атрибут, який використовується для призначення унікального ідентифікатора порожньому вузлу, який представляє ресурс, який не має URI.
- **`rdf:type`:** властивість, яка використовується для визначення типу ресурсу, представленого елементом XML з атрибутом rdf:resource, встановленим як URI типу.
- **`rdf:resource`:** атрибут, який використовується для вказівки URI пов’язаного ресурсу в заяві властивості.
- **`rdf:parseType`:** Атрибут, який використовується для вказівки типу об’єкта в заяві властивості. До можливих значень належать "`Resource`", "`Literal`", "`Collection`", "`Seq`".
- **`rdf:li`:** Елемент, який використовується для представлення елементів у списку RDF. Щоб використовувати `rdf:li`, для атрибута `rdf:parseType` потрібно встановити значення "`Collection`" або "`Seq`".

Ці теги й атрибути дозволяють RDF/XML представляти RDF-вирази в ієрархічній та структурованій формі, полегшуючи машинам аналіз і обробку даних RDF.

RDF/Turtle

Turtle ***(Terse RDF Triple Language)*** − популярний і зрозумілий синтаксис для представлення даних **RDF** ***(Resource Description Framework)***. Це дозволяє записувати трійки RDF у стислому та легкому для читання форматі, використовуючи загальний синтаксис і пунктуацію.

У Turtle кожна трійка RDF представлена як оператор `суб’єкт-предикат-об’єкт`, розділений крапками та крапками з комами.

Суб’єкт представлено URI або порожнім ідентифікатором вузла, укладеним у кутові дужки (<>), предикат представлено URI або ім’ям із префіксом, після якого стоїть двокрапка (:), а об’єкт представлено URI, префіксом ім’я, літеральне значення або порожній ідентифікатор вузла. Предикати та об’єкти також можна брабмти в дужки для кращої читабельності.

Ось приклад оператора Turtle:

```
<http://example.com/books/mobydick> dc:creator
<http://example.com/authors/hermanmelville> .
```

У цьому операторі суб’єктом є URI “*http://example.com/books/mobydick*”, предикатом є властивість “`dc:creator`”, а об’єктом є URI “*http://example.com/authors/hermanmelville*”.

Turtle також дозволяє визначати префікси для просторів імен, щоб полегшити написання URI та імен із префіксами. Наприклад, ви можете визначити префікс “`dc`” для простору імен `Dublin Core` за допомогою такого оператора:

```
@prefix dc: <http://purl.org/dc/elements/1.1/>.
```

Визначивши цей префікс, ви можете використовувати "dc:creator" замість повного URI у своїх заявах.

Turtle також підтримує різні типи даних для літералів, включаючи цілі числа, числа з плаваючою точкою, дати та рядки. Ви можете вказати тип даних за допомогою нотації ^^, наприклад:

```
<http://example.com/books/mobydick> dc:publicationDate "1851-10-18"^^xsd:date .
```

У цьому операторі об’єкт є літералом дати з типом даних “`xsd:date`”.

Turtle − це простий і виразний формат, який широко використовується для представлення та обміну даними RDF. Йому часто надають перевагу перед

іншими форматами RDF, оскільки він легше читається людиною та легше писати від руки.

Ви можете знайти більше інформації про формат Turtle у специфікації RDF 1.1 Turtle, яка доступна за такою URL-адресою:

https://www.w3.org/TR/turtle/

Цей документ містить детальне пояснення синтаксису Turtle, включаючи приклади та вказівки щодо використання. Тут також описано, як Turtle відображає абстрактний синтаксис RDF, що корисно, якщо ви хочете зрозуміти, як RDF-трійки представлені в Turtle.

Крім того, є кілька онлайн-ресурсів і навчальних посібників, які надають більше інформації та приклади Turtle:

- Документація RDFlib, яка надає бібліотеку Python для роботи з даними RDF, включаючи розбір і серіалізацію Turtle: *https://rdflib.readthedocs.io/en/stable/gettingstarted.html#parsing-and-serializing-rdf*
- Підручник Jena, який надає RDF-фреймворк на основі Java, який підтримує розбір і серіалізацію Turtle: *https://jena.apache.org/tutorials/rdf_api.html#ch-Turtle*
- Специфікація ***Linked Data Platform (LDP)***, яка використовує Turtle як стандартний формат серіалізації RDF: *https://www.w3.org/TR/ldp-primer/#ldpr-serialization*

JSON-LD

JSON-LD ***(JavaScript Object Notation for Linked Data)*** − популярний формат серіалізації для даних RDF (Resource Description Framework). Він дозволяє представляти RDF-трійки за допомогою **JSON** ***(JavaScript Object Notation)***, який є широко використовуваним форматом обміну даними у веб-додатках.

У JSON-LD трійки RDF представлені як об’єкти JSON із властивістю “@context”, яка визначає зіставлення між URI, що використовуються в

документі, та їхніми відповідними короткими назвами (також відомими як “терміни” або “префікси”). Це полегшує написання та читання документів JSON-LD, оскільки URI можна замінити їх короткими назвами, щоб зменшити багатослівність і покращити читабельність.

Ось приклад оператора JSON-LD:

```
{
   "@context": {
     "dc": "http://purl.org/dc/elements/1.1/",
     "ex": "http://example.com/"
   },
   "@id": "наприклад: книга",
   "dc:title": "Мобі Дік",
   "dc:creator": "ex:Melville",
   "dc:date": "1851-10-18"
}
```

У цьому операторі властивість “@context” визначає дві короткі назви для просторів імен Dublin Core і example.com і зіставляє їх із відповідними URI. Властивість “@id” визначає URI описуваного ресурсу, тоді як властивості “dc:title”, “dc:creator” і “dc:date” представляють трійки RDF для назви книги, автора та дати публікації. , відповідно.

JSON-LD також підтримує різні параметри для представлення порожніх вузлів і типів даних, а також інші функції, такі як кадрування та нормалізація. Ці функції роблять його потужним інструментом для роботи з даними RDF у веб-додатках, особливо у випадках, коли JSON є кращим форматом обміну даними.

Ось кілька основних параметрів роботи з JSON-LD:

- “@context”: ця властивість визначає контекст для документа JSON-LD. Він визначає відображення між термінами (короткими назвами) та їхніми відповідними URI. Контекст можна визначити як об’єкт JSON, URI або посилання на інший контекст.

- “@id”: ця властивість визначає URI ресурсу, описаного в документі JSON-LD.
- “@type”: ця властивість визначає тип описуваного ресурсу. Його можна використовувати для визначення класу або типу ресурсу, наприклад “Особа” або “Організація”.
- “@value”: ця властивість визначає значення літерального вузла, наприклад рядок або число.
- “@language”: ця властивість визначає мову рядкового літералу.
- “@container”: ця властивість визначає тип контейнера, який використовується для набору значень, наприклад масиву або набору.
- “@reverse”: ця властивість визначає зв'язок зворотної властивості між двома ресурсами.
- “@graph”: ця властивість визначає іменований або типовий графік у документі JSON-LD.
- “@vocab”: ця властивість визначає простір імен за умовчанням для документа, який можна використовувати для спрощення синтаксису, дозволяючи визначати терміни без префікса.
- “@base”: ця властивість визначає базовий URI для вирішення відносних URI в документі.

Це деякі з основних параметрів для роботи з JSON-LD, але також є багато додаткових функцій і параметрів. **JSON-LD** − це потужний інструмент для роботи з даними RDF, який можна використовувати в різноманітних програмах і сценаріях використання.

Є кілька доступних ресурсів, щоб навчитися працювати з JSON-LD. ***Ось кілька підручників і посібників, які можуть бути вам корисними:***

- Веб-сайт JSON-LD містить посібник із початку роботи, який охоплює основи JSON-LD, зокрема те, як визначити контекст, створити документ JSON-LD, а також серіалізувати та десеріалізувати дані JSON-LD: *https://json-ld. org/learn/getting-started/*
- W3C має повну специфікацію JSON-LD, яка містить детальну інформацію про синтаксис і семантику JSON-LD: *https://www.w3.org/TR/json-ld/*
- На веб-сайті Google Developers є навчальний посібник із використання JSON-LD для структурованих даних, який містить приклади того, як використовувати JSON-LD для покращення пошукової оптимізації (SEO) вашого веб-сайту: *https://developers.google.com/search /docs/guides/intro-structured-data*
- На веб-сайті Digital Bazaar є серія навчальних посібників із використання JSON-LD, зокрема про те, як створити контекст, використовувати JSON-LD для автентифікації та працювати з такими розширеними функціями, як кадрування та нормалізація: *https://json-ld.org/learn .html*
- Бібліотека RDFLib для Python надає навчальний посібник із роботи з JSON-LD, у якому описано, як читати й записувати дані JSON-LD, а також як використовувати RDFLib для запитів і обробки даних JSON-LD: *https://rdflib.readthedocs.io /en/stable/apidocs/rdflib.plugin.html#module-rdflib.plugins.serializers.jsonld*

Часто вживані простори імен

RDF дозволяє використовувати будь-який дійсний URI як назву властивості або класу. Проте є деякі загальні простори імен, які часто використовуються в RDF для визначення загальновживаних термінів і понять. Ось деякі з найбільш часто використовуваних просторів імен у RDF:

Простір імен **RDF**: *http://www.w3.org/1999/02/22-rdf-syntax-ns#*

Цей простір імен використовується для визначення основних термінів RDF, включаючи rdf:type, rdf:Property, rdf:subject, rdf:predicate та rdf:object.

Простір імен **RDFS**: *http://www.w3.org/2000/01/rdf-schema#*

Цей простір імен використовується для визначення словника схеми RDF, включаючи rdfs:Class, rdfs:subClassOf, rdfs:subPropertyOf, rdfs:range та rdfs:domain.

Простір імен **OWL**: *http://www.w3.org/2002/07/owl#*

Цей простір імен використовується для визначення словника мови веб-онтології (OWL), включаючи owl:Class, owl:ObjectProperty, owl:DatatypeProperty, owl:Restriction, owl:intersectionOf і owl:unionOf.

Простір імен **DC**: *http://purl.org/dc/elements/1.1/*

Цей простір імен використовується для визначення набору елементів метаданих Dublin Core, включаючи dc:title, dc:creator, dc:publisher, dc:date і dc:description.

Простір імен **FOAF**: *http://xmlns.com/foaf/0.1/*

Цей простір імен використовується для визначення словника Friend of a Friend (FOAF), включаючи foaf:Person, foaf:name, foaf:knows, foaf:homepage та foaf:interest.

Простір імен **SKOS**: *http://www.w3.org/2004/02/skos/core#*

Цей простір імен використовується для визначення словника системи простої організації знань (SKOS), включаючи skos:Concept, skos:broader, skos:narrower, skos:related і skos:inScheme.

Простір імен **GEO**: *http://www.w3.org/2003/01/geo/wgs84_pos#*

Цей простір імен використовується для визначення словника геопозиціонування WGS84, включаючи geo:lat, geo:long і geo:alt.

Це лише кілька прикладів із багатьох просторів імен, які можна використовувати в RDF. Вибір простору імен залежить від словника чи онтології, що використовується для опису даних, а також від будь-яких конкретних вимог програми чи системи, що використовується.

RDFS

RDFS розшифровується як “схема RDF”, яка є мовою для визначення структури даних RDF. **RDF** ***(Resource Description Framework)*** − це стандартна модель для опису ресурсів в Інтернеті, а RDFS надає спосіб визначення класів і властивостей, які можна використовувати для опису цих ресурсів.

RDFS дозволяє створювати ієрархії класів і визначати зв’язки між ними за допомогою властивостей. Наприклад, ви можете визначити клас “Особа” з такими властивостями, як “ім’я” та “вік”, а потім визначити клас “Студент”, який є підкласом “Особа” з додатковими властивостями, такими як “дата зарахування” та “спеціальність”. "

RDFS також містить конструкції для визначення обмежень властивостей, таких як мінімальна та максимальна потужність, і для визначення зв’язків між класами, наприклад зв’язків підкласів і підвластивостей.

Основними елементами RDFS є:

1. **Класи:** RDFS дозволяє визначати класи ресурсів, наприклад “Особа”, “Книга” або “Організація”. Класи можуть мати підкласи та можуть бути організовані в ієрархію.
2. **Властивості:** RDFS дозволяє визначати властивості ресурсів, такі як “ім’я”, “вік” або “автор”. Властивості можуть мати **домени** (клас, до якого застосовується властивість) і **діапазони** (тип значення, яке може мати властивість).
3. **Екземпляри:** екземпляри − це окремі ресурси, які належать до певного класу. Наприклад, "John Smith" може бути екземпляром класу "Person".

4. **Відносини підкласів:** RDFS дозволяє визначати відношення підкласів між класами. Наприклад, ви можете визначити клас “Студент” як підклас “Особа”, вказуючи, що всі студенти також є людьми.
5. **Зв’язки підвластивостей:** RDFS дозволяє визначати зв’язки підвластивостей між властивостями. Наприклад, ви можете визначити властивість “givenName” як підвластивість властивості “name”, вказуючи, що “givenName” є більш конкретним типом “name”.
6. **Обмеження домену та діапазону:** RDFS дозволяє вказати обмеження на домени та діапазони властивостей. Наприклад, можна вказати, що властивість “автор” може використовуватися лише для ресурсів класу “Особа”.
7. **Правила виведення:** RDFS містить набір правил, які можна використовувати для отримання додаткових знань із визначень RDFS. Наприклад, якщо ви визначаєте зв’язок підкласу між “Студентом” і “Особою”, RDFS може зробити висновок, що всі студенти також є людьми.

Ресурси

У **RDFS ресурс** − це все, що можна описати за допомогою RDF. Ресурс може бути фізичною річчю, як-от книга чи людина, або абстрактним поняттям, як-от стосунки чи навички. У RDFS ресурси ідентифікуються за допомогою **URI** ***(уніфікованих ідентифікаторів ресурсів)***.

Ресурси можуть належати до одного або кількох класів, які визначають характеристики ресурсу. Наприклад, ресурс може бути екземпляром класу “Person”, який може мати такі властивості, як “ім’я”, “вік” і “стать”.

Окрім екземплярів, RDFS також дозволяє визначати самі класи як ресурси. Ці ресурси класу можуть мати властивості та бути організовані в ієрархії з підкласами та суперкласами.

Класи та властивості

У RDFS ви можете визначити класи за допомогою конструкції "rdfs:Class". Ось приклад того, як визначити клас під назвою "Person":

```
<rdfs:Class rdf:about="http://example.org/ontology/Person">
    <rdfs:label>Person</rdfs:label>
    <rdfs:comment>A person.</rdfs:comment>
</rdfs:Class>
```

У цьому прикладі атрибут "rdf:about" визначає URI класу. Елемент "rdfs:label" надає зрозумілу людині мітку для класу, а елемент "rdfs:comment" надає опис класу.

Ви також можете визначити підкласи за допомогою властивості "rdfs:subClassOf". Ось приклад того, як визначити підклас "Особи" під назвою "Студент":

```
<rdfs:Class rdf:about="http://example.org/ontology/Student">
    <rdfs:label>Student</rdfs:label>
    <rdfs:comment>A student, who is a person.</rdfs:comment>
    <rdfs:subClassOf
rdf:resource="http://example.org/ontology/Person"/>
</rdfs:Class>
```

У цьому прикладі атрибут “rdf:resource” властивості “rdfs:subClassOf” визначає URI суперкласу.

Ви також можете визначити властивості класів за допомогою властивості "rdfs:domain". Наприклад, щоб визначити властивість “name” для класу “Person”, ви можете використати такий оператор RDF:

```
<rdf:Property rdf:about="http://example.org/ontology/name">
    <rdfs:label>Name</rdfs:label>
    <rdfs:comment>The name of a person.</rdfs:comment>
```

```
    <rdfs:domain
rdf:resource="http://example.org/ontology/Person"/>
</rdf:Property>
```

У цьому прикладі властивість “rdfs:domain” вказує, що властивість “name” застосовується до ресурсів класу “Person”.

Екземпляри

У RDFS ви можете визначити екземпляри класів за допомогою операторів RDF. Екземпляр — це певний ресурс, який належить до певного класу. Ось приклад того, як визначити екземпляр класу "Person":

```
<rdf:Description rdf:about="http://example.org/people/john">
    <rdf:type rdf:resource="http://example.org/ontology/Person"/>
    <name>John Smith</name>
    <age>35</age>
    <gender>Male</gender>
</rdf:Description>
```

У цьому прикладі властивість “rdf:type” вказує, що ресурс із URI “http://example.org/people/john” є екземпляром класу “Person”. Властивості "ім'я", "вік" і "стать" надають додаткову інформацію про ресурс.

Ви також можете визначити екземпляри, які належать до підкласів класу. Наприклад, якщо ви визначили підклас "Person" під назвою "Student", ви можете визначити екземпляр класу "Student" таким чином:

```
<rdf:Description rdf:about="http://example.org/people/jane">
    <rdf:type rdf:resource="http://example.org/ontology/Student"/>
    <name>Jane Doe</name>
    <age>20</age>
    <gender>Female</gender>
    <studentID>12345</studentID>
```

```
</rdf:Description>
```

У цьому прикладі властивість “rdf:type” вказує, що ресурс із URI “*http://example.org/people/jane*” є екземпляром класу “Student”, який є підкласом “Person”. Властивість “studentID” є специфічною для класу “Student”.

Обмеження домену та діапазону

У RDFS ви можете визначити обмеження домену та діапазону властивостей за допомогою властивостей “rdfs:domain” і “rdfs:range” відповідно.

Властивість "rdfs:domain" використовується для визначення класу чи класів ресурсів, до яких можна застосувати властивість. Наприклад, наведений нижче оператор RDF визначає властивість hasAuthor, яку можна застосовувати лише до ресурсів, які є екземплярами класу Book:

```
<rdf:Property rdf:about="http://example.org/ontology/hasAuthor">
    <rdfs:label>Has Author</rdfs:label>
    <rdfs:comment>The author of a book.</rdfs:comment>
    <rdfs:domain rdf:resource="http://example.org/ontology/Book"/>
</rdf:Property>
```

У цьому прикладі властивість “rdfs:domain” визначає, що властивість “hasAuthor” можна застосовувати лише до ресурсів, які є екземплярами класу “Book”.

Властивість "rdfs:range" використовується для визначення класу або класів ресурсів, які можуть бути значенням властивості. Наприклад, наступний оператор RDF визначає властивість "hasPublisher", яка може мати лише значення, які є екземплярами класу "Publisher":

```
<rdf:Property
rdf:about="http://example.org/ontology/hasPublisher">
    <rdfs:label>Has Publisher</rdfs:label>
```

```
    <rdfs:comment>The publisher of a book.</rdfs:comment>
    <rdfs:range
rdf:resource="http://example.org/ontology/Publisher"/>
</rdf:Property>
```

У цьому прикладі властивість “rdfs:range” визначає, що властивість “hasPublisher” може мати лише значення, які є екземплярами класу “Publisher”.

Використовуючи обмеження домену та діапазону в RDFS, ви можете переконатися, що ваші дані є правильно сформованими та послідовними. Вказавши класи, які мають відношення до кожної властивості, ви також можете полегшити міркування про зв’язки між ресурсами у ваших даних.

