Select AI for Python

Release 1.0.0b1

Oracle

Aug 16, 2025

CONTENTS

`select_ai` is a Python module which enables integrating DBMS_CLOUD_AI `PL/SQL` package into Python workflows. It bridges the gap between `PL/SQL` package's AI capabilities and Python's rich ecosystem.

CHAPTER

ONE

GETTING STARTED

1.1 Introduction to Select AI for Python

`select_ai` is a Python module that helps you invoke DBMS_CLOUD_AI using Python. It supports text-to-SQL generation, retrieval augmented generation (RAG), synthetic data generation, and several other features using Oracle-based and third-party AI providers.

`select_ai` supports both synchronous and concurrent(asynchronous) programming styles.

The Select AI Python API supports Python versions 3.9, 3.10, 3.11, 3.12 and 3.13.

1.2 Installing `select_ai`

1.2.1 Installation requirements

To use `select_ai` you need:

- Python 3.9, 3.10, 3.11, 3.12 or 3.13
- `python-oracledb` - This package is automatically installed as a dependency requirement
- `pandas` - This package is automatically installed as a dependency requirement

1.2.2 `select_ai` installation

`select_ai` can be installed from Python's package repository PyPI using pip.

1. Install Python 3 if it is not already available. Use any version from Python 3.9 through 3.13.
2. Install `select_ai`:

```
python3 -m pip install select_ai --upgrade --user
```

3. If you are behind a proxy, use the `--proxy` option. For example:

```
python3 -m pip install select_ai --upgrade --user --proxy=http://proxy.example.
↪com:80
```

4. Create a file `select_ai_connection_test.py` such as:

```
import select_ai

user = "<your_db_user>"
password = "<your_db_password>"
dsn = "<your_db_dsn>"
select_ai.connect(user=user, password=password, dsn=dsn)
print("Connected to the Database")
```

5. Run `select_ai_connection_test.py`

```
python3 select_ai_connection_test.py
```

Enter the database password when prompted and message will be shown:

```
Connected to the Database
```

1.3 Connecting to Oracle Database

`select_ai` uses the Python thin driver i.e. `python-oracledb` to connect to the database and execute PL/SQL sub-programs.

1.3.1 Synchronous connection

To connect to an Oracle Database synchronously, use `select_ai.connect()` method as shown below

```
import select_ai

user = "<your_db_user>"
password = "<your_db_password>"
dsn = "<your_db_dsn>"
select_ai.connect(user=user, password=password, dsn=dsn)
```

1.3.2 Asynchronous connection

Asynchronous applications should use `select_ai.async_connect()` along with `await` keyword:

```
import select_ai

user = "<your_db_user>"
password = "<your_db_password>"
dsn = "<your_db_dsn>"
await select_ai.async_connect(user=user, password=password, dsn=dsn)
```

Note

For m-TLS (wallet) based connection, additional parameters like `wallet_location`, `wallet_password`, `https_proxy`, `https_proxy_port` can be passed to the `connect` and `async_connect` methods

CHAPTER

TWO

ACTIONS

An action in Select AI is a keyword that instructs Select AI to perform different behavior when acting on the prompt.

2.1 Supported Actions

Following list of actions can be performed using `select_ai`

Table 1: Select AI Actions

Actions	Enum	Description
chat	`select_ai.Action.CHAT`	Enables general conversations with the LLM, potentially for clarifying prompts, exploring data, or generating content.
explainsql	`select_ai.Action.EXPLAINSQL`	Explain the generated SQL query
narrate	`select_ai.Action.NARRATE`	Explains the output of the query in natural language, making the results accessible to users without deep technical expertise.
runsql	`select_ai.Action.RUNSQL`	Executes a SQL query generated from a natural language prompt. This is the default action.
showprompt	`select_ai.Action.SHOWPROMPT`	Show the details of the prompt sent to LLM
showsql	`select_ai.Action.SHOWSQL`	Displays the generated SQL statement without executing it.

CHAPTER

THREE

PROVIDER

An AI Provider in Select AI refers to the service provider of the LLM, transformer or both for processing and generating responses to natural language prompts. These providers offer models that can interpret and convert natural language for the use cases highlighted under the LLM concept.

See Select your AI Provider for the supported providers

3.1 Provider

class select_ai.**Provider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str | None = None*, *provider_endpoint: str | None = None*, *region: str | None = None*)

Base class for AI Provider

To create an object of Provider class, use any one of the concrete AI provider implementations

Parameters

- **embedding_model** (*str*) – The embedding model, also known as a transformer. Depending on the AI provider, the supported embedding models vary
- **model** (*str*) – The name of the LLM being used to generate responses
- **provider_name** (*str*) – The name of the provider being used
- **provider_endpoint** (*str*) – Endpoint URL of the AI provider being used
- **region** (*str*) – The cloud region of the Gen AI cluster

3.2 AnthropicProvider

class select_ai.**AnthropicProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'anthropic'*, *provider_endpoint: str | None = None*, *region: str | None = None*)

Anthropic specific attributes

3.3 AzureProvider

class select_ai.**AzureProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'azure'*, *provider_endpoint: str | None = None*, *region: str | None = None*, *azure_deployment_name: str | None = None