Правила виведення

Можна визначити власні правила виведення в RDFS за допомогою механізму виведення RDFS. RDFS надає набір вбудованих правил виведення, які дозволяють виводити додаткові трійки на основі наявних трійок у вашому RDF-графіку. Ці правила виведення базуються на словнику RDFS і включають правила для виведення зв’язків підкласів, обмежень домену та діапазону та транзитивних властивостей.

Однак, якщо вбудованих правил виведення недостатньо для ваших потреб, ви можете визначити власні правила виведення за допомогою нотації виведення SPARQL (SPIN) або формату обміну правилами (RIF). SPIN − це мова на основі SPARQL для визначення правил виведення, тоді як RIF − це мова правил більш загального призначення, яку можна використовувати для визначення правил для широкого діапазону програм.

Використовуючи користувацькі правила виведення, ви можете визначати додаткові зв’язки між ресурсами у вашому RDF-графі, отримувати нову інформацію з наявних даних або накладати обмеження на дані. Наприклад, ви можете визначити спеціальне правило висновку, яке визначає зв’язок “hasChild” між двома ресурсами на основі їх зв’язку “hasParent”:

```
PREFIX : <http://example.org/ontology/>
CONSTRUCT {
  ?parent :hasChild ?child .
}
WHERE {
  ?parent :hasParent ?child .
}
```

У цьому прикладі правило використовує зв’язок “hasParent”, щоб визначити зв’язок “hasChild” між двома ресурсами.

Визначення настроюваних правил виведення в RDFS може бути потужним інструментом для міркувань про ваші дані RDF і отримання нових ідей з них. Однак для цього потрібно добре розуміти механізм виведення RDFS і мови правил, які використовуються для визначення правил.

Міркування

Міркування (*reasoning)* в онтологіях є процесом використання логічних і логічних методів висновків, щоб робити висновки, робити умовиводи та отримувати нові знання з явної та неявної інформації, представленої в онтології. Міркування дозволяє нам використовувати структуровану природу онтологій для виявлення прихованих зв’язків, перевірки інформації та відповідей на запити систематичним і автоматизованим способом.

Міркування в онтологіях передбачає застосування логічних правил, аксіом і механізмів висновку до знань, закодованих в онтології. Типовий механізм автоматичного міркування містить засоби для висновків, перевірки узгодженості, класифікації, висновки про властивості, побудову транзитивних висновків, перевірку втілення. перевірку повноти, механізм відповіді на запити, виконання правил.

Висновки включають використання встановлених логічних правил для виведення нової інформації з існуючих даних онтології. Наприклад, якщо

онтологія стверджує, що “всі люди смертні” і “Сократ є людиною”, міркування можуть зробити висновок, що “Сократ смертний”.

Перевірка узгодженості: міркування використовується для перевірки того, що інформація в онтології є логічно узгодженою та не призводить до суперечливих висновків. Якщо онтологія містить твердження, які суперечать одне одному, міркування виявлять ці невідповідності.

Класифікація: онтології часто визначають ієрархічні зв’язки між поняттями. Міркування може класифікувати екземпляри (індивідууми) у відповідні класи на основі відносин і аксіом, визначених в онтології. Наприклад, якщо онтологія визначає “Птах” як підклас “Тварини”, а екземпляр “Робін” має характеристики птаха, міркування класифікують “Робіна” як тварину.

Висновки про властивості: міркування можуть вивести зв’язки властивостей між екземплярами на основі визначених аксіом властивостей. Наприклад, якщо онтологія стверджує, що “hasParent” є зворотним до “isParentOf”, міркування можуть автоматично зробити висновок, що якщо A є батьківським для B, то B є дочірнім для A.

Транзитивні висновки: транзитивні зв’язки можна зробити за допомогою міркувань. Якщо онтологія стверджує, що “A є батьком B”, а “B є батьком C”, міркування можуть зробити висновок, що “A є батьком C”.

Перевірка втілення: Міркування може визначити, чи певне твердження випливає (логічно випливає) з аксіом і правил, визначених в онтології. Це допомагає підтвердити правильність тверджень онтології.

Перевірка повноти: міркування можуть ідентифікувати відсутні зв’язки або інформацію в онтології на основі визначених обмежень і аксіом. Він може запропонувати потенційні відносини, які повинні існувати, але не вказано явно.

Відповіді на запити: міркування дозволяє онтології відповідати на складні запити, використовуючи свою логічну структуру та зв’язки. Наприклад, на такий запит, як “Знайти всіх ссавців, які можуть літати”, можна відповісти, міркуючи через ієрархію онтології та зв’язки властивостей.

Виконання правил: деякі онтології містять правила або вирази **SWRL** ***(мова правил семантичної мережі)***, які визначають конкретні умови та висновки. Міркування може застосовувати ці правила для отримання нової інформації або перевірки наявних даних.

Загалом міркування в онтологіях дає змогу онтології виходити за межі свого явного змісту та використовувати свою логічну структуру для надання глибшого розуміння, перевірки знань і підтримки процесів прийняття рішень. Це фундаментальний аспект онтологічної інженерії та представлення знань.

Програмні інструменти для виконання автоматичного міркування

Відомими програмними інструменти для виконання автоматичного міркування є **Pellet, HermiT, Protégé, Fact++**. Кожен інструмент має свої сильні сторони, особливості та можливості.

Pellet

Pellet − це високооптимізований і широко використовуваний резонер з відкритим кодом для мови веб-онтології (OWL). Він підтримує профілі OWL DL і OWL 2. Pellet пропонує ефективні алгоритми міркування, підтримку завдань міркування OWL та інтеграцію з різними середовищами розробки онтологій. Його перевагами є високоефективне міркування, підтримка профілів міркування OWL 2, підтримка міркування на основі правил і сумісність з різними форматами онтології. Pellet може погано масштабуватися для надзвичайно великих онтологій, а його підтримка складних міркувань на основі правил може бути не такою великою, як деякі інші інструменти.

HermiT

HermiT − це ще один популярний аргумент з відкритим кодом для онтологій OWL. Він відомий своїми високопродуктивними можливостями міркування. HermiT підтримує різні профілі OWL 2 і пропонує оптимізовані алгоритми для класифікації та перевірки узгодженості. Сильною стороною HermiT є швидке та ефективне міркування, гарна масштабованість, сумісність із різними редакторами онтологій та API. Він має обмежену підтримку певних розширених завдань міркування, і він може не працювати з дуже великими онтологіями так ефективно, як деякі інші інструменти.

API OWL

OWL API − це API на основі Java для роботи з онтологіями OWL. Він забезпечує структуру для створення, аналізу, маніпулювання та міркування за допомогою онтологій. API OWL сам по собі не виконує міркування, але надає інтерфейси для інтеграції з різними аргументами, включаючи Pellet і HermiT. Його перевагою є стандартизований спосіб взаємодії з різними міркуваннями, підтримка різних профілів OWL. Він дозволяє розробникам створювати власні додатки для міркування. Але розробникам потрібно інтегрувати зовнішній аргумент для виконання завдань міркування.

Protégé:

Protégé − це широко використовуваний редактор онтології, який також підтримує онтологічне міркування через інтеграцію з резонерами, такими як Pellet, HermiT та інші.

Protégé надає зручний інтерфейс для розробки онтологій і дозволяє користувачам виконувати завдання міркування в онтологіях за допомогою інтегрованих резонерів.

Protégé простий у використанні інтерфейс для моделювання онтології, підтримує кілька реализацій міркувань і дозволяє інтерактивно міркувати.

На продуктивність аргументації може вплинути використана реалізація міркування, і він може не забезпечувати такий самий рівень оптимізації, як спеціальні інструменти міркування.

Fact++:

Fact++ − це аргументація для онтологій OWL, яка зосереджена на високопродуктивному міркуванні. Fact++ підтримує OWL 2 і забезпечує ефективні алгоритми для класифікації та перевірки узгодженості. Забезпечує високу ефективність аргументації, сумісність з різними форматами онтологій та інтеграція з різними редакторами онтологій. Fact++ може не підтримувати деякі розширені завдання міркування, які виконують інші міркувачі.

Вибираючи інструмент міркування, важливо враховувати такі фактори, як складність вашої онтології, завдання міркування, які вам потрібно виконати, легкість інтеграції та підтримка спільноти.

Types of reasoning: consistency, classification, inference.

Reasoning engines like Pellet and HermiT.

Перевірка онтології на коректність

 # Перевірка онтології на коректність

Необхідність бути впевненим у тому, що онтологія правильно формалізована.

- за допомогою **редакторів онтологій** (використовується машина міркування для перевірки логічної несуперечливості)
- за допомогою **RDF Validator** у разі якщо важливо перевірити коректність RDF формалізації
- за допомогою **OOPS! - Ontology Pitfalls Scanner** − визначення аномалій та помилок (логічних, структурних, повноти представлення метаданих)

Міркування (reasoning) в перевірці онтологій відноситься до процесу використання автоматизованих методів логічного міркування для оцінки

узгодженості, правильності та повноти онтологій. Перевірка онтології має вирішальне значення для забезпечення того, що онтологія точно представляє призначену область і що її можна ефективно використовувати для таких завдань, як інтеграція даних, обмін знаннями та прийняття рішень.

Методи міркування в перевірці онтології включають використання логічних правил і аксіом, визначених в онтології, для виявлення нових фактів та виявлення потенційних невідповідностей або помилок. Під час перевірки коректності онтологій виконуються:

1. **Перевірка узгодженості:** онтології часто містять кілька концепцій, властивостей і зв'язків. Міркування можна використовувати для перевірки наявності в онтології будь-яких суперечливих тверджень. Наприклад, якщо онтологія стверджує, що “Усі люди смертні” і “Деякі люди безсмертні”, міркування може виявити цю невідповідність.
2. **Перевірка ієрархії класів:** Онтології зазвичай визначають ієрархічні зв’язки між класами (концепціями). Міркування може допомогти перевірити, чи відповідає ієрархія класів певним очікуваним властивостям. Наприклад, якщо онтологія визначає, що “М’ясоїдна тварина” є підкласом “Тварини”, міркування можуть перевірити, чи узгоджується ця ієрархія з іншими фактами в онтології.
3. **Твердження про властивості:** міркування можна використовувати для висновку про наявність додаткових властивостей на основі визначених аксіом. Якщо онтологія вказує, що “всі ссавці мають серце”, міркування можуть автоматично зробити висновок, що “у собак є серце” і “у котів є серце”.
4. **Перевірка домену та діапазону властивостей:** Онтології часто визначають обмеження домену та діапазону для властивостей. Міркування може допомогти перевірити, чи фактичні екземпляри онтології дотримуються цих обмежень. Наприклад, якщо онтологія стверджує, що

властивість “hasParent” має домен “Person”, міркування можуть визначити випадки, коли це обмеження порушується.

5. **Перевірка екземплярів:** онтології можуть мати конкретні екземпляри класів. Міркування може перевірити, чи правильно класифіковані екземпляри та чи мають вони відповідні зв’язки та атрибути відповідно до аксіом онтології.
6. **Перевірка повноти:** міркування можна використовувати для виявлення відсутніх або неявних зв’язків у онтології. Наприклад, якщо онтологія визначає, що “Люди є тваринами”, міркування можуть припустити, що “Люди дихають” на основі ширших знань про те, що зазвичай дихають тварини.
7. **Виявлення неявних знань:** міркування можуть вивести знання, які не викладені явно в онтології, але можуть бути логічно виведені з існуючих аксіом. Це може допомогти у збагаченні вмісту онтології.
8. **Перевірка правил:** деякі онтології містять правила, які визначають певні умови та висновки. Міркування може гарантувати, що ці правила визначені належним чином і що зроблені висновки узгоджуються з онтологією.

Загалом міркування під час перевірки онтології дозволяють автоматизовано оцінювати правильність і узгодженість онтології, використовуючи логічні зв’язки та обмеження, закодовані в ній. Цей процес допомагає гарантувати, що онтологія точно фіксує передбачувані знання про предметну область і на неї можна покладатися для різних завдань, пов’язаних із знаннями.

Виведення нових знань

Онтології можуть бути використані для отримання нових знань в ряді способів, в тому числі:

Class inclusion: Це найбільш основна форма міркування в онтології. Це означає, що конкретний екземпляр є членом більш загального класу. Наприклад, якщо ми знаємо, що клас “тварина” включає підклас “собака”, то можна зробити висновок, що будь-яка конкретна собака − тварина.

Property inheritance: Цей тип міркування включає в себе виведення того, що конкретний екземпляр має властивість, яка успадковується від його класу. Наприклад, якщо ми знаємо, що клас “тварина” має властивість “має хутро”, то можна зробити висновок, що у будь-якої конкретної тварини є хутро.

Logical deduction: Цей тип міркування передбачає використання логічних правил для виведення нових знань з існуючих знань. Наприклад, якщо ми знаємо, що клас “тварина” включає в себе підкласи “собака” і “кішка”, а також знаємо, що клас “собака” несуглобовий від класу “кішка”, то можна зробити висновок, що жодна тварина не може бути і собакою, і кішкою.

Reasoning about relationships: Онтології також можуть бути використані для міркування про відносини між поняттями. Наприклад, ми можемо використовувати онтологію для того, щоб вважати, що зв’язок є “частиною” є перехідним, тобто якщо A є частиною B і B є частиною C, то A також є частиною C.

Це лише кілька способів, якими онтології можуть бути використані для отримання нових знань. Конкретний тип міркування, який може бути використаний, залежить від онтології та знань, які вона представляє.

Ось деякі приклади того, як онтології використовуються для отримання нових знань в реальних додатках:

− у медичній сфері онтології використовуються для представлення знань про хвороби, симптоми і лікування. Ці знання потім можуть бути

використані, щоб міркувати про ймовірність конкретного діагнозу, кращий курс лікування, і потенційні побічні ефекти ліків;

- у правовій сфері онтології використовуються для представлення знань законів, правил і прецедентів. Ці знання потім можуть бути використані для того, щоб міркувати про застосовність конкретного закону до конкретної ситуації, потенційні наслідки порушення закону, і кращий спосіб захистити від юридичного виклику;
- у виробничій сфері онтології використовуються для представлення знань про продукти, компоненти та виробничі процеси.

Ці знання потім можуть бути використані, щоб міркувати про кращий спосіб розробки продукту, потенційні ризики виробничого процесу, і кращий спосіб усунення проблеми.

Спеціальні правила міркування

Спеціалізовані правила міркування є типом правила, які спеціально розроблені для конкретного домену або застосування. Вони, як правило, більш ефективні, ніж загальні правила міркування, тому що вони адаптовані до конкретних знань, які представлені в онтології.

Наприклад, в медичній сфері може бути спеціальне правило міркування, яке дозволяє міркуванню зробити висновок, що пацієнт знаходиться під ризиком конкретного захворювання, враховуючи його вік, стать і історію хвороби. Це правило було б більш ефективним, ніж правило загального призначення, яке просто настоюється, що будь−який пацієнт, який має певний симптом, знаходиться під загрозою захворювання.

Спеціалізовані правила міркування можуть бути використані для поліпшення продуктивності онтологічних систем в ряді способів. Вони можуть:

- зменшити кількість часу, який потрібно, щоб міркувати про конкретну проблему.

- підвищити точність результатів процесу міркування.
- зробити процес міркування більш масштабованим, так що він може бути застосований до великих і більш складних онтологій.

Спеціальні правила міркування можуть бути створені вручну або автоматично. Створені вручну правила, як правило, більш точні, але вони можуть бути трудомісткими для розробки. Методи автоматичного генерування правил можуть бути використані для створення правил швидше, але вони можуть бути не такими точними.

Використання спеціалізованих правил міркування є важливою областю досліджень в галузі онтології. Оскільки онтології стають більш складними і складними, потреба в ефективних і точних методів міркування стане все більш важливою.

Ось деякі приклади спеціалізованих правил міркування:

- у медичній сфері діє правило, яке полягає в тому, що пацієнт знаходиться під загрозою певного захворювання, враховуючи його вік, стать і історію хвороби.
- у правовому домені діє правило, що настоюється, що особа винна у вчиненні злочину, з урахуванням доказів, які пред’явлені в судовому справі.
- у виробничому домені правило, що настоюється, що продукт є дефектним, враховуючи результати перевірки якості.

Способи задання правил

Ось порівняння **RuleML, SWRL, ShaCL, Jena Rules і RIF**, які є добре відомими методами опису правил:

Особливість	RuleML	SWRL	ShaCL	Jena Rules	RIF
Мета	Мова для представлення правил у машиночитаному форматі.	Мова правил для семантичної мережі, яка поєднує виразність OWL з обчислювальною ефективністю систем на основі правил.	Мова для опису обмежень на графах RDF.	Мова правил для семантичної мережі, яка базується на SWRL.	Мова правил для семантичної мережі, яка призначена для взаємодії з іншими мовами правил, такими як SWRL і RuleML.
Синтаксис	Заснований на XML.	Заснований на XML.	Заснований на RDF.	Заснований на XML.	Заснований на XML.
Виразність	Може представляти широкий спектр знань і відносин.	Може представляти широкий спектр знань і відносин.	Може представляти широкий спектр обмежень на графіки RDF.	Може представляти широкий спектр знань і відносин.	Може представляти широкий спектр знань і відносин.
Оперативна сумісність	Не сумісний з іншими мовами правил.	Сумісна з іншими мовами правил, такими як RuleML.	Не сумісний з іншими мовами правил.	Сумісна з іншими мовами правил, такими як SWRL.	Сумісні з іншими мовами правил, такими як SWRL і RuleML.
Підтримка	Підтримується різними інструментами та додатками.	Підтримується різними інструментами та додатками.	Підтримується різними інструментами та додатками.	Підтримується різними інструментами та додатками.	Підтримується різними інструментами та додатками.

Як бачите, всі мови мають свої сильні і слабкі сторони. Краща мова для конкретного додатка буде залежати від конкретних вимог програми.

Ось більш детальне порівняння п’яти мов:

- **RuleML** − це мова правил загального призначення, яка може бути використана для представлення широкого спектру знань і зв’язків. Вона не сумісна з іншими мовами правил, що може ускладнити використання в деяких додатках.
- **SWRL** − це мова правил для семантичної мережі, яка поєднує виразність OWL з обчислювальною ефективністю систем на основі правил. Він сумісний з іншими мовами правил, такими як RuleML.
- **ShaCL** − мова для опису обмежень на графах RDF. Вона не сумісна з іншими мовами правил, але може бути використана в поєднанні з іншими

мовами правил для визначення обмежень на знання, які представлені в основі правил.

- **Jena Rules** − це мова правил для семантичної мережі, яка базується на SWRL. Він сумісний з іншими мовами правил, такими як SWRL.
- **RIF: RIF** є мовою правил для семантичної мережі, яка призначена для взаємодії з іншими мовами правил, такими як SWRL і RuleML. Він не такий виразний, як деякі інші мови, але його легше вивчити і використовувати.

Загальний робочий процес, який використовує RuleML та онтологію

Ось загальний робочий процес, який використовує RuleML та онтологію:

1. **Define the ontology (*Визначити онтологію*).** Перший крок – визначити онтологію. Онтологія є формальним представленням знань у сфері інтересів. Це повинно бути зроблено з використанням такої мови, як OWL.
2. **Define the rules (*Визначте правила*).** Наступним кроком є визначення правил. Правила використовуються, щоб міркувати про знання, які представлені в онтології. Вони можуть бути написані за допомогою такої мови, як RuleML.
3. **Load the ontology and rules into a rule engine (*Завантажте онтологію та правила в рушій правил*).** Онтологія і правила повинні бути завантажені в рушій правил. Рушій правил − це програмна система, яка може міркувати про правила.
4. **Query the rule engine (*Запит рушія правил*).** Рушій правил можна поставити запитання, щоб відповісти на питання про знання в області інтересів.

5. **Update the ontology and rules (*Оновлення онтології та правил*).** Оскільки знання в області інтересів змінюється, онтологія та правила можуть бути оновлені.

RuleML і онтологія можуть бути в одному файлі. Це часто робиться на практиці, так як це може полегшити управління знаннями і відносинами, які представлені.

Існує кілька різних способів об’єднати RuleML і онтологію в одному файлі. Одним із способів є використання мови, яка підтримує як RuleML, так і онтологію, наприклад SWRL. SWRL − мова правил, заснована на OWL, тому його можна використовувати для представлення як правил, так і онтологій.

Ще один спосіб об’єднати RuleML і онтологію − це використовувати окремий файл для кожного. Файл онтології може бути написаний мовою OWL, а файл правил може бути написаний мовою, наприклад RuleML. Два файли можуть бути об’єднані за допомогою інструменту або бібліотеки, яка підтримує обидві мови.

Кращий спосіб об’єднати RuleML і онтологію буде залежати від конкретного застосування. Якщо додаток вимагає здатності міркувати про правила і онтології, то така мова, як SWRL, може бути хорошим вибором. Якщо додаток вимагає тільки можливість представляти правила і онтології, то окремі файли можуть бути кращим вибором.

Програмні інструменти

Ось деякі з інструментів і бібліотек, які підтримують RuleML і OWL:

- **HermiT: HermiT** Відлюдник є розмірковувачем, який підтримує СОВУ і SWRL. Це популярний вибір для міркування про онтології і правила.
- **Protege: Protege** − редактор онтології, який підтримує OWL і SWRL. Це популярний вибір для створення та редагування онтологій.

- **Jena: Jena** є основою для роботи з RDF і OWL. Він підтримує RuleML і SWRL через бібліотеку правил Jena.
- **Drools** − система управління бізнес−правилами, яка підтримує RuleML і OWL2RL. Це популярний вибір для розробки додатків на основі правил.
- **Apache Stanbol** − платформа для створення семантичних додатків. Він підтримує RuleML і OWL через репозиторій правил Stanbol.
- Це лише деякі з багатьох інструментів і бібліотек, які підтримують RuleML і OWL. Кращий інструмент або бібліотека для конкретного додатка буде залежати від конкретних вимог програми.

Детальні описи мов визначення правил

RuleML

RuleML розшифровується як Rule Markup Language. Це мова розмітки для представлення правил у машинозчитуваному форматі. RuleML заснований на XML і може бути використаний для представлення як правил вперед (знизу вгору), так і назад (зверху вниз).

RuleML використовується в різних додатках, включаючи:

- **Rule-based systems:** Rule-based systems − це системи, які використовують правила для прийняття рішень. RuleML може бути використаний для представлення правил в системі на основі правил, що полегшує розробку і підтримку системи.
- **Ontology engineering:** Ontology engineering − це процес створення та підтримки онтологій. RuleML може бути використаний для представлення правил, які використовуються для визначення семантики онтології.
- **Knowledge representation:** Knowledge representation − це процес представлення знань у машиночитаному форматі. RuleML може використовуватися для представлення знань у різних форматах, включаючи онтології, логічні програми та дерева рішень.

- **Data mining:** Data mining − це процес виявлення закономірностей в даних. RuleML може бути використаний для представлення правил, які виявляються алгоритмом інтелектуального аналізу даних.

RuleML є потужним інструментом для представлення правил у машинозчитуваному форматі. Він використовується в різних додатках і постійно розробляється для задоволення потреб нових додатків.

Ось деякі з особливостей RuleML:

- він заснований на XML, що дозволяє легко читати і писати.
- він підтримує як форвардних, так і відсталих правил.
- він може бути використаний для представлення різних типів правил, включаючи правила виробництва, дедуктивні правила та викраденні правила.
- він підтримує різні механізми міркування, в тому числі просування вперед, назад фарбування, і гібридні міркування.
- він розширюваний, тому нові функції можуть бути додані в міру необхідності.

RuleML є цінним інструментом для різних застосувань. Він використовується в системах на основі правил, онтології, представлення знань і видобутку даних. Оскільки потреба в системах на основі правил зростає, RuleML, ймовірно, стане ще більш широко використовуваним.

Основний синтаксис RuleML виглядає наступним чином:

```
<rule>
<head>
<conditions>
 ...
</conditions>
</head>
<body>
<actions>
 ...
</actions>
</body>
</rule>
```

<rule> − елемент є кореневим елементом правила RuleML. Елемент <head> містить умови правила, а елемент <body> містить дії, які виконуються при виконанні умов.

<conditions> − елемент може містити будь-яку кількість елементів <condition>. <condition> − елемент є предикатом, який оцінюється як істинний або хибний. <actions> − елемент може містити будь-яку кількість елементів <action>. <action> − елемент − це інструкція, яка виконується, коли виконуються умови.

Умови і дії правила можуть виражатися різними способами. Нижче наведені деякі з найбільш поширених способів вираження умов і дій:

- **Literals: Literal (*літералів*)** є простим значенням, таким як число (number), рядок (string) або Boolean value (булеве значення).
- **Variables: Variable (*змінні*)** є заповнювачем для value.
- **Predicates: Predicate (*предикати*)**− це функція, яка приймає один або кілька аргументів і повертає Boolean value.
- **Operators: Operators** використовуються для об’єднання literals, variables, та predicates для формування більш складних виразів.

Ось приклад простого правила RuleML:

```
<rule>
<head>
<conditions>
<condition>
<predicate>age > 18 </predicate>
</condition>
</conditions>
</head>
<body>
<actions>
<action>grant_ding_license</action>
</actions>
</body>
</rule>
```

Це правило стверджує, що якщо вік людини перевищує 18 років, то їм повинні бути надані водійські права.

RuleML пов’язаний з онтологією

RuleML пов’язаний з онтологією і СОВОЮ кількома способами.

По-перше, онтології можуть бути використані для представлення понять і відносин в області знань. Потім правила можуть бути використані для визначення семантики онтології, або для пояснення про знання, яке представлено в онтології.

Наприклад, онтологія для галузі медицини може визначити поняття “хвороба”, “симптом” і “лікування”. Потім правила можуть бути використані для визначення взаємозв’язку між цими поняттями, наприклад, той факт, що хвороба може мати кілька симптомів, або що лікування може бути використано для лікування хвороби.

По-друге, OWL є мовою для представлення онтологій. RuleML може бути використаний для представлення правил, які є специфічними для конкретної онтології, або які можуть бути використані для пояснення онтологій в цілому.

Наприклад, правило в OWL може стверджувати, що якщо клас **A** є підкласом класу **B**, то будь-який екземпляр класу A також є екземпляром класу B. це правило може бути використано для пояснення зв’язків між класами в онтології OWL.

По-третє, RuleML можна використовувати для об’єднання онтологій з різних доменів. Це може бути корисним для таких завдань, як інтеграція знань і міркування про міждоменні знання.

Наприклад, онтологію для галузі медицини можна поєднати з онтологією для галузі права, щоб міркувати про правові наслідки лікування.

Ось приклад правила, яке поєднує RuleML і OWL:

```
<rule>
<head>
<conditions>
<condition>
<predicate>person(?x)</predicate>
</condition>
<condition>
<predicate>GraduatedBiд (?x,? Y)</predicate>
</condition>
</conditions>
</head>
<body>
<actions>
<action>EligigbleForJob(?x)</action>
</actions>
</body>
</rule>
```

Це правило стверджує, що якщо особа є екземпляром класу `Person` і вона закінчила університет, то вона має право на роботу.

Елемент `<head>` правила містить умови, які мають бути виконані для запуску правила. У цьому випадку дві умови полягають у тому, що змінна **`?x`** має бути екземпляром класу `Person` і що **`?x`** має закінчити університет.

Елемент `<body>` правила містить дії, які виконуються при виконанні умов. У цьому випадку дія полягає у встановленні змінної **`?x`** як екземпляр класу `EligibleForJob` .

Предикати `Person()`, `GraduatedFrom()` та `EligibleForJob()` визначені в онтології OWL.

Це лише простий приклад того, як можна поєднати RuleML і OWL. Більш складні правила можуть бути створені для представлення ширшого діапазону знань і відносин.

SWRL

SWRL означає мову правил семантичної мережі. Це мова правил для семантичної мережі, яка поєднує виразність OWL з обчислювальною ефективністю систем, заснованих на правилах.

Правила SWRL написані в підмножині Datalog, мови логічного програмування. Datalog є декларативною мовою, що означає, що правила визначають зв’язки між фактами, не вказуючи, як обчислювати результати. Це робить правила SWRL легшими для написання та розуміння, ніж інші мови правил, такі як Prolog.

Правила SWRL можна використовувати для представлення широкого діапазону знань і відносин. Вони можуть бути використані для визначення семантики онтологій, міркувань про знання, які представлені в онтологіях, і для інтеграції онтологій з різних областей.

SWRL є потужним інструментом для представлення та міркування про знання в семантичній мережі. Він використовується в різних сферах застосування, зокрема:

- **Ontology engineering (*розробка онтологій*):** правила SWRL можна використовувати для визначення семантики онтологій і міркування про знання, які представлені в онтологіях.
- **Knowledge representation (*представлення знань*) :** правила SWRL можна використовувати для представлення знань у різних форматах, включаючи онтології, логічні програми та дерева рішень.
- **Data mining (*інтелектуальний аналіз даних*):** правила SWRL можна використовувати для представлення правил, які виявляє алгоритм інтелектуального аналізу даних.
- **Rule−based systems (*системи на основі правил*) :** правила SWRL можна використовувати для створення систем на основі правил, які можуть міркувати про знання.
- **Natural language processing (*обробка природної мови*) :** правила SWRL можна використовувати для представлення семантики природної мови та міркування про значення тексту.

SWRL є перспективною технологією для семантичної мережі. Це потужний інструмент для представлення та міркування про знання, який використовується в різноманітних програмах.

Ось деякі з особливостей SWRL:

- він заснований на Datalog, що робить його декларативною мовою.
- його можна використовувати для представлення широкого діапазону знань і відносин.
- ефективно обчислювати.
- він сумісний з OWL, провідною мовою онтології.
- він підтримується різними інструментами та програмами.

SWRL є цінним інструментом для різноманітних програм. Він використовується в розробці онтології, представленні знань, аналізі даних, системах на основі правил і обробці природної мови. Оскільки потреба в представленні знань і міркуваннях зростає, SWRL, ймовірно, стане ще більш широко використовуватися.

Синтаксис SWRL такий:

```
rule ::= head :- body
head ::= atom
body ::= atom | atom , body
atom ::= predicate(term+)
term ::= variable | constant | uri
```

Правила є кореневим елементом правила SWRL . Елемент `head` містить умови правила, а елемент `body` містить дії, які виконуються, коли умови виконуються .

`atom` є основною одиницею правила SWRL . Він складається з predicate та list of terms (*списку термінів*). Predicate − це функція, яка приймає один або кілька термінів як вхідні дані та повертає логічне значення. Терміни можуть бути змінними, константами або URI.

Variable element є заповнювачем для значення. Константним елементом є literal value, наприклад number, string, або Boolean value. Елемент `uri` − це уніфікований ідентифікатор ресурсу (URI), який є унікальним ідентифікатором ресурсу.

Ось приклад простого правила SWRL:

```
head :- Person(?x), GraduatedFrom(?x, ?y)
```

Це правило стверджує, що якщо особа є екземпляром класу **`Person`** і вона закінчила університет, то вона має право на роботу.

Головний `head element` містить умови, які мають бути виконані для запуску правила. У цьому випадку дві умови полягають у тому, що змінна **`?x`** має бути екземпляром класу `Person` і що `?x` має закінчити університет.

Елемент body порожній, що означає, що немає дій, які виконуються, коли умови виконуються.

ShaCL

Shapes Constraint Language (SHACL) − це стандартна мова консорціуму **World Wide Web (W3C)** для опису обмежень на RDF-графах. SHACL був розроблений для покращення рівня семантичної та технічної сумісності онтологій, виражених у вигляді RDF-графів.

SHACL використовує декларативний синтаксис для визначення обмежень на графах RDF. Обмеження можна використовувати для перевірки RDF-графів, для висновку про відсутню інформацію та для обґрунтування семантики RDF-графів.

SHACL є потужним інструментом для забезпечення якості та узгодженості RDF-графів. Він використовується в різних сферах застосування, зокрема:

- **Ontology engineering (*Розробка онтологій*):** SHACL можна використовувати для визначення обмежень на онтологіях, перевірки онтологій і виведення відсутньої інформації з онтологій.
- **Data integration (*Інтеграція даних*):** SHACL можна використовувати для перевірки та інтеграції даних RDF з різних джерел.
- **Knowledge representation (*Представлення знань*):** SHACL можна використовувати для представлення знань у різних форматах, включаючи онтології, логічні програми та дерева рішень.
- **Rule−based systems (*Системи на основі правил*):** SHACL можна використовувати для створення систем на основі правил, які можуть міркувати про RDF-графи.

- **Natural language processing (*Обробка природної мови*):** SHACL можна використовувати для представлення семантики природної мови та міркування про значення тексту.

Якщо вам цікаво дізнатися більше про SHACL, є багато ресурсів, доступних в Інтернеті. W3C SHACL Primer − гарне місце для початку.

Ось деякі з переваг використання SHACL:

- це декларативна мова, що робить її легкою для розуміння та використання.
- він розширюваний, тому за потреби можна додавати нові обмеження.
- він підтримується різними інструментами та програмами.
- це стандарт W3C, що означає, що він широко прийнятий і підтримується.

Синтаксис SHACL базується на RDF. Він використовує комбінацію RDF-трійок і конструкцій SHACL для визначення обмежень на RDF-графах.

Основною одиницею обмеження SHACL є форма. Форма − це ресурс RDF, який описує набір обмежень. Обмеження можна застосувати до будь-якого ресурсу RDF, який є екземпляром цільового класу форми.

Фігура визначається за допомогою властивості `sh:shape`. Властивість `sh:shape` приймає як значення ресурс RDF. Цей ресурс RDF є визначенням форми.

Визначення фігури може містити різноманітні конструкції SHACL, зокрема:

- **Property constraints: обмеження властивості** вказують дозволені значення для властивості.
- **Node constraints: обмеження вузлів** визначають дозволені типи вузлів.
- **Group constraints: групові обмеження** поєднують кілька обмежень в одне обмеження.

- **Validation reports: у звітах про перевірку** вказуються помилки та попередження, про які слід повідомляти, коли обмеження порушується.

Ось приклад простої форми SHACL:

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
<http://example.org/PersonShape>
a sh:Форма ;
sh:targetClass <http://example.org/Person> ;
sh:власність [
sh:шлях rdfs:мітка ;
sh:minLength 2 ;
] .
```

Ця фігура визначає форму для класу `Person`. Фігура визначає, що властивість `rdfs:label` повинна мати мінімальну довжину 2 символи.

Jena Rules

Jena Rules − це мова правил для семантичної мережі, яка базується на Semantic Web Rule Language (SWRL). Jena Rules реалізовано у фреймворку Apache Jena, який є популярним фреймворком Java для роботи з даними RDF.

Jena Rules можна використовувати для представлення широкого кола знань і відносин. Вони можуть бути використані для визначення семантики онтологій, міркувань про знання, які представлені в онтологіях, і для інтеграції онтологій з різних областей.

Jena Rules − потужний інструмент для представлення та міркування про знання в семантичній мережі. Він використовується в різних сферах застосування, зокрема:

- **Ontology engineering:** Jena Rules можна використовувати для визначення семантики онтологій і міркування про знання, які представлені в онтологіях.

- **Knowledge representation:** Jena Rules можна використовувати для представлення знань у різних форматах, включаючи онтології, логічні програми та дерева рішень.
- **Data mining:** Jena Rules можна використовувати для представлення правил, які виявляє алгоритм інтелектуального аналізу даних.
- **Rule−based systems:** Jena Rules можна використовувати для створення систем на основі правил, які можуть міркувати про знання.
- **Natural language processing:** Jena Rules можна використовувати для представлення семантики природної мови та міркування про значення тексту.

Ось деякі з особливостей Jena Rules:

- він заснований на SWRL, що робить його потужною та виразною мовою правил.
- він реалізований у середовищі Apache Jena, яке є популярним і добре підтримуваним.
- він розширюваний, тому за потреби можна додавати нові функції.
- він підтримується різними інструментами та програмами.

Синтаксис Jena Rules подібний до синтаксису SWRL. Він використовує комбінацію RDF-трійок і конструкцій Jena Rules для визначення правил.

Основною одиницею Jena Rules є елемент правила. Елемент правила складається з голови та тіла. Заголовок визначає умови, які повинні бути виконані для запуску правила. Орган визначає дії, які виконуються, коли умови виконуються.

Заголовок і тіло правила можуть містити різноманітні конструкції правил Jena, зокрема:

- **Triple patterns:** потрійні шаблони вказують потрійні RDF, які мають бути присутніми в базі знань, щоб запрацювало правило.
- **Variables:** Змінні використовуються для представлення невідомих значень.
- **Predicates:** Предикати використовуються для представлення зв’язків між сутностями.
- **Operators:** оператори використовуються для поєднання потрійних шаблонів і змінних для формування складніших виразів.

Ось приклад простого правила Jena Rules:

```
rule r1
 when
 triple (?x rdf:type Person)
 then
 triple (?x foaf:knows ?y)
```

Це правило стверджує, що якщо в базі знань є трійка RDF, яка стверджує, що **`?x`** є екземпляром класу `Person`, тоді в базі знань також має бути трійка RDF, яка стверджує, що **`?x`** знає **`?y`** .

RIF

RIF означає формат обміну правилами. Це мова правил для семантичної мережі, яка розроблена для взаємодії з іншими мовами правил, такими як SWRL і RuleML.

RIF є декларативною мовою, що означає, що вона визначає зв’язки між фактами, не вказуючи, як обчислювати результати. Це робить правила RIF легшими для написання та розуміння, ніж інші мови правил, такі як Prolog.

Правила RIF можна використовувати для представлення широкого діапазону знань і відносин. Вони можуть бути використані для визначення семантики онтологій, міркувань про знання, які представлені в онтологіях, і для інтеграції онтологій з різних областей.

RIF є потужним інструментом для представлення та міркування про знання в семантичній мережі. Він використовується в різних сферах застосування, зокрема:

- **Ontology engineering:** правила RIF можна використовувати для визначення семантики онтологій і міркування про знання, які представлені в онтологіях.
- **Knowledge representation:** правила RIF можна використовувати для представлення знань у різноманітних форматах, включаючи онтології, логічні програми та дерева рішень.
- **Data mining:** правила RIF можна використовувати для представлення правил, які виявляє алгоритм інтелектуального аналізу даних.
- **Rule−based systems:** правила RIF можна використовувати для створення систем на основі правил, які можуть міркувати про знання.
- **Natural language processing:** правила RIF можна використовувати для представлення семантики природної мови та міркування про значення тексту.

Ось деякі з особливостей RIF:

- це декларативна мова, що робить її легкою для розуміння та використання.
- він сумісний з іншими мовами правил, такими як SWRL і RuleML.
- він розширюваний, тому за потреби можна додавати нові функції.
- він підтримується різними інструментами та програмами.

Ось приклад простого правила RIF:

```
rule person-knows-person
 when
 $x rdf:type foaf:Person
 $y rdf:type foaf:Person
 $x foaf:knows $y
```

```
then
$x rdfs:subClassOf foaf:Friend
```

Це правило стверджує, що якщо в базі знань є дві трійки RDF, які стверджують, що **x** є екземпляром класу `foaf:Person`, **y** є екземпляром класу `foaf:Person`, а **x** знає **y**, тоді **x** також є екземпляром класу `foaf:Friend`.

Речення `when` правила визначає умови, які мають бути виконані для запуску правила. У цьому випадку в базі знань має бути дві трійки RDF, які відповідають умовам.

Пункт `then` правила визначає дії, які виконуються, коли умови виконуються. У цьому випадку до бази знань додається нова трійка RDF, яка стверджує, що x є екземпляром класу `foaf:Friend`.

Існує кілька інших мов визначення правил:

- **Prolog:** Prolog − це мова логічного програмування, яка використовується для представлення знань і правил. Це декларативна мова, що означає, що вона визначає зв’язки між фактами, не вказуючи, як обчислювати результати. Це робить Prolog легшим для написання та розуміння, ніж інші мови правил, такі як SWRL.
- **Datalog:** Datalog є підмножиною Prolog, оптимізованою для міркувань про бази даних. Це більш стисла та ефективна мова, ніж Пролог, але вона також менш виразна.
- **LOOM:** LOOM − це мова правил, призначена для використання з онтологіями. Він заснований на логіці опису SHOIN(D), що робить його потужною та виразною мовою.
- **DAML+OIL:** DAML+OIL − це мова правил, яка базується на мові онтології OWL. Він більше не розробляється активно, але все ще використовується в деяких програмах.

- **OWL2RL:** OWL2RL − це мова правил, яка базується на мові онтології OWL2. Він розроблений, щоб бути більш ефективним, ніж SWRL, але він також менш виразний.

Це лише деякі з багатьох доступних мов визначення правил. Найкраща мова для конкретної програми залежатиме від конкретних вимог програми.

OWL 1

OWL 1 Lite

OWL Lite − це підмова мови веб-онтології (OWL), яка надає підмножину конструкцій OWL, що робить її простішою та зручнішою у використанні версією OWL. Ось деякі з ключових функцій OWL Lite:

1. Ієрархія класів: OWL Lite підтримує визначення ієрархій класів, де класи можуть бути організовані в ієрархічну структуру на основі їхніх зв’язків підкласів і суперкласів.
2. Ієрархія властивостей: OWL Lite підтримує визначення властивостей об’єктів та їх ієрархій, які можна використовувати для представлення зв’язків між окремими особами.
3. Обмеження кардинальності: OWL Lite підтримує визначення обмежень кардинальності властивостей, які можна використовувати для визначення точної кількості значень, які властивість може мати для певного класу.
4. Рівність: OWL Lite підтримує визначення рівності між особами, яке можна використовувати для вказівки на те, що дві особи еквівалентні.
5. Типи даних: OWL Lite підтримує обмежену кількість типів даних, таких як цілі числа, рядки та логічні значення, для визначення властивостей та осіб.
6. Зворотні властивості: OWL Lite підтримує визначення зворотних властивостей, які можна використовувати для представлення зворотних зв’язків між індивідами.

7. Обмеження: OWL Lite підтримує використання обмежень для визначення класів, які мають певні характеристики, наприклад, клас усіх індивідів, які мають певне значення властивості.

Загалом, OWL Lite надає спрощену версію OWL, яка підходить для багатьох програм, де достатньо менш складної мови онтології.

OWL 1 DL

OWL 1 DL (Description Logic) розширює OWL 1 Lite, додаючи до мови більш виразні конструкції. Ось деякі з ключових способів, якими OWL 1 DL розширює OWL 1 Lite:

1. **Описи складних класів:** OWL 1 DL дозволяє визначати описи складних класів за допомогою таких конструкцій, як перетин, об’єднання та доповнення. Це дозволяє більш точні та гнучкі визначення класів.
2. **Кваліфіковані обмеження кардинальності:** OWL 1 DL дозволяє визначити кваліфіковані обмеження кардинальності, які вказують точну кількість значень, які властивість може мати для певного класу на основі певної умови. Наприклад, кваліфіковане обмеження кардинальності може вказувати, що особа має рівно двох дітей, які є дівчатками.
3. **Ієрархії ролей:** OWL 1 DL дозволяє визначати ієрархії ролей, які визначають ієрархію стосунків між екземплярами. Це дозволяє точніше та гнучкіше відображати стосунки між окремими екземплярами.
4. Розбіжність: OWL 1 DL допускає визначення розбіжності між класами, що вказує на те, що два класи не мають спільних екземплярів. Це можна використовувати, щоб уникнути суперечливих або непослідовних визначень класів.

Загалом, додаткові конструкції в OWL 1 DL роблять його більш потужною та гнучкою мовою для моделювання онтології, ніж OWL 1 Lite.

Однак підвищена виразність також супроводжується більшою складністю та обчислювальними витратами.

OWL 1 Full

OWL 1 DL (Description Logic) і OWL 1 Full − це дві різні версії мови OWL з різною виразною потужністю та обчислювальною складністю. Ось деякі з ключових відмінностей між двома версіями:

1. Виразність: OWL 1 Full є більш виразним, ніж OWL 1 DL, що дозволяє представляти більш складні онтології. Наприклад, OWL 1 Full дозволяє використовувати довільні графіки RDF (Resource Description Framework) як визначення класів і властивостей, тоді як OWL 1 DL обмежує визначення класів і властивостей підмножиною RDF.
2. Обчислювальна складність: OWL 1 DL розроблено таким чином, щоб бути зручним для автоматизованих міркувань і класифікації, тоді як OWL 1 Full дозволяє виконувати більш складні завдання, які можуть бути неможливими з точки зору обчислення. Як наслідок, OWL 1 Full, як правило, важче міркувати, ніж OWL 1 DL.
3. Типи аксіом: OWL 1 Full містить додаткові типи аксіом, окрім тих, що доступні в OWL 1 DL, включаючи аксіоми ланцюжків властивостей, рефлексивності та аксіоми симетрії. Це дозволяє створювати більш виразні онтології, але також збільшує складність міркувань.
4. Перевірка узгодженості: OWL 1 DL забезпечує сильніші гарантії узгодженості та вирішуваності, ніж OWL 1 Full. В OWL 1 DL можна визначити узгодженість онтології та перевірити, чи вона відповідає набору обмежень. У OWL 1 Full можуть існувати нерозв’язні або суперечливі онтології, які неможливо повністю перевірити або аргументувати.

Загалом, OWL 1 DL розроблено для балансування виразної потужності з обчислювальною зручністю, що робить його корисним інструментом для моделювання онтології та автоматизованих міркувань. OWL 1 Full є більш

виразним, але також більш складним і складним з точки зору обчислень, що робить його кращим для певних типів завдань міркування або випадків використання, де потрібна більша потужність виразності.

Ось приклад, який ілюструє різницю між OWL 1 DL і OWL 1 Full.

Розглянемо наступну онтологію, яка визначає класи для тварин та їхні характеристики:

```
Ontology: AnimalOntology
Class: Animal
    SubClassOf: hasCharacteristic some Characteristic
Class: Characteristic
    SubClassOf: hasValue some xsd:string
```

Ця онтологія стверджує, що тварина характеризується певною характеристикою, і що характеристика має значення типу string. Ця онтологія дійсна як для OWL 1 DL, так і для OWL 1 Full.

Тепер розглянемо наступну модифікацію онтології, яка додає аксіому ланцюга властивостей для визначення нової властивості:

```
Ontology: AnimalOntology
Class: Animal
    SubClassOf: hasCharacteristic some Characteristic
Class: Characteristic
    SubClassOf: hasValue some xsd:string
```

ObjectProperty: hasCharacteristicValue

Characteristics: hasCharacteristic o hasValue

Ця онтологія дійсна в OWL 1 Full, але не в OWL 1 DL. Аксіома ланцюга властивостей не допускається в OWL 1 DL, оскільки вона надто складна для алгоритмів міркувань, які використовуються в OWL 1 DL. Навпаки, OWL 1 Full дозволяє використовувати ланцюжки властивостей, а також інші додаткові типи аксіом, які недоступні в OWL 1 DL.

Цей приклад ілюструє, як більш виразна сила OWL 1 Full дозволяє створювати більш складні онтології, які не можуть бути представлені в OWL 1 DL. Однак ця більша виразна сила також супроводжується більшою обчислювальною складністю, що може ускладнити міркування та класифікацію.

Аксіоми в OWL 1

Аксіоми в OWL 1 − це твердження, які декларують факти про сутності в онтології. Ось кілька прикладів аксіом в OWL 1:

1. Аксіома класу: ця аксіома стверджує, що клас A є підкласом класу B.

```
A SubClassOf B
```

Наприклад, наступна аксіома OWL 1 стверджує, що клас “Собака” є підкласом класу ”Ссавець”:

```
Dog SubClassOf Mammal
```

2. Аксіома властивості: Ця аксіома стверджує, що властивість P є підвластивістю властивості Q.

```
P SubPropertyOf Q
```

Наприклад, наступна аксіома OWL 1 стверджує, що властивість “hasFather” є підвластивістю властивості “hasParent”:

```
hasFather SubPropertyOf hasParent
```

3. Аксіома еквівалентного класу: ця аксіома стверджує, що клас A еквівалентний класу B.

```
A EquivalentTo B
```

Наприклад, наступна аксіома OWL 1 стверджує, що клас “Студент” еквівалентний класу “Особа, яка навчається на курсі”:

```
Student EquivalentTo Person and hasEnrollment some Course
```

4. Аксіома розбіжності: ця аксіома стверджує, що класи A та клас B є непересічними, тобто вони не мають спільних екземплярів.

```
DisjointClasses(A B)
```

Наприклад, наступна аксіома OWL 1 стверджує, що класи “Собака” і “Кіт” не перетинаються:

```
DisjointClasses(Dog Cat)
```

5. Індивідуальна аксіома: ця аксіома стверджує, що індивід a є екземпляром класу A.

```
a Type A
```

Наприклад, наступна аксіома OWL 1 стверджує, що індивід “Fido” є екземпляром класу “Dog”:

```
Fido Type Dog
```

Зверніть увагу, що ці приклади наведено в синтаксисі OWL 1, а OWL з тих пір оновлено до OWL 2 з деякими відмінностями в синтаксисі та додатковими функціями.

OWL 2

OWL 1 і **OWL 2** − це дві версії мови **OWL** для представлення та міркування про онтології.

Відмінності між OWL 1 та OWL 2

Виразність: OWL 2 є більш виразним, ніж OWL 1, що дозволяє представляти більш складні онтології. Наприклад, OWL 2 включає додаткові конструкції для обробки властивостей типів даних, ланцюжків властивостей, кваліфікованих обмежень кардинальності тощо.

Синтаксис: OWL 2 представляє новий синтаксис для представлення онтологій під назвою OWL 2 Functional Syntax, який є більш лаконічним і читабельним, ніж синтаксис, що використовується в OWL 1.

Типи аксіом: OWL 2 включає додаткові типи аксіом, окрім тих, що доступні в OWL 1, включаючи непересічні об’єднання, ключі та обернені властивості.

Профілі: OWL 2 представляє систему профілів, які забезпечують підмножини мови з різними рівнями обчислювальної складності та підтримкою аргументації. Профілі включають OWL 2 EL, OWL 2 QL і OWL 2 RL, які оптимізовані для використання в різних програмах і випадках використання.

Сумісність: OWL 2 має зворотну сумісність з OWL 1, тобто онтології OWL 1 можуть бути автоматично переведені в OWL 2 без втрати інформації.

Загалом, OWL 2 є більш виразною та гнучкою мовою, ніж OWL 1, з додатковими конструкціями для представлення складних онтологій та покращеним синтаксисом і підтримкою міркувань. Введення профілів також робить OWL 2 більш адаптованим до різних варіантів використання та програм. Однак підвищена складність OWL 2 також означає, що міркування та класифікація можуть бути більш складними з точки зору обчислень, ніж у OWL 1.

Визначення типів даних

Визначення типів даних: OWL 2 дозволяє визначати нові типи даних за допомогою вбудованого набору примітивних типів даних або інших визначених користувачем типів даних. Наприклад, наступна конструкція OWL 2 визначає новий тип даних під назвою "EvenInteger" як підмножину вбудованого цілочисельного типу даних:

Datatype: EvenInteger

```
SubClassOf: integer[hasValue some xsd:integer[mod 2 = 0]]
```

Це визначає “EvenInteger” як підтип цілого числа, де ціле число має модуль 2, який дорівнює 0, тобто це парне ціле число.

Обмеження типу даних: OWL 2 дозволяє обмежити властивості типу даних певними діапазонами або значеннями. Наприклад, наступна конструкція

OWL 2 обмежує властивість типу даних "hasAge" цілими значеннями від 0 до 120:

Datatype: xsd:integer

DatatypeProperty: hasAge

```
        Range:    integer[xsd:minInclusive    "0"^^xsd:integer,
xsd:maxInclusive "120"^^xsd:integer]
```

Це обмежує властивість "hasAge" цілими значеннями від 0 до 120, використовуючи вбудований цілочисельний тип даних і аспекти "minInclusive" і "maxInclusive".

Фасети типу даних: OWL 2 дозволяє використовувати додаткові фасети для обмеження типів даних, таких як “length”, “pattern” і “totalDigits”. Наприклад, наступна конструкція OWL 2 обмежує властивість типу даних "hasName" рядковими значеннями довжиною щонайменше 2 символи:

Datatype: xsd:string

DatatypeProperty: hasName

```
Range: string[xsd:length "2"^^xsd:nonNegativeInteger]
```

Це обмежує властивість "hasName" рядковими значеннями довжиною принаймні 2 символи, використовуючи вбудований рядковий тип даних і аспект "length".

Загалом, ці додаткові конструкції в OWL 2 дозволяють більш точне та гнучке поводження з властивостями типу даних, забезпечуючи більш складне та точне представлення значень даних в онтологіях.

Ланцюжки властивостей

В OWL 2 ланцюжки властивостей дозволяють композицію властивостей створювати складніші зв’язки між класами. Ось кілька прикладів того, як працюють ланцюжки власності:

Транзитивні ланцюжки властивостей: OWL 2 дозволяє створювати транзитивні ланцюжки властивостей, де ланцюжок властивостей довжиною n еквівалентний транзитивній властивості глибини `n+1`. Наприклад, наступна конструкція OWL 2 створює транзитивний ланцюжок властивостей для властивості "isAncestorOf":

ObjectProperty: isAncestorOf

```
ObjectPropertyChain: hasParent o isAncestorOf
  TransitiveObjectProperty: isAncestorOf
```

Це визначає властивість "isAncestorOf" як транзитивну на основі ланцюжка властивостей "hasParent o isAncestorOf", що означає, що якщо A hasParent B і B є AncestorOf C, то A isAncestorOf C.

Інверсні ланцюжки властивостей: OWL 2 дозволяє створювати інверсні ланцюжки властивостей, де ланцюжок властивостей довжиною `n` еквівалентний зворотній ланцюжку властивостей довжиною `n` у протилежному напрямку. Наприклад, наступна конструкція OWL 2 створює інверсний ланцюжок властивостей для властивості “hasGrandchild”:

ObjectProperty: hasGrandchild

```
ObjectPropertyChain: hasChild o hasChild o hasChild
  InverseObjectProperty: hasGrandchild
```

Це визначає властивість “hasGrandchild” як зворотний ланцюжок властивостей “hasChild o hasChild o hasChild”, що означає, що якщо A hasGrandchild C, то існує ланцюжок із трьох зв’язків “hasChild” від A до C.

Кваліфіковані обмеження властивостей: OWL 2 дозволяє створювати кваліфіковані обмеження властивостей, де ланцюжок властивостей обмежується певною кількістю випадків певної властивості. Наприклад, така конструкція OWL 2 створює кваліфіковане обмеження властивості для властивості "hasChild":

ObjectProperty: hasChild

```
Class: Family
  SubClassOf: hasChild exactly 2 Person
Class: Person
```

Це визначає обмеження на властивість "hasChild", де будь-який екземпляр "Сім'ї" повинен мати рівно двох осіб "Person", з'єднаних через властивість "hasChild".

Загалом, ці додаткові конструкції в OWL 2 дозволяють більш складне та виразне представлення зв’язків між класами, забезпечуючи більш точне та точне моделювання доменів реального світу.

Кваліфіковані обмеження потужності. У OWL 2 кваліфіковані обмеження потужності дозволяють створювати більш складні обмеження на властивості, де обмеження застосовуються до певної підмножини значень властивостей на основі певної умови. Ось кілька прикладів того, як працюють кваліфіковані обмеження кардинальності:

Кваліфіковані обмеження потужності з обмеженнями значень: OWL 2 дозволяє створювати кваліфіковані обмеження кількості, які застосовуються до певної підмножини значень властивостей на основі обмеження значення. Наприклад, наступна конструкція OWL 2 створює кваліфіковане обмеження потужності для властивості "hasChild":

ObjectProperty: hasChild

Class: Family

```
  SubClassOf: hasChild exactly 2 Person and hasChild only (hasAge
some xsd:integer[>=18])
Class: Person
  DatatypeProperty: hasAge
```

Це визначає обмеження на властивість “hasChild”, де будь-який екземпляр “Сім’ї” повинен мати рівно дві особи “Person”, пов’язані через властивість “hasChild”, а значення їх властивості “hasAge” має бути більше або дорівнювати 18.

Кваліфіковані обмеження кардинальності з обмеженнями класу: OWL 2 також дозволяє створювати кваліфіковані обмеження кардинальності, які застосовуються до певної підмножини значень властивостей на основі обмеження класу. Наприклад, наступна конструкція OWL 2 створює кваліфіковане обмеження потужності для властивості “hasChild”:

ObjectProperty: hasChild

```
Class: Family
   SubClassOf: hasChild exactly 2 Adult and hasChild only (hasAge some xsd:integer[>=18])
Class: Person
  DatatypeProperty: hasAge
Class: Adult
  SubClassOf: Person and (hasAge some xsd:integer[>=18])
```

Це визначає обмеження на властивість “hasChild", де будь-який екземпляр "Сім'ї" повинен мати рівно двох “Дорослих" осіб, з'єднаних через властивість “hasChild", а значення їх властивості “hasAge" має бути більше або дорівнювати 18.

Загалом, ці додаткові конструкції в OWL 2 дозволяють більш складне та виразне представлення обмежень на властивості, уможливлюючи більш точне та точне моделювання доменів реального світу.

Функціональний синтаксис OWL 2

Функціональний синтаксис OWL 2 − це компактний і зрозумілий синтаксис для представлення онтологій OWL 2. Ось кілька прикладів аксіом OWL 2, представлених у функціональному синтаксисі:

1. Аксіома визначення класу: `SubClassOf(Man Person)`

Ця аксіома визначає, що клас “Людина” є підкласом класу “Людина”.

2. Аксіома визначення властивості об’єкта: `ObjectPropertyAssertion(hasChild John Mary)`

Ця аксіома вказує, що між особами “Джон” і “Мері” існує твердження про властивість об’єкта через властивість об’єкта "hasChild".

3. Аксіома визначення властивості типу даних: DatatypePropertyAssertion(hasAge John "25"^^xsd:integer)

Ця аксіома визначає, що екземпляр “Джон” має твердження типу даних для властивості “hasAge” зі значенням "25" типу даних xsd:integer.

4. Аксіома еквівалентного класу: `EquivalentClasses(Person Male or Female)`

Ця аксіома вказує, що клас “Людина” еквівалентний об’єднанню класів “Чоловік” і “Жінка”.

5. Аксіома класів, що не перетинаються: `DisjointClasses(Dog Cat)`

Ця аксіома вказує на те, що класи “Собака” і “Кіт” є непересічними, тобто вони не мають спільних індивідів.

Загалом, функціональний синтаксис OWL 2 забезпечує стислий і читабельний спосіб представлення онтологій OWL 2, полегшуючи людям розуміння та роботу з цими складними моделями.

Нові типи аксіом

В OWL 2 є кілька додаткових типів аксіом, окрім основних, які дозволяють більш складне та виразне моделювання доменів реального світу. Ось кілька прикладів додаткових типів аксіом:

1. **Аксіома непересічного об’єднання:** ця аксіома визначає, що клас є непересічним об’єднанням двох або більше інших класів.

Наприклад: `DisjointUnion(Pet Dog Cat)`

Ця аксіома вказує, що клас “Домашня тварина” є непересічним об’єднанням класів “Собака” і “Кішка”.

2. **Ключова аксіома:** Ця аксіома визначає, що набір властивостей однозначно ідентифікує екземпляри класу.

Наприклад: `HasKey(Man hasSSN)`

Ця аксіома визначає, що екземпляри класу "Man" однозначно ідентифікуються значенням їх властивості "hasSSN".

3. **Аксіома твердження:** ця аксіома визначає одне твердження щодо особи.

Наприклад: `Assertion(Alice likes Bob)`

Ця аксіома вказує на те, що особа “Аліса” любить особистість “Боб”.

4. **Аксіома правила SWRL:** ця аксіома визначає правило мови правил семантичної мережі (SWRL), яке можна використовувати для виведення нових знань із наявних.

Наприклад:

`Rule1: Person(?x) ∧ hasChild(?x, ?y) ∧ hasChild(?y, ?z) → Grandparent(?x, ?z)`

Ця аксіома вказує на правило SWRL, яке визначає зв’язок “Grandparent” між особами на основі зв’язку “hasChild” між іншими особами. Загалом, ці додаткові типи аксіом в OWL 2 забезпечують більш складні та потужні можливості міркування, дозволяючи точніше та точніше моделювати домени реального світу.

Профілі OWL 2

OWL 2 (***Web Ontology Language***) − це сімейство мов представлення знань для створення онтологій у семантичній мережі. Існує кілька профілів OWL 2, кожен з яких призначений для певних цілей і має різні рівні виразності та обчислювальної складності.

Профілями OWL 2 є:

1. **OWL 2 Full:** ей профіль дозволяє використовувати всі мовні конструкції OWL 2, але не гарантовано, що він буде доступним. Він використовується в ситуаціях, коли потрібна повна виразність, але обчислювальна здатність не викликає занепокоєння.
2. **OWL 2 DL** (***Description Logic***)**:** це найбільш часто використовуваний профіль OWL 2. Він заснований на логіці опису SROIQ і розроблений для балансу виразності та обчислювальної зручності. Він дозволяє використовувати підмножину мовних конструкцій OWL 2 і має ефективні алгоритми міркування.
3. **OWL 2 QL** (***Qualified Restriction***)**:** цей профіль розроблено для розробки онтології в ситуаціях, коли обчислювальна здатність є основною проблемою. Він заснований на підмножині конструкцій OWL 2 DL і має ефективні алгоритми міркування.
4. **OWL 2 EL (*Expressive Logic*):** цей профіль призначений для розробки онтології в ситуаціях, коли масштабованість є основною проблемою. Він заснований на підмножині конструкцій OWL 2 DL і має дуже ефективні алгоритми міркування.
5. **OWL 2 RL (*Rule Language*):** цей профіль призначений для розробки онтології в ситуаціях, коли правила є головною проблемою. Він дозволяє використовувати підмножину мовних конструкцій OWL 2 і базується на мові правил під назвою Rule Interchange Format (RIF).

OWL 2 DL (Description Logic)

OWL 2 DL (Description Logic) − це профіль мови веб-онтологій OWL 2, розроблений для балансування експресивності та обчислювальної здатності онтологій. Це найбільш часто використовуваний профіль OWL 2 і заснований на логіці опису SROIQ. Ось деякі відмінні риси OWL 2 DL:

- *Формальна семантика:* OWL 2 DL має чітко визначену формальну семантику, засновану на логіці опису, яка є сімейством логік, які добре підходять для представлення та міркування щодо структурованих знань.
- *Виразність:* OWL 2 DL достатньо виразний, щоб представляти складні онтології, включно зі складними зв’язками між класами та властивостями.
- *Можливість вирішення:* OWL 2 DL розроблено таким чином, щоб бути вирішальним, що означає, що існує алгоритм, який може вирішити, чи є певна онтологія узгодженою чи ні.
- *Ефективне міркування:* OWL 2 DL має ефективні алгоритми міркування, які дозволяють автоматизовано міркувати над великомасштабними онтологіями.
- *Строгий синтаксис і семантика:* OWL 2 DL має строгий синтаксис і семантику, що означає, що існує чітка і однозначна інтерпретація кожної конструкції в мові.
- *Підтримка аксіом і висновків:* OWL 2 DL підтримує використання аксіом, які дозволяють явно відображати зв’язки між класами та властивостями, і висновки, які дозволяють автоматично виводити нові знання на основі аксіом в онтології.

Загалом, OWL 2 DL забезпечує баланс між експресивною потужністю та обчислювальною здатністю, що робить його корисним профілем для широкого спектру завдань розробки онтологій.

OWL 2 QL (Qualified Restriction)

OWL 2 QL (***Qualified Restriction***) − це профіль мови веб-онтологій OWL 2, розроблений для збалансування експресивності та обчислювальної здатності онтологій.

Він має обмеження мовних конструкцій, базується на логіці SROIQ, має ефективні алгоритми міркування, підтримує конструктори класів і властивостей, підтримує конструктори діапазонів даних, забезпечує гарантовану вирішуваність.

Обмеження мовних конструкцій: OWL 2 QL обмежує використання певних мовних конструкцій, щоб підтримувати обчислювальну здатність. Зокрема, він обмежує використання номіналів, обмеження кількості та деякі типи заперечень. OWL 2 QL базується на логіці опису **SROIQ**, яка є тією самою логікою, що лежить в основі OWL 2 DL. OWL 2 QL має **ефективні алгоритми міркування**, які дозволяють міркувати над великомасштабними онтологіями. OWL 2 QL дозволяє використовувати такі **конструктори класів**, як перетин, об’єднання, доповнення та обмеження існування (`someValuesFrom`) у властивостях. Це також дозволяє використовувати конструктори властивостей, такі як зворотне, ланцюгове та транзитивне замикання. OWL 2 QL дозволяє використовувати **конструктори діапазонів даних**, такі як універсальне обмеження (`allValuesFrom`) на типи даних та їх відповідні логічні комбінації (наприклад, перетину та доповнення). OWL 2 QL є розв’язним фрагментом OWL 2, що означає, що гарантовано існує алгоритм, який може вирішити, чи є дана онтологія узгодженою чи ні.

Загалом, OWL 2 QL забезпечує баланс між виразною потужністю та обчислювальною здатністю, що робить його корисним профілем для завдань розробки онтології, де ефективність міркувань є головною проблемою, але онтологія досить складна, щоб вимагати нетривіального використання конструкторів класів і властивостей.

OWL 2 EL (мова веб-онтології OWL 2)

OWL 2 EL (мова веб-онтології OWL 2, профіль для логіки опису з виразом обмежень) − це профіль OWL 2, оптимізований для масштабованих

міркувань із великими складними базами знань. Він заснований на логіці опису EL++, яка є доступним фрагментом більшої мови логіки опису.

OWL 2 EL досягає своєї масштабованості, обмежуючи виразність OWL 2, зберігаючи достатню кількість функцій для підтримки більшості практичних програм. Профіль особливо підходить для додатків, які вимагають ефективного обґрунтування, таких як інтеграція біомедичних даних, керування корпоративними даними та соціальні мережі.

Деякі з ключових особливостей OWL 2 EL включають:

1. Обмежені вирази класу: OWL 2 EL обмежує використання складних виразів класу, таких як перетин, об’єднання та доповнення, щоб підтримувати зручність.
2. Обмежені вирази властивостей: OWL 2 EL також обмежує використання складних виразів властивостей, таких як транзитивність і інверсія, щоб переконатися, що аргументація залишається ефективною.
3. Ієрархія ролей: OWL 2 EL підтримує обмежену форму ієрархії ролей, що дозволяє легко організувати та підтримувати великі бази знань.
4. Властивості типу даних: OWL 2 EL підтримує властивості типу даних, які дозволяють вказувати обмеження на значення даних.
5. Масштабовані міркування: обмежена виразність і оптимізовані алгоритми міркування OWL 2 EL роблять його добре придатним для масштабованих міркувань із великими складними базами знань.

Загалом OWL 2 EL забезпечує баланс між виразністю та масштабованістю, що робить його корисним профілем для широкого спектру практичних застосувань у різних областях.

Вирази класу, дозволені в OWL 2 EL

В OWL 2 EL вирази класу обмежені підмножиною повної експресивності OWL 2. Це робиться для того, щоб підтримувати зручність і ефективне міркування з великими базами знань.

Вирази класу, дозволені в OWL 2 EL, обмежені такими конструкціями:

1. Перетин атомарних класів: перетин двох або більше атомарних класів дозволено в OWL 2 EL. Наприклад, “Person ⊓ Doctor” є дійсним виразом класу в OWL 2 EL.
2. Екзистенціальні обмеження з атомарними іменами ролей: OWL 2 EL дозволяє використовувати екзистенціальні обмеження з атомарними іменами ролей. Наприклад, “Person ⊓ ∃hasChild.Child” є дійсним виразом класу в OWL 2 EL.
3. Обмеження на значення з властивостями типу даних: OWL 2 EL підтримує використання обмежень на значення з властивостями типу даних. Наприклад, “Person ⊓ hasAge some xsd:integer[>=18]” є дійсним виразом класу в OWL 2 EL.
4. Універсальні обмеження: OWL 2 EL підтримує використання універсальних обмежень, які дозволяють визначати класи, визначені з точки зору відсутності певних властивостей. Наприклад, “Person ⊓ ¬hasDisease.top” є дійсним виразом класу в OWL 2 EL.
5. Заперечення атомарних класів: OWL 2 EL підтримує заперечення атомарних класів. Наприклад, “Person ⊓ ¬Doctor” є дійсним виразом класу в OWL 2 EL.

Загалом, ці обмеження на вирази класу в OWL 2 EL допомагають гарантувати, що міркування залишаються доступними навіть для великих і складних баз знань.

Вирази властивостей, дозволені в OWL 2 EL

В OWL 2 EL вирази властивостей також обмежені підмножиною повної виразності OWL 2, щоб забезпечити зручність і ефективне міркування з великими базами знань.

Вирази властивостей, дозволені в OWL 2 EL, обмежені такими конструкціями:

1. Прості ланцюжки властивостей: OWL 2 EL дозволяє використовувати прості ланцюжки властивостей, які складаються з однієї або кількох атомарних властивостей, з'єднаних операторами композиції ролей. Наприклад, "hasParent o hasParent" є дійсним простим ланцюжком властивостей у OWL 2 EL.
2. Інверсні властивості: OWL 2 EL підтримує використання інверсних властивостей, які дозволяють уточнювати стосунки між двома особами в протилежному напрямку. Наприклад, "hasParent^-1" є дійсним виразом зворотної властивості в OWL 2 EL.
3. Перехідні властивості: OWL 2 EL підтримує використання перехідних властивостей, які дозволяють виводити нові відносини між особами на основі існуючих відносин. Наприклад, "hasAncestor o hasAncestor" є дійсним виразом транзитивної властивості в OWL 2 EL.
4. Рефлексивні властивості: OWL 2 EL підтримує використання рефлексивних властивостей, які дозволяють уточнювати стосунки між особою та собою. Наприклад, "hasSelf" є дійсним виразом рефлексивної властивості в OWL 2 EL.
5. Симетричні властивості: OWL 2 EL підтримує використання симетричних властивостей, які визначають, що якщо індивід A пов'язаний з індивідом B властивістю, то B також пов'язаний з A цією властивістю. Наприклад, "hasSibling^-1" є дійсним виразом симетричної властивості в OWL 2 EL.

Загалом ці обмеження на вирази властивостей у OWL 2 EL допомагають гарантувати, що міркування залишаються доступними навіть для великих і складних баз знань. Обмежуючи виразність виразів властивостей, OWL 2 EL забезпечує баланс між виразністю та масштабованістю, що робить його корисним профілем для широкого спектру практичних застосувань у різних областях.

OWL 2 RL (Rule Language)

OWL 2 RL (Rule-based Reasoning Language) − це профіль мови веб-онтології (OWL) 2, семантичної веб-мови, призначеної для представлення та міркування про знання структурованим і зрозумілим для машини способом. OWL 2 RL спеціально розроблено для забезпечення ефективного міркування з підмножиною OWL 2, що робить його придатним для додатків, які вимагають масштабованих та легких виводів.

Основною метою OWL 2 RL є досягнення балансу між експресивною потужністю та обчислювальною складністю. Це забезпечує компроміс між більш виразними функціями OWL 2 і бажанням ефективного виводу в середовищах з обмеженими ресурсами. OWL 2 RL має на меті забезпечити рівень виразності, корисний для багатьох практичних застосувань, одночасно гарантуючи, що завдання виводу можуть виконуватися з відносно низькими обчислювальними витратами.

Основні характеристики профілю OWL 2 RL:

1. Вивід на основі правил: OWL 2 RL значною мірою покладається на використанні правил. Він визначає набір правил виводу, які фіксують семантику мовних конструкцій. Ці правила дозволяють ефективно виводити нові знання з існуючих в онтології.
2. Масштабованість: OWL 2 RL розроблений, щоб бути обчислювально ефективним і масштабованим. Заснований на правилах характер

аргументації дозволяє оптимізувати продуктивність порівняно з більш виразними профілями OWL.

3. Спрощені вирази: OWL 2 RL є підмножиною повної мови OWL 2. Він включає в себе вибір конструкцій, які досягають балансу між виразною силою та обчислювальною складністю. Деякі експресивні риси опущено, щоб забезпечити ефективне міркування.
4. Область застосування: OWL 2 RL особливо підходить для програм, які вимагають легких виводів, таких як інтеграція даних, запити, опосередковані онтологіями, і певні типи завдань перевірки даних. Він не призначений для програм, які вимагають повної експресивної потужності OWL 2.
5. Мовні обмеження: OWL 2 RL накладає певні обмеження на використання конструкцій, таких як ланцюжки властивостей, обмеження типів даних і складні вирази класу. Ці обмеження гарантують, що міркування залишаються ефективними та доступними.

Важливо відзначити, що OWL 2 RL жертвує деякими виразними можливостями заради досягнення ефективності, тому він може не підходити для всіх сценаріїв моделювання онтології. Якщо додаток потребує більш складних міркувань або виразності, ніж те, що надає OWL 2 RL, тоді може бути доцільним використання іншого профілю OWL 2, такого як OWL 2 EL або OWL 2 DL.

Подолання обмеження OWL для задання обмежень на дані

Обмеження OWL

* неможливо описати загальні правила типу If-ELSE

* неможливо задати обмеження на “дані” (ABOX) (тільки на класи, тобто рівень TBOX)

* неможливо використовувати нечітку, ймовірнісну, Басову логіку

* обмежено підтримується темпоральне/просторове виведення

* онтологія OWL − це не база даних, якщо даних явно немає у ABOX, база даних дасть відповідь “немає даних”, а машина виведення для **OWL** − “не знаю”) (Open World припущення замість Clone World)
* не підтримує унікальність імен
Подолання обмеження OWL для задавання обмежень на дані

Shape Constraint Language(SHACL) − новий стандарт (2017) для представлення обмежень на RDF граф.

Shape − це шаблон підграфа − “зірка” з одним центром, яка описує, як дані “мають бути” представлені.

Shacl − доповнення до OWL, але у певних ситуаціях може замінити його.

Інші мови загального користування для специфікації онтологій

Крім уже описаних вище **RDF (*Resource Description Framework*), RDF-Turtle, RDF-Notation 3, OWL (*Web Ontology Language*), RDFS (*RDF Schema*)**, існують інші широко використовувані мови опису онтології:

SHACL (Shapes Constraint Language / мова обмеження форм):

SHACL − це мова для опису та перевірки графових структур RDF на попередньо визначені форми чи шаблони.

SKOS (*Simple Knowledge Organization System / проста система організації знань*):
SKOS призначений для представлення систем організації знань, таких як таксономії та тезауруси.

Формат OBO (*Open Biomedical Ontologies / відкриті біомедичні онтології*):

Формат OBO широко використовується в біомедичній сфері для створення онтологій, пов’язаних з генами, білками та захворюваннями.

Common Logic (CL): Common Logic − це сімейство мов представлення знань, які використовуються для визначення та спільного використання онтологій.

Description Logics (DL): DLs − це сімейство формальних мов, які часто використовуються як основа для OWL та інших мов онтології.

Синтаксис Manchester OWL: Більш зручний для людини синтаксис для онтологій OWL, який часто використовується для редагування та візуалізації.

TPTP (Thousands of Problems for Theorem Provers / Тисячі задач для доказів теорем): TPTP − це мова для опису проблем автоматизованого доведення теорем, яка часто використовується в розробці онтології.

RIF (Rule Interchange Format / формат обміну правилами): RIF − це сімейство мов, розроблених для вираження правил, які часто використовуються разом з онтологіями.

LinkML

LinkML − це мова моделювання з відкритим вихідним кодом і структура, спеціально розроблена для створення моделей даних і онтологій і керування ними. Це особливо корисно в галузі наук про життя та біоінформатики, але його можна застосовувати і в інших областях. LinkML має на меті спростити процес створення, підтримки та спільного використання структурованих моделей даних і онтологій.

SHACL

SHACL (мова обмеження форм), що розшифровується як **Shapes Constraint Language**, є потужним і гнучким стандартом для визначення та перевірки обмежень на графах даних RDF. **RDF (Resource Description Framework)** − це широко використовувана структура для представлення та обміну даними в Інтернеті. SHACL дозволяє визначати фігури або шаблони, які

описують структуру та обмеження, яким повинні відповідати екземпляри даних у графі RDF.

SHACL в основному використовується для перевірки структури та вмісту даних RDF. Це дозволяє визначати обмеження, яким мають задовольняти дані, щоб вважатися дійсними. У SHACL “форма” є формальним описом очікуваної структури даних RDF. Фігури визначають класи, властивості та умови, яким мають відповідати екземпляри даних.

SHACL працює з графами даних RDF, які складаються з трійок (висловлювань суб’єкт-предикат-об’єкт), які представляють знання. Форми SHACL використовуються для визначення очікуваної структури цих графіків. SHACL підтримує широкий спектр обмежень і правил, які можна застосовувати до даних. Обмеження включають такі вимоги, як типи даних, діапазони значень, потужність (наприклад, “рівно один”) тощо. Правила дозволяють більш складні умови та перетворення на основі значень даних.

SHACL надає багатий і виразний синтаксис для визначення форм і обмежень. Він використовує сам RDF для визначення фігур, що означає, що форми SHACL можна зберігати та обмінювати як даними RDF. Коли ви застосовуєте форми SHACL до діаграми даних RDF, ви можете перевірити, чи дані відповідають заданим обмеженням. Результати перевірки можуть виявити проблеми, наприклад дані, які порушують обмеження або не відповідають очікуваним формам.

SHACL розроблений для роботи з різними джерелами даних RDF і може використовуватися для перевірки даних з різних систем і програм на основі RDF. Це покращує взаємодію, гарантуючи, що дані відповідають загальним стандартам і очікуванням. SHACL використовується в різних областях, включаючи інтеграцію даних, забезпечення якості даних, розробку графів знань і перевірку даних у семантичних веб-додатках.

SHACL − це галузева стандартна мова та специфікація, розроблена **World Wide Web Consortium / Консорціумом Всесвітньої павутини (W3C)**, що забезпечує її широке впровадження та підтримку.

SKOS

SKOS (Проста система організації знань), що розшифровується як **Simple Knowledge Organization System**, є стандартизованою моделлю та словником для представлення та організації знань у структурованому та машиночитаному вигляді. Він призначений головним чином для управління та обміну системами організації знань (KOS), такими як тезауруси, таксономії, схеми класифікації та предметні рубрики. SKOS забезпечує загальну структуру для кодування та обміну концепціями та зв’язками, які використовуються для категоризації та опису інформаційних ресурсів. Ось деякі ключові функції та поняття, пов’язані зі SKOS:

SKOS використовується для представлення різних форм KOS, включаючи тезауруси, контрольовані словники, предметні рубрики та системи класифікації. Ці системи допомагають упорядковувати та класифікувати інформаційні ресурси для кращого пошуку та навігації. SKOS фокусується на представленні окремих понять або термінів у системі організації знань. Кожне поняття має унікальний ідентифікатор і може мати мітки, синоніми та визначення.

SKOS дозволяє моделювати зв’язки між поняттями, наприклад ширші/вужчі (ієрархічні), пов’язані поняття та асоціативні зв’язки. Ієрархічні зв’язки представляють ширші та вужчі поняття, тоді як асоціативні зв’язки поєднують поняття, які так чи інакше пов’язані, але не ієрархічно.

SKOS підтримує кілька міток для концепцій, включаючи бажані мітки та альтернативні мітки (синоніми). Поняття також можуть бути пов’язані з документацією, включаючи примітки щодо обсягу, визначення та редакційні примітки. SKOS зазвичай представлено за допомогою Resource Description

Framework (RDF), стандарту для представлення даних і метаданих в Інтернеті. SKOS використовує можливості RDF, щоб зв’язувати концепції та ресурси в Інтернеті. SKOS розроблено таким чином, щоб бути дружнім до Інтернету та сприяти доступності систем організації знань у мережі. Концепції в SKOS можна пов’язувати з такими ресурсами, як документи, веб-сайти та інші онлайн-матеріали. SKOS сприяє взаємодії, надаючи стандартизований спосіб представлення та обміну даними KOS. Це дає змогу різним системам і програмам використовувати та розуміти однакові концепції та зв’язки. SKOS розроблено як рекомендація W3C, що робить його міжнародно визнаним і широко прийнятим стандартом. Специфікація відома як “Довідник системи простої організації знань SKOS”.

SKOS використовується в різних програмах, включаючи бібліотечну та інформаційну науку, управління контентом, інтеграцію даних, пошукові системи та семантичну мережу. Це покращує організацію та пошук інформаційних ресурсів у цифрових бібліотеках, каталогах та системах пошуку інформації.

Формат OBO

Open Biomedical Ontologies (OBO / відкриті біомедичні онтології) − це формат файлів і пов’язана модель даних, які використовуються для представлення та обміну біомедичними онтологіями та контрольованими словниками. Формат OBO спеціально розроблений для наук про життя та біомедичних областей для охоплення структурованих знань про біологічні об’єкти, такі як гени, білки, хвороби та їхні взаємозв’язки.

Формат OBO використовується для представлення онтологій, які є структурованими моделями, що визначають поняття (класи) і зв’язки в певній області, наприклад геноміці, анатомії чи хворобі. Він також використовується для створення контрольованих словників, які є наборами термінів, що

використовуються для стандартизації та категоризації інформації в певній області.

Формат OBO розроблено таким чином, щоб він був зрозумілим і відносно легким для розуміння порівняно зі складнішими мовами онтології, такими як OWL (мова веб-онтології). Формат використовує звичайний текст і простий синтаксис для визначення термінів та їхніх зв’язків. Формат OBO містить положення для визначення визначень термінів, синонімів та інших метаданих, щоб забезпечити контекст і ясність для кожного терміна чи поняття. Онтології, представлені у форматі OBO, часто мають ієрархічну структуру з батьківсько-начірніми зв’язками між термінами, що дозволяє класифікувати та організовувати поняття.

Формат OBO підтримує представлення різних типів зв’язків між термінами, таких як зв’язки “`is_a`” (субсумпція), зв’язки `частина-ціле` тощо. Ці зв’язки допомагають визначити семантику та взаємозв’язки між термінами.

Розробка та підтримка онтологій OBO та самого формату OBO зазвичай є результатом спільноти. Кілька організацій і дослідницьких груп співпрацюють для створення та розширення онтологій у форматі OBO.

Формат OBO сприяє взаємодії, надаючи загальний спосіб представлення та спільного використання онтологій і контрольованих словників у науках про життя. Інструменти та програмне забезпечення, які підтримують формат OBO, можуть легше обмінюватися даними онтології та використовувати їх. Деякі добре відомі онтології, представлені у форматі OBO, включають онтологію генів (GO), фундаментальну модель анатомії (FMA), онтологію захворювань людини (DO) і онтологію білків (PRO) тощо. Для роботи з онтологіями у форматі OBO доступні різноманітні програмні інструменти та бібліотеки, включаючи парсери, валідатори, редактори та інструменти перегляду онтологій.

Таким чином, формат OBO є широко використовуваним форматом для представлення та спільного використання онтологій і контрольованих словників у біомедичних областях і науках про життя. Він забезпечує зрозумілий для людини підхід до моделювання та організації знань про біологічні об’єкти та їхні стосунки, керований спільнотою, що робить його цінним ресурсом для дослідників і професіоналів у цих галузях.

Common Logic (CL)

Common Logic/Загальна логіка (CL) − це сімейство мов представлення знань, призначених для забезпечення спільної основи для вираження та обміну знаннями в різних областях і комп’ютерних системах. Він спрямований на подолання розриву між різними формалізмами на основі логіки та мовами представлення знань, забезпечуючи єдину основу, яка може вмістити різні логіки та онтології. Common Logic розроблено робочою групою Common Logic Standard Working Group.

Загальна логіка служить загальною абстрактною структурою для представлення знань.

Він забезпечує спосіб визначення онтологій, баз знань і формальних моделей, які можна використовувати в різних програмах і контекстах.

Common Logic підтримує модульність, дозволяючи організовувати знання в окремі модулі або бібліотеки, які можна повторно використовувати в різних проектах.

Цей модульний підхід сприяє розробці стандартизованих онтологій і бібліотек.

Загальна логіка розроблена для розміщення кількох логічних систем, включаючи логіку першого порядку (FOL), логіку вищого порядку (HOL) і модальну логіку, серед інших.

Ця гнучкість дозволяє користувачам вибирати логіку, яка найкраще підходить для їх домену чи програми.

Common Logic має кілька мовних варіантів, зокрема **Common Logic Interchange Format (CLIF) і Common Logic Knowledge Interchange Format (CL-KIF)**. Ці варіанти забезпечують конкретний синтаксис і семантику для представлень на основі Common Logic.

Загальна логіка включає набір логічних операторів і конструкцій для визначення понять, відносин, аксіом і правил. Ці оператори використовуються для визначення структури та змісту представлень знань. Однією з основних цілей Common Logic є сприяння спільному використанню та обміну знаннями між різними системами та доменами. Він забезпечує взаємодію, визначаючи спільну мову для вираження знань.

Common Logic було розроблено як формальний стандарт, що робить його надійною та широко прийнятою основою для представлення знань. Робоча група Common Logic Standard Working Group відповідає за розробку та підтримку стандарту. Common Logic має застосування в різних сферах, включаючи штучний інтелект, формальні методи, розробку онтології, управління знаннями та семантичну мережу.

Він використовується для представлення онтологій, формальних специфікацій, знань предметної області тощо.

Існують програмні інструменти та бібліотеки, які підтримують Common Logic, надаючи синтаксичні аналізатори, резонери та редактори для створення та роботи з знаннями на основі Common Logic.

Підводячи підсумок, Common Logic (CL) − це сімейство мов представлення знань, які мають на меті забезпечити загальну структуру для вираження та обміну знаннями в різних областях і логічних системах. Він пропонує гнучкість, модульність і стандартизацію, що робить його цінним

інструментом для інженерів і дослідників, які працюють у різних наукомістких областях.

Description Logics (DLs)

Description Logics/Логіки опису (DLs) − це сімейство формальних мов представлення знань, які використовуються в **штучному інтелекті (ШІ), інженерії знань і Semantic Web**. DLs призначені для структурованого та логічно точного представлення та міркування про концепції, індивідів і відносини в межах конкретної області. Вони є основою для багатьох мов онтології, включаючи OWL (Web Ontology Language).

Логіки опису вкорінені у формальній логіці, зокрема в логіці першого порядку (FOL) і логіці висловлювань. Вони забезпечують формальну та сувору основу для представлення та міркування про знання.

DLs дозволяють моделювати концепції (класи), індивідуумів (екземпляри чи об’єкти) і ролі (властивості чи зв’язки) у межах домену. Концепції представляють набори індивідів, а ролі представляють відносини між індивідами.

DLs підтримують створення таксономій та ієрархій понять, що дозволяє моделювати зв’язки підпорядкування. Відносини субсумпції вказують на те, що одне поняття є більш конкретною (підклас) або більш загальною (суперклас) формою іншого поняття. DLs мають чітко визначену формальну семантику, що визначає, як інтерпретувати та міркувати про значення понять, осіб і ролей в онтології. Онтології DLs можуть включати аксіоми, які надають додаткові обмеження та правила для міркувань. Механізми логічного висновку можуть використовувати ці аксіоми для отримання неявних знань і відповідей на запити.

DLs мають різні смаки з різними рівнями експресивності та обчислювальної складності. Деякі DLs розроблені для виконання простих

завдань на міркування, тоді як інші є більш виразними та підтримують складні міркування.

Web Ontology Language (OWL) − це широко використовувана мова онтології, заснована на логіці опису. OWL є стандартом для представлення та спільного використання онтологій і є основою для семантичної мережі.

Опис Логіки знаходять застосування в різних сферах, включаючи інженерію знань, інтеграцію даних, онтологічне моделювання, медичну інформатику та обробку природної мови. Розсудники DL, також відомі як класифікатори DLs або аргументи DLs, це програмні засоби, які виконують завдання міркування, такі як перевірка узгодженості, класифікація та пошук примірників, на основі онтологій DL. Дослідження в логіці опису зосереджені на розробці ефективних алгоритмів міркування та інструментів для обробки великих і складних онтологій.

Manchester OWL Syntax

Manchester OWL Syntax/Манчестерський синтаксис OWL − це зручний і компактний синтаксис для написання онтологій і графів знань за допомогою Web Ontology Language (OWL). **OWL** − семантичний веб-стандарт для представлення та обміну онтологіями та знаннями в машиночитаному форматі. Манчестерський синтаксис OWL має на меті полегшити як розробникам онтологій, так і нефахівцям читання, написання та розуміння онтологій OWL. Синтаксис Manchester OWL розроблений таким чином, щоб бути більш зручним для людини та читабельним, ніж традиційний синтаксис RDF/XML або OWL/XML. Він використовує синтаксис, який ближче до природної мови та легший для розуміння нефахівцям. Синтаксис Manchester OWL відомий своєю лаконічністю та стислістю. Це дозволяє розробникам онтологій виражати складні конструкції онтології з мінімальною багатослівністю. Ця компактність може призвести до більш інтуїтивно зрозумілих і керованих представлень онтології.

У синтаксисі Manchester OWL вирази класів записуються за допомогою простих фраз, схожих на англійську.

Наприклад, ви можете визначити клас як "`Person and hasAge some integer[>=18]`", щоб представити осіб, які є особами та мають вік більше або дорівнює 18.

Вирази властивостей так само спрощені в синтаксисі Manchester OWL. Ви можете визначити властивості та зв’язки більш інтуїтивно зрозумілим способом, наприклад “`hasParent some Person`”, щоб вказати, що особа має батька, який є особою. Синтаксис Manchester OWL дозволяє вказати обмеження властивостей простим способом. Наприклад, ви можете визначити обмеження властивості як “**`hasChild точно 2 особи`**”, щоб вказати, що особа має рівно двох дітей, які є особами. Синтаксис Manchester OWL підтримує включення аксіом і анотацій для додавання додаткової інформації або обмежень до елементів онтології. Анотації можна використовувати для документації та метаданих. Manchester OWL Syntax полегшує визначення та навігацію зв’язків та ієрархій в онтологіях. Це спрощує представлення відношень субсумпції та ієрархій класів.

Багато інструментів розробки онтологій і редакторів підтримують синтаксис Manchester OWL, що робить його доступним для розробників і дослідників онтології. Манчестерський синтаксис OWL часто використовується в освітніх установах для навчання моделюванню онтології та OWL для початківців завдяки його доступному та зручному для читання формату. Таким чином, синтаксис Манчестерської OWL є зручним і компактним способом представлення онтологій OWL. Це спрощує процес читання та запису онтологій, роблячи його більш доступним для ширшої аудиторії, включаючи неекспертів і тих, хто новачок у семантичних веб-технологіях. Це особливо корисно для покращення читання та розуміння онтології, особливо для освітніх та спільних проектів розробки онтології.

TPTP

Thousands of Problems for Theorem Provers (TPTP/ Тисячі задач для доказів теорем) − це бібліотека контрольних задач і інструментів, призначених для тестування й оцінки автоматизованих систем доказу теорем і пов’язаних формальних інструментів міркування. TPTP було створено, щоб надати стандартизований і вичерпний набір проблем, які дослідники та розробники можуть використовувати для оцінки продуктивності та правильності перевірок теорем і програмного забезпечення для автоматизованого міркування. Ось опис TPTP:

TPTP було розроблено для підтримки досліджень і розробок у сфері автоматизованого доведення теорем, формальних методів і суміжних областей. Він має на меті надати різноманітну колекцію контрольних задач, які охоплюють широкий спектр математичних і логічних областей. TPTP служить сховищем для тисяч тестових проблем, виражених різними формальними мовами та логіками. Проблеми в TPTP включають теореми, аксіоми, визначення та інші математичні твердження, які можна використовувати для тестування та експериментування.

TPTP надає стандартизовані формати та угоди для представлення проблем на різних логічних мовах, включаючи логіку першого порядку, логіку вищого порядку та логіку висловлювань. Ці стандартизовані формати спрощують порівняння та оцінку різних засобів доказування теорем і міркувань. TPTP класифікує проблеми за різними областями та класами, такими як теорія чисел, теорія множин, алгебра тощо. Проблеми також можна класифікувати за рівнями складності, що дозволяє дослідникам вибирати завдання, які відповідають їхнім цілям.

TPTP сумісний з різними автоматизованими засобами доказування теорем і формальними засобами міркування.

Проблеми порівняння можна використовувати як вхідні дані для цих інструментів для оцінки їх продуктивності та правильності. Дослідники та розробники можуть використовувати TPTP, щоб проводити порівняльні дослідження засобів доведення теорем, досліджувати сильні та слабкі сторони різних систем і вдосконалювати сучасні автоматизовані міркування. TPTP є розширюваним, що дозволяє користувачам вносити нові проблеми та розширювати бібліотеку додатковими наборами тестів. Цей підхід, керований спільнотою, гарантує, що TPTP залишається цінним ресурсом для дослідницької спільноти. TPTP доступний в Інтернеті, і користувачі можуть отримати доступ до проблем тестування, документації та відповідних інструментів через веб-сайт TPTP. TPTP включає дослідницькі виклики, де конкретні набори проблем організовуються в змаганнях або оцінювальних вправах, сприяючи співпраці та інноваціям в автоматизованих міркуваннях.

Таким чином, **TPTP (Thousands of Problems for Theorem Provers)** − це бібліотека контрольних задач і пов’язаних інструментів, які використовуються для оцінювання та тестування автоматизованих засобів доказування теорем і формальних систем міркування. Він відіграє вирішальну роль у просуванні досліджень і розробок автоматизованих міркувань, логіки та формальних методів, надаючи стандартизований і різноманітний набір проблем для експериментів і оцінювання.

RIF

RIF (Формат обміну правилами)− це сімейство мов, розроблених для вираження правил, які часто використовуються разом з онтологіями.

Rule Interchange Format (RIF) − це стандартизована специфікація для представлення та обміну правилами в машиночитаному форматі. RIF був розроблений як частина Консорціуму Всесвітньої павутини (W3C) для забезпечення взаємодії та обміну правилами між різними механізмами правил і системами, заснованими на знаннях. Ось огляд RIF:

RIF призначений для полегшення обміну та спільного використання правил між різними механізмами правил, експертними системами та системами представлення знань. Він має на меті забезпечити загальний формат для правил кодування, полегшуючи інтеграцію систем на основі правил і обмін знаннями на основі правил між різними платформами. RIF підтримує низку типів правил, у тому числі правила виробництва (пряме з’єднання), правила реакції та обмеження цілісності. Він розроблений для адаптації до різних парадигм правил, включаючи правила на основі логіки, правила у стилі ruleML тощо.

RIF дозволяє модульне визначення наборів правил, що дає змогу створювати багаторазові модулі правил і бібліотеки. Це сприяє належній інженерній практиці правил і спрощує обслуговування правил. RIF забезпечує чітко визначений синтаксис для представлення правил за допомогою форматів на основі XML і RDF. Синтаксис на основі XML називається RIF-XML, тоді як синтаксис на основі RDF називається RIF-Turtle. RIF є розширюваним і дозволяє визначати діалекти правил, які є конкретними мовами правил, адаптованими до різних доменів або програм. Кожен діалект визначає власний синтаксис, семантику та правила.

Основною метою RIF є забезпечення взаємодії між різними механізмами правил і системами аргументації. Системи, які підтримують RIF, можуть імпортувати та експортувати правила у форматі RIF, забезпечуючи спільний доступ до правил і їх виконання на різних платформах. RIF розроблено для інтеграції з іншими технологіями семантичного вебу, такими як RDF і OWL, щоб уможливити аргументацію на основі правил у контексті семантичного вебу. RIF визначає різні рівні відповідності, від основного до повного, щоб відповідати різним рівням можливостей і складності системи правил.

Різні механізми правил і програмні інструменти реалізували підтримку RIF, що робить його практичним для додатків, заснованих на правилах.

Розробка RIF була спільним зусиллям за участю експертів з наукових кіл, промисловості та органів стандартизації, таких як W3C.

LinkML

LinkML − це мова моделювання з відкритим вихідним кодом і структура, спеціально розроблена для створення моделей даних і онтологій і керування ними. Це особливо корисно в галузі наук про життя та біоінформатики, але його можна застосовувати і в інших областях. LinkML має на меті спростити процес створення, підтримки та спільного використання структурованих моделей даних і онтологій. Ось деякі функції LinkML:

Моделювання даних і розробка онтології: LinkML використовується для визначення та представлення моделей даних, включаючи онтології, схеми та доменно-спеціальні структури даних.

Він надає спосіб охопити концепції домену, зв’язки та обмеження в структурованому та стандартизованому форматі.

Визначення схеми: LinkML використовує зручний синтаксис YAML для визначення елементів схеми, що полегшує читання та запис. Схеми в LinkML визначають сутності, атрибути, зв’язки та обмеження.

Міждоменна сумісність: моделі LinkML можна конвертувати в різні формати обміну даними, включаючи RDF (Resource Description Framework), OWL (Web Ontology Language), JSON та інші. Це дозволяє LinkML подолати розрив між різними стандартами моделювання та онтології, що робить його універсальним інструментом для інтеграції даних.

Анотації метаданих: LinkML підтримує додавання анотацій метаданих до елементів схеми, покращуючи документацію та семантичну ясність моделей.

Ці анотації можуть містити описи, посилання та іншу інформацію про елементи схеми.

Інструменти та інтеграція: LinkML надає ряд інструментів і утиліт для роботи з моделями даних, включаючи генерацію коду для багатьох мов програмування та перевірку моделей. Його можна інтегрувати з різними середовищами розробки та системами керування даними.

Співпраця та контроль версій: LinkML заохочує спільну розробку онтологій і підтримує системи контролю версій, такі як Git. Це допомагає командам керувати змінами в онтологіях і відстежувати їх розвиток з часом.

Розширюваність: LinkML розроблений таким чином, щоб бути розширюваним, дозволяючи користувачам визначати власні типи даних, обмеження та інші елементи моделювання відповідно до їхніх конкретних потреб.

Спільнота та документація: LinkML має активну спільноту користувачів і розробників, які роблять внесок у його розвиток і надають підтримку. Доступна вичерпна документація та приклади, які допоможуть користувачам розпочати роботу з LinkML.

Сфери застосування: LinkML особливо популярний у сферах наук про життя та біоінформатики для моделювання біологічних даних, але його можна застосовувати в інших галузях, таких як фінанси, геопросторові дані тощо.

Етапи створення онтології

Існує декілька відомих формальних процесів розробки онтології: Ontology101, Methontology, LOT (Linked Open Terms).

Ontology101 (Онтологія101)

Згідно з Ontology101, процес розробки онтології включає кілька етапів, які виконуються під час створення онтології для представлення знань у певній області. Цей процес має важливе значення для структурування та організації інформації для різних програм, включаючи інтеграцію даних, управління

знаннями та технології семантичної мережі. Ось детальні кроки процесу розробки онтології:

1. **Identify the Domain and Scope (*Визначте домен і область*):** Визначте область, для якої ви розробляєте онтологію. Ця сфера може бути будь-якою сферою інтересів, як-от біологія, фінанси чи транспорт. Визначте обсяг вашого онтологічного проекту, вказавши, які аспекти домену ви маєте намір охопити, а які плануєте виключити.
2. **Gather Requirements (*Зберіть вимоги*):** Зберіть і задокументуйте вимоги до онтології. Це передбачає розуміння цілей і завдань онтології, а також конкретних потреб і очікувань її користувачів.
3. **Conceptualization (*Концептуалізація*):** Почніть з визначення та визначення ключових понять у межах домену. Концепції представляють об’єкти, сутності або ідеї, які мають відношення до домену. Визначте відношення між поняттями. Ці відношення описують, як концепти пов’язані або пов’язані в межах домену.
4. **Select an Ontology Language (*Виберіть мову онтології*):** Виберіть мову онтології для представлення вашої онтології. Загальні варіанти включають OWL (мова веб-онтології), RDF (система опису ресурсів), RDFS (схема RDF) або інші, залежно від ваших потреб і складності онтології.
5. **Design the Ontology Structure (*Спроектуйте структуру онтології*):** Створіть попередню структуру для вашої онтології, включаючи класи (поняття), властивості (атрибути та зв’язки) та екземпляри (індивідууми). Визначте ієрархію класів, вказуючи батьківсько-дочірні зв’язки, які представляють таксономію понять у домені.
6. **Specify Properties and Relationships (*Вкажіть властивості та зв’язки*):** Визначте властивості для опису атрибутів і зв’язків між поняттями. Укажіть кількість властивостей (наприклад, один-до-одного, один-до-багатьох), щоб вказати кількість зв’язків, дозволених між примірниками.

7. **Create Instances (*Створіть екземпляри*):** Заповніть онтологію екземплярами, які є конкретними особами або об’єктами з домену. Переконайтеся, що екземпляри пов’язані з відповідними класами та мають значення для відповідних властивостей.
8. **Model Complex Relationships (*Модель складних відносин*):** Якщо домен містить складні зв’язки або аксіоми, використовуйте вибрану мову онтології для їх представлення. Наприклад, визначте правила, обмеження або логіку, щоб отримати більш складні знання.
9. **Validation and Testing (*Перевірка та тестування*):** Перевірте онтологію, щоб переконатися, що вона відповідає визначеним обмеженням і вимогам. Використовуйте інструменти міркування та валідатори, щоб перевірити наявність невідповідностей, відсутньої інформації або логічних помилок в онтології.
10. **Documentation and Metadata (*Документація та метадані*):** Задокументуйте онтологію, надаючи описи, мітки, визначення та інші метадані для класів, властивостей та екземплярів. Включіть документацію, яка пояснює мету, область і використання онтології.
11. **Evaluation and Review (*Оцінка та перегляд*):** Співпрацюйте з експертами в області та зацікавленими сторонами, щоб переглянути та оцінити онтологію на предмет точності, повноти та відповідності. Внесіть необхідні зміни на основі відгуків.
12. **Deployment and Integration (*Розгортання та інтеграція*):** Після того, як онтологія завершена та перевірена, розгорніть її у відповідному середовищі або інтегруйте в цільову програму чи систему.
13. **Maintenance and Evolution (*Технічне обслуговування та розвиток*):** Продовжуйте підтримувати та розвивати онтологію в міру розвитку домену або появи нових вимог. Підтримуйте онтологію в актуальному стані, щоб забезпечити її актуальність і точність.

Процес розробки онтології є повторюваним і може передбачати перегляд попередніх кроків у міру розвитку онтології або отримання додаткових знань. Важливо залучати експертів у галузі та зацікавлених сторін протягом усього процесу, щоб переконатися, що онтологія точно представляє знання в межах вибраної області.

Methontology (Методологія)

Методологія − це усталена методологія розробки онтологій, яка забезпечує структурований і систематичний підхід до створення онтологій. Методологія, розроблена групою дослідників, містить найкращі практики та рекомендації щодо розробки онтології. Нижче наведено ключові етапи процесу розробки онтології відповідно до методології:

1. **Specification (*Специфікація*):** Процес розробки онтології починається з визначення мети та обсягу онтології. Визначте область інтересів і цілі, яких ви хочете досягти за допомогою онтології.
2. **Knowledge Acquisition (*Отримання знань*):** Збирайте знання та інформацію, пов’язану з доменом. Цей крок передбачає визначення та консультування експертів у галузі, аналіз існуючих документів, баз даних та інших джерел знань у галузі.
3. **Preliminary Design (*Ескізний проект*):** Створіть попередній проект для структури онтології. Визначте ключові поняття, зв’язки та аксіоми, які використовуватимуться для представлення знань у предметній області.
4. **Formalization (*Формалізація*):** Формалізуйте онтологію, вибравши мову онтології (наприклад, OWL, RDF, RDFS) і представивши структуру онтології, концепції, зв’язки та обмеження за допомогою вибраної мови.
5. **Implementation (*Реалізація*):** Реалізуйте онтологію за допомогою засобів розробки онтології або редакторів, які підтримують вибрану мову онтології. Наповніть онтологію класами, властивостями та екземплярами.

6. **Evaluation (*Оцінка*):** Оцініть онтологію на предмет правильності, повноти та узгодженості. Цей крок передбачає як автоматичну перевірку, так і перевірку вручну. Переконайтеся, що онтологія відповідає вказаним вимогам.
7. **Integration (*Інтеграція*):** Інтегруйте онтологію в цільову програму або систему, де вона буде використовуватися. Переконайтеся, що він може ефективно представляти та організовувати знання для запланованої мети.
8. **Documentation (*Документація*):** Створіть повну документацію для онтології. Задокументуйте призначення онтології, область, структуру, класи, властивості, зв’язки та будь-які конкретні вказівки щодо використання.
9. **Maintenance and Evolution (*Технічне обслуговування та розвиток*):** Встановіть процес підтримки та розвитку онтології з часом. Підтримуйте онтологію в актуальному стані, оскільки домен розвивається та нові знання стають доступними.
10. **Validation and Verification (*Тестування та перевірки*):** Перевірте точність і функціональність онтології за допомогою процедур тестування та перевірки. Перевірте наявність логічних невідповідностей і переконайтеся, що онтологія поводиться належним чином.
11. **Evaluation by Domain Experts (*Оцінка доменних експертів*):** Співпрацюйте з експертами в області та зацікавленими сторонами, щоб переглянути й оцінити точність і релевантність онтології. Внесіть зміни на основі їхніх відгуків.
12. **Deployment (*Розгортання*):** Розгорніть онтологію у виробничому середовищі, де її можна використовувати для підтримки різноманітних програм, таких як керування знаннями, інтеграція даних або семантичний пошук.

13. **Quality Assurance (*Гарантія якості*):** Переконайтеся, що онтологія відповідає стандартам забезпечення якості, включаючи найкращі методи розробки онтології, щоб підтримувати її загальну якість.

Методологія підкреслює важливість співпраці з експертами в галузі та зацікавленими сторонами протягом усього процесу розробки онтології, щоб гарантувати, що результуюча онтологія точно представляє знання в межах вибраної області. Ця методологія забезпечує структурований і добре задокументований підхід до розробки онтології, що робить її цінним ресурсом для розробників і дослідників онтології.

LOT (Linked Open Terms / пов’язані відкриті умови)

Linked Open Terms (LOT) − це легка методологія для розробки та публікації контрольованих словників і термінологічних ресурсів у мережі як пов’язаних даних. Він дотримується принципів семантичної мережі та пов’язаних даних, щоб забезпечити легку інтеграцію та пошук термінологічних ресурсів. Процес розробки LOT включає кілька ключових етапів:

1. **Define the Scope (*Визначте сферу застосування*):** Визначте обсяг і домен ресурсу контрольованої лексики чи термінології, який ви збираєтеся розробити. Чітко визначте, що охоплюватиме ресурс та його призначення.
2. **Conceptual Modeling (*Концептуальне моделювання*):** Створити концептуальну модель термінологічного ресурсу, визначивши ключові поняття та зв’язки. Визначте ієрархії, синоніми та інші семантичні аспекти термінології.
3. **Select a Vocabulary Format (*Виберіть формат словника*):** Виберіть формат словника для представлення термінології. До поширених форматів належать SKOS (проста система організації знань), RDF (система опису ресурсів) або OWL (мова веб-онтології), усі вони підходять для зв’язаних даних.

4. **Create the Vocabulary (*Створіть словниковий запас*):** Наповніть словник термінами, поняттями та пов’язаними з ними метаданими (мітки, визначення, синоніми тощо). Встановіть зв’язки між термінами, наприклад ширші/вужчі зв’язки або пов’язані поняття.
5. **Publish as Linked Data (*Опублікувати як пов’язані дані*):** Опублікуйте ресурс контрольованої лексики чи термінології в Інтернеті як пов’язані дані. Переконайтеся, що кожен термін і концепція має унікальний URI (уніфікований ідентифікатор ресурсу), щоб на нього можна було посилатися окремо та посилатися на нього.
6. **Use RDF Serialization (*Використовуйте серіалізацію RDF*):** Серіалізуйте лексику в RDF (наприклад, RDF/XML, Turtle, JSON-LD), щоб зробити її машиночитаною та зрозумілою технологіями семантичної мережі.
7. **Enable Dereferencing (*Увімкніть розіменування*):** Налаштуйте сервер, на якому розміщено словник, щоб увімкнути розіменування URI. Це означає, що коли доступ до URI здійснюється через HTTP, він повинен повертати дані RDF про ресурс.
8. **Create an RDF/XML or TTL File (*Створіть файл RDF/XML або TTL*):** Надайте файл RDF/XML або TTL (Turtle), який містить дані RDF для словника. Цей файл має бути доступним через HTTP і мати посилання з головної цільової сторінки словника.
9. **Link to Other Resources (*Посилання на інші ресурси*):** Пов’яжіть контрольований словник з іншими відповідними ресурсами та онтологіями в Інтернеті за допомогою предикатів RDF, таких як skos:related, rdfs:seeAlso або owl:sameAs.
10. **Metadata and Documentation (*Метадані та документація*):** Надайте метадані та документацію для словника. Включіть інформацію про творців, версії, ліцензування та правила використання.

11. **Promote Discoverability (*Сприяти видимості*):** Сприяйте видимості свого ресурсу Linked Open Terms, зареєструвавши його у відповідних репозиторіях і каталогах словників, таких як каталог Linked Open Vocabularies (LOV).
12. **Maintain and Update (*Підтримуйте та оновлюйте*):** Регулярно підтримуйте та оновлюйте словниковий запас, щоб він був актуальним і актуальним. Зверніть увагу на відгуки користувачів і експертів домену для вдосконалення.
13. **Community Involvement (*Залучення громади*):** Заохочуйте участь громади та внесок у словниковий запас. Зробіть його відкритим для співпраці та відгуків.

Linked Open Terms (LOT) дотримуються принципів відкритості, сумісності та можливості виявлення, гарантуючи, що термінологічні ресурси легко доступні та можуть бути інтегровані в широкий спектр програм і систем знань у семантичній мережі.

Порівняння Ontology101", "Methontology" та "Linked Open Terms (LOT)

"Ontology101", "Methontology" та "Linked Open Terms (LOT)" − це три різні підходи або методології, пов'язані з онтологіями та розробкою термінології. Давайте порівняємо ці підходи з точки зору їхньої спрямованості, процесів і ключових принципів:

1. Фокус:

Ontology101 − це вступний курс або посібник, який забезпечує базове розуміння онтологій та їх принципів.

Він зосереджений на поясненні основних концепцій, компонентів і використання онтологій, що робить його доступним для початківців.

Methontology − це комплексна методологія для розробки онтології, яка наголошує на структурованих і систематичних процесах. Він розроблений для

побудови складних онтологій для певних областей, з великим акцентом на інженерії знань та формалізації.

Linked Open Terms (LOT): LOT − це легка методологія для розробки та публікації контрольованих словників і термінологічних ресурсів як пов’язаних даних. Його основна увага зосереджена на тому, щоб термінологічні ресурси були доступні в Інтернеті в стандартизованому, машиночитаному форматі.

2. Процес розробки:

Ontology101 надає високорівневий огляд розробки онтології без заглиблення в конкретні етапи чи формальні процеси. Він охоплює основні концепції та принципи, але не пропонує структурованої методології розробки.

Methontology окреслює структурований процес розробки онтології з такими етапами, як специфікація, отримання знань, формалізація, впровадження, перевірка тощо. Він призначений для розробки складних і предметно-орієнтованих онтологій.

Linked Open Terms (LOT) зосереджені на спрощеній розробці та публікації контрольованих словників. Його процес включає концептуальне моделювання, створення словника, публікацію як пов’язаних даних і сприяння видимості.

3. Формалізація та складність:

Ontology101 не зосереджена на формалізації чи складному представленні знань. Він забезпечує базове розуміння принципів онтології без занурення у формальну логіку.

Методологія наголошує на формалізації та використовує такі мови онтології, як OWL і RDF. Він підходить для розробки складних онтологій, які потребують формальної семантики.

Пов’язані відкриті умови (LOT) є відносно легкими та мають на меті надати прості термінологічні ресурси як пов’язані дані. Він не передбачає такого ж рівня формалізації та складності, як методологія.

4. Цільова аудиторія:

Ontology101 призначений для початківців і тих, хто шукає вступне розуміння онтологій. Він доступний для широкої аудиторії, включаючи неспеціалістів.

Methontology розроблена для інженерів знань, експертів у предметній області та розробників онтологій. Це вимагає глибшого розуміння моделювання онтології та формальної логіки.

Пов’язані відкриті терміни (LOT) підходять для тих, хто зацікавлений у публікації контрольованих словників і термінологічних ресурсів як пов’язаних даних. Він може зацікавити низку професіоналів, у тому числі бібліотекарів, термінологів і практиків семантичної мережі.

Підводячи підсумок, **Ontology101** забезпечує базовий вступ до онтологій, **Methontology** пропонує структуровану методологію для розробки складних онтологій, а **Linked Open Terms (LOT)** − це легкий підхід для публікації термінологічних ресурсів як пов’язаних даних. Вибір підходу залежить від конкретних цілей, досвіду та складності проекту розробки онтології чи термінології.

Semantic Web

Semantic Web − це набір стандартів та інструментів, які спрямовані на те, щоб зробити веб більш інтелектуальним та машиночитаним. Семантична павутина ***(Semantic Web)*** будується на вершині існуючої веб-інфраструктури, щоб створити більш взаємопов’язану та змістовну павутину.

Основною технологією Semantic Web є ***Resource Description Framework (RDF)***. RDF надає стандартний спосіб опису ресурсів в Інтернеті, таких як веб-

сторінки, зображення та відео, таким чином, щоб комп’ютери могли зрозуміти. RDF використовує модель даних на основі графів, де ресурси представлені як вузли, а відносини між ресурсами представлені як ребра.

Крім RDF, Semantic Web також включає кілька інших технологій, таких як **OWL** ***(Web Ontology Language)*** і **SPARQL (*SPARQL Protocol and RDF Query Language)***. **OWL** − мова для визначення онтологій, які є формальними описами понять і відносин в певній області. ***SPARQL*** − мова запитів, що використовується для отримання та маніпулювання даними RDF.

Semantic Web має кілька переваг, таких як забезпечення кращої інтеграції даних і оперативної сумісності в різних додатках і системах, полегшення автоматизованої обробки і аналізу даних, а також надання можливості інтелектуальним агентам міркувати і приймати рішення, засновані на семантичному значенні даних.

Деякі real-world applications of Semantic Web technologies включають в себе розробку графів знань, які являють собою великі, взаємопов’язані колекції структурованих даних, а також використання семантичних пошукових систем, здатних розуміти запити природної мови і надавати більш точні і відповідні результати пошуку.

В цілому, Semantic Web technologies є важливим напрямком досліджень і розробок в області штучного інтелекту і мають потенціал для значного розширення можливостей Інтернету і включення нових та інноваційних додатків.

Real-world applications of Semantic Web

Ось кілька вражаючих прикладів Semantic Web technologies:

1. **DBpedia: DBpedia** − це проект, керований спільнотою, який витягує структуровані дані з Wikipedia та робить їх доступними як пов’язані дані. Проект створив величезний граф знань, який може бути використаний для

широкого кола застосувань, включаючи інтеграцію даних, управління знаннями та пошук інформації.

2. **Google’s Knowledge Graph: Google’s Knowledge Graph** − це база знань, яка надає структуровану та детальну інформацію про людей, місця та речі. Він використовує комбінацію машинного навчання та Semantic Web technologies, щоб зрозуміти сенс пошукових запитів та забезпечити більш точні та відповідні результати.
3. **Amazon’s Alexa: Amazon’s Alexa** − це голосовий помічник, який використовує обробку природної мови та Semantic Web technologies для розуміння та відповіді на запити користувачів. Alexa може відповідати на запитання, відтворювати музику, керувати пристроями розумного будинку та виконувати широкий спектр інших завдань.
4. **Watson Assistant: Watson Assistant** − це платформа чат-ботів, яка використовує обробку природної мови та семантичні веб-технології для розуміння та відповіді на запити користувачів. Він може бути налаштований для підтримки широкого спектру додатків, включаючи обслуговування клієнтів, продажі та маркетинг.
5. **OpenStreetMap: OpenStreetMap** − це мапа, яка використовує Semantic Web technologies для надання структурованих даних про географічні розташування. Карта може бути використана для широкого спектру застосувань, включаючи навігацію, містобудування та реагування на катастрофи.
6. **BBC’s Linked Data Platform:** BBC розробила Linked Data Platform, яка надає структуровані дані про свої програми та послуги. Платформа дозволяє користувачам відкривати відповідний контент і надає рекомендації, засновані на їх інтересах.

В цілому, ці додатки демонструють силу та потенціал семантичних веб-технологій для забезпечення інтелектуальних і взаємопов’язаних систем і надання цінної інформації.

DBpedia

DBpedia − це проект, керований спільнотою, який витягує структуровані дані з Wikipedia та робить їх доступними як пов'язані дані. Мета DBpedia − забезпечити загальне представлення даних знань, що містяться у Wikipedia, і дозволити використовувати їх для широкого кола застосувань.

DBpedia збирає свої дані через процес, який називається видобуванням. Видобуток включає в себе розбір сирого тексту статей Wikipedia і перетворення його в структуровані дані, які можуть бути збережені в графі знань. Процес вилучення включає в себе кілька кроків, включаючи виявлення сутностей і концепцій в тексті, відображення їх на існуючі ресурси в графіку знань, а також створення нових ресурсів і відносин у міру необхідності.

Окрім вилучення даних з Wikipedia, DBpedia також посилається на зовнішні джерела даних для збагачення свого графа знань. Ці зовнішні джерела включають структуровані дані з таких джерел, як GeoNames, MusicBrainz і CIA World Factbook, а також неструктуровані дані з таких джерел, як Flickr і Linked Movie Database.

DBpedia робить свої дані доступними безкоштовно в Інтернеті як серію завантажуваних RDF-файлів. Дані також доступні через кінцеву точку SPARQL, що дозволяє користувачам запитувати дані за допомогою мови запитів SPARQL. Крім того, DBpedia надає цілий ряд інструментів та послуг для підтримки використання та аналізу своїх даних, включаючи веб-інтерфейс запитів, інструмент відображення для створення відображень між статтями Wikipedia та зовнішніми ресурсами, та інструмент візуалізації графів знань.

Загалом DBpedia є цінним ресурсом для тих, хто зацікавлений у використанні структурованих знань, що містяться у Wikipedia, для досліджень,

розробок або інших застосувань. Забезпечуючи загальне представлення даних цих знань, DBpedia дозволяє користувачам легко отримувати доступ і використовувати ці дані у власних проектах і додатках.

OpenStreetMap

OpenStreetMap (OSM) − картографічна платформа, що надає користувачам можливість вносити та редагувати географічні дані. OSM використовує кілька семантичних веб-технологій для надання структурованих даних про географічні розташування та забезпечення сумісності своїх даних з іншими системами.

Однією з ключових семантичних веб-технологій, що використовуються OSM, є **Resource Description Framework (RDF)**. OSM використовує RDF для забезпечення загальної моделі даних для представлення географічних даних, що дозволяє користувачам легко ділитися та інтегрувати дані OSM з іншими системами.

OSM також використовує **Simple Knowledge Organization System (SKOS)**, щоб забезпечити стандартизований спосіб опису понять і відносин, пов’язаних з географічними даними. **SKOS** дозволяє користувачам створювати та підтримувати словники та онтології, які описують відносини між різними типами географічних особливостей, таких як дороги, будівлі та пам’ятки.

Крім того, OSM використовує **Web Ontology Language (OWL)** для визначення формальних онтологій, які описують відносини між географічними особливостями та поняттями. OWL дозволяє OSM створювати більш складні і структуровані представлення географічних даних, які можуть бути використані для підтримки більш просунутих додатків і аналізів.

В цілому, використання **Semantic Web** в **OpenStreetMap** дозволяє користувачам легко ділитися, інтегрувати та аналізувати географічні дані, а також забезпечує загальну модель даних, яка може використовуватися іншими системами та додатками.

SPARQL

SPARQL ***(вимовляється як “sparkle”)*** − це мова запитів, яка використовується для отримання даних із баз даних **RDF** ***(Resource Description Framework)***. RDF − це модель даних, яка використовується для представлення інформації в машиночитаному форматі, а SPARQL дозволяє запитувати цю інформацію за допомогою синтаксису, схожого на SQL (мова структурованих запитів).

SPARQL використовується для запиту даних у семантичній павутині, яка є розширенням Всесвітньої павутини, що дозволяє обмінюватися та повторно використовувати дані між програмами, підприємствами та спільнотами. Він розроблений як гнучкий, виразний і потужний, що дозволяє користувачам отримувати дані на основі складних критеріїв і зв’язків.

Запити SPARQL зазвичай пишуться в текстовому редакторі, а потім виконуються в базі даних RDF за допомогою механізму SPARQL. Результати запиту можна повернути в різних форматах, включаючи XML, JSON і CSV.

Загалом, SPARQL є важливим інструментом для роботи з даними RDF і максимального використання можливостей семантичної мережі.

SPARQL пропонує низку операторів для забезпечення запиту даних RDF:

1. **`SELECT:`** цей оператор визначає змінні, які будуть повернуті в результаті запиту. Він використовується для отримання даних із бази даних RDF.
2. **`WHERE:`** Цей оператор визначає шаблони графів або потрійні шаблони, які мають бути задоволені, щоб запит повернув результат. Він використовується для визначення критеріїв відбору даних.
3. **`FILTER:`** Цей оператор використовується для фільтрації результатів запиту на основі певних умов. Його можна використовувати для виконання

основних арифметичних операцій, операцій із рядками та інших операцій порівняння.

4. **OPTIONAL:** Цей оператор визначає, що результат запиту має включати необов’язкові шаблони графів. Він використовується, коли в базі даних RDF для певної змінної можуть бути або не відповідати дані.
5. **UNION:** Цей оператор об’єднує результати двох або більше запитів в один результат. Він використовується для об’єднання даних із кількох джерел.
6. **GROUP BY:** цей оператор групує результати запиту на основі вказаної змінної. Він використовується для групування даних на основі загальних властивостей.
7. **ORDER BY:** Цей оператор упорядковує результати запиту на основі вказаної змінної. Він використовується для сортування даних у порядку зростання або спадання.

Приклади

Ось приклад оператора SELECT у запиті SPARQL:

Припустімо, у вас є RDF-граф, який описує книги та їхніх авторів із даними в такому форматі:

```
@prefix ex: <http://example.com/> .

ex:book1 ex:title "The Great Gatsby" .
ex:book1 ex:author "F. Scott Fitzgerald" .
ex:book2 ex:title "To Kill a Mockingbird" .
ex:book2 ex:author "Harper Lee" .
ex:book3 ex:title "1984" .
ex:book3 ex:author "George Orwell" .
ex:book3 ex:published "1954" .
```

Щоб отримати список усіх назв книг і їхніх авторів, ви можете використати наступний запит SPARQL з оператором SELECT:

```
PREFIX ex: <http://example.com/>

SELECT ?title ?author
WHERE {
  ?book ex:title ?title ;
        ex:author ?author .
}
```

```
PREFIX ex: <http://example.com/>

SELECT ?title ?author ?book
WHERE {
  ?book ex:title ?title .
  ?book ex:author ?author .
}
```

Цей запит вибирає змінні ?title та ?author, які будуть повернуті в результаті. Він використовує оператор WHERE для визначення критеріїв вибору даних, якими в даному випадку є назва книги та автор. Результатом буде:

```
title                    | author
-------------------------|-----------------------
"The Great Gatsby"        | "F. Scott Fitzgerald"
"To Kill a Mockingbird" | "Harper Lee"
"1984"                   | "George Orwell"
```

Ось приклад, який розширює попередній приклад, додаючи оператор FILTER до запиту SPARQL:

Припустімо, ви хочете отримати список усіх назв книг і авторів, але ви хочете включити лише книги, назва яких починається з літери “T”. Ви можете використати наступний запит SPARQL з операторами SELECT і FILTER:

```
PREFIX ex: <http://example.com/>

SELECT ?title ?author
WHERE {
  ?book ex:title ?title ;
     ex:author ?author .
  FILTER (regex(?title, "^T", "i"))
}
```

Цей запит вибирає змінні ?title та ?author, які будуть повернуті в результаті. Він використовує оператор WHERE, щоб визначити критерії вибору даних, якими є назва книги та автор. Потім оператор FILTER використовується для фільтрації результатів на основі шаблону регулярного виразу. У цьому

випадку шаблон “^T” відповідає будь-якому рядку, який починається з літери “T”, а прапорець “i” робить відповідність нечутливою до регістру.

Результатом буде

```
title                   | author
------------------------|----------------------
"To Kill a Mockingbird" | "Harper Lee"
"The Great Gatsby"       | "F. Scott Fitzgerald"
```

Зверніть увагу, що в результат включено лише книги, назви яких починається на літеру “T”.

Ось приклад оператора OPTIONAL у запиті SPARQL:

Припустімо, у вас є RDF-граф, який описує книги, їхніх авторів і видавців із даними в такому форматі:

```
@prefix ex: <http://example.com/> .

ex:book1 ex:title "The Great Gatsby" .
ex:book1 ex:author "F. Scott Fitzgerald" .
ex:book2 ex:title "To Kill a Mockingbird" .
ex:book2 ex:author "Harper Lee" .
ex:book3 ex:title "1984" .
ex:book3 ex:author "George Orwell" .
ex:book3 ex:publisher "Penguin Books" .
```

Щоб отримати список усіх книг і їхніх видавців, якщо доступний, ви можете використати наступний запит SPARQL з операторами SELECT і OPTIONAL:

```
PREFIX ex: <http://example.com/>

SELECT ?title ?publisher
WHERE {
  ?book ex:title ?title .
  OPTIONAL { ?book ex:publisher ?publisher }
}
```

Цей запит вибирає змінні ?title і ?publisher, які будуть повернуті в результаті. Він використовує оператор WHERE, щоб визначити критерії вибору даних, якими є назва книги. Потім оператор OPTIONAL використовується для визначення того, що змінна видавця є необов’язковою, тобто вона може існувати або не існувати для певної книги.

Результатом буде:

```
title                   | publisher
------------------------|----------------------
"The Great Gatsby"      |
"To Kill a Mockingbird" |
"1984"                  | "Penguin Books"
```

Зауважте, що назви книг завжди включаються в результат, але видавець включається лише за наявності. У цьому випадку лише книга з назвою “1984” має видавця в списку, тому це єдина книга зі значенням у стовпці видавця.

Ось приклад оператора GROUP BY у запиті SPARQL:

Припустімо, у вас є RDF-граф, який описує книги та їхніх авторів із даними в такому форматі:

```
@prefix ex: <http://example.com/> .

ex:book1 ex:title "The Great Gatsby" .
ex:book1 ex:author "F. Scott Fitzgerald" .
ex:book2 ex:title "To Kill a Mockingbird" .
ex:book2 ex:author "Harper Lee" .
ex:book3 ex:title "1984" .
ex:book3 ex:author "George Orwell" .
ex:book4 ex:title "Animal Farm" .
ex:book4 ex:author "George Orwell" .
```

Щоб отримати список авторів і кількість написаних ними книг, можна використати наступний запит SPARQL з операторами SELECT, COUNT і GROUP BY:

```
PREFIX ex: <http://example.com/>

SELECT ?author (COUNT(?book) AS ?numBooks)
WHERE {
  ?book ex:author ?author .
}
GROUP BY ?author
```

Цей запит вибирає змінні ?author і кількість книг, які вони написали ? numBooks. Він використовує оператор WHERE, щоб визначити критерії вибору даних, якими є автор та пов’язана книга(и). Потім оператор COUNT використовується для підрахунку кількості книг на автора, а оператор GROUP BY використовується для групування результату за автором.

Результатом буде:

```
author                 | numBooks
-----------------------|---------
"F. Scott Fitzgerald" | 1
"George Orwell"       | 2
"Harper Lee"          | 1
```

SPARQL має оператори для зміни даних RDF, включаючи додавання нових даних. Двома основними операторами для зміни даних у SPARQL є INSERT і DELETE.

Оператор INSERT використовується для додавання нових RDF-трійок до існуючого графіка, тоді як оператор DELETE використовується для видалення трійок з графіка. Ці оператори зазвичай використовуються в запитах SPARQL Update, які є окремим типом запитів від запитів SPARQL Select, які використовуються для отримання даних.

Ось приклад використання оператора INSERT у запиті на оновлення SPARQL для додавання нової книги до існуючого графа RDF:

```
PREFIX ex: <http://example.com/>

INSERT DATA {
  ex:book4 ex:title "The Catcher in the Rye" .
  ex:book4 ex:author "J.D. Salinger" .
}
```

У цьому запиті використовується оператор PREFIX, щоб визначити префікс простору імен для URI "http://example.com/", і оператор INSERT DATA, щоб указати дані, які потрібно додати до графіка. У цьому випадку запит додає до графу нову книгу з назвою “The Catcher in the Rye” та автором “J.D. Salinger”. Зверніть увагу, що оператор INSERT можна використовувати з іншими операторами SPARQL, такими як WHERE і FILTER, щоб додати складніші шаблони даних до графіка.

Ось кілька прикладів використання оператора INSERT з оператором WHERE у запиті на оновлення SPARQL для додавання нових даних на основі наявних даних:

Припустімо, у вас є RDF-граф, який описує книги та їхніх авторів із даними в такому форматі:

```
@prefix ex: <http://example.com/> .

ex:book1 ex:title "The Great Gatsby" .
ex:book1 ex:author "F. Scott Fitzgerald" .
ex:book2 ex:title "To Kill a Mockingbird" .
ex:book2 ex:author "Harper Lee" .
ex:book3 ex:title "1984" .
ex:book3 ex:author "George Orwell" .
```

Щоб додати нову книгу до графіка з тим же автором, що й наявна книга, ви можете використати такий запит SPARQL Update з операторами INSERT і WHERE:

```
PREFIX ex: <http://example.com/>

INSERT {
  ex:book4 ex:title "Animal Farm" .
  ex:book4 ex:author ?author .
}
WHERE {
  ?book ex:title "1984" .
  ?book ex:author ?author .
}
```

У цьому запиті використовується оператор PREFIX, щоб визначити префікс простору імен для URI "http://example.com/", і оператор INSERT, щоб указати дані, які потрібно додати до графіка. У цьому випадку запит додає нову книгу з назвою “Колгосп тварин” і автора книги з назвою “1984”. Оператор WHERE використовується для визначення критеріїв вибору автора, яким є автор книги з назвою “1984”.

Результатом буде:

```
ex:book1 ex:title "The Great Gatsby" .
ex:book1 ex:author "F. Scott Fitzgerald" .
ex:book2 ex:title "To Kill a Mockingbird" .
ex:book2 ex:author "Harper Lee" .
ex:book3 ex:title "1984" .
ex:book3 ex:author "George Orwell" .
ex:book4 ex:title "Animal Farm" .
ex:book4 ex:author "George Orwell" .
```

Зауважте, що на графік додано нову книгу під назвою “Animal Farm” того ж автора, що й книга під назвою “1984”.

Іншим прикладом використання операторів INSERT і WHERE для додавання нових даних на основі існуючих є додавання нової властивості до існуючої трійки. Наприклад, щоб додати нового видавця до книги під назвою “The Great Gatsby”, ви можете використати такий запит SPARQL Update:

```
PREFIX ex: <http://example.com/>

INSERT {
  ex:book1 ex:publisher "Scribner" .
}
WHERE {
  ex:book1 ex:title "The Great Gatsby" .
}
```

У цьому запиті використовується оператор INSERT, щоб додати нову властивість “publisher” до трійки з суб’єктом “ex:book1” і об’єктом “The Great Gatsby”. Оператор WHERE використовується для визначення трійки, яку слід змінити.

Результатом буде:

```
ex:book1 ex:title "The Great Gatsby" .
ex:book1 ex:author "F. Scott Fitzgerald" .
ex:book1 ex:publisher "Scribner" .
ex:book2 ex:title "To Kill a Mockingbird" .
ex:book2 ex:author "Harper Lee" .
ex:book3 ex:title "1984" .
ex:book3 ex:author "George Orwell" .
```

Зверніть увагу, що нова властивість `ex:publisher` додана до трійки для книги під назвою “The Great Gatsby”.

Рекомендована література

1. Kotis K., Spiliotopoulos D. Semantic Web Technology and Recommender Systems, Eds. Published: April 2023 P. 284. ISBN 978-3-0365-7210-9 (hardback); ISBN 978-3-0365-7211-6 (PDF) URL :https://doi.org/10.3390/books978-3-0365-7211-6
2. Barrasa J., Webber J. Building Knowledge Graphs. Released June 2023 Publisher(s): O'Reilly Media, Inc. ISBN: 9781098127107
3. Hitzler P., Krötzsch M., Rudolph S. Foundations of Semantic Web Technologies. Chapman & Hall/CRC, 2009. P. 455.
4. What is a Knowledge Base? Ontotext Fundamentals, 2023. URL : https://cutt.ly/bwkuvPUu
5. Debellis M., A Practical Guide to Building OWL Ontologies Using Protégé 5.5 and Plugins, 2021. URL: https://cutt.ly/pwjKJH7N
6. Natalya F. Noy and Deborah L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, 2001.
7. Gruber T. R. Towards principles for the design of ontologies used for knowledge sharing. T. R. Gruber. Formal Ontology in Conceptual Analysis and Knowledge Representation. 1993.
8. Guarino N. What Is an Ontology? N. Guarino, D. Oberle, S. Staab. International Handbooks on Information Systems. 2009.
9. Jean S. Domain Ontologies: A Database-Oriented Analysis. S. Jean, J. Pierra, Y. Ait-Ameur. Web Information Systems and Technologies, International Conferences, WEBIST 2005 and WEBIST 2006. Revised Selected Papers. 2007. P. 238-254.

10. The Gene Ontology Consortium. (2019) “The Gene Ontology Resource: 20 years and still GOing strong”. Nucleic acids research, 47(D1), D330-D338.
11. Ontology engineering: Current state, challenges, and future directions December 2019 Semantic Web 11(1):1-14. DOI: 10.3233/SW-190382
12. Protégé Official site. URL : https://protege.stanford.edu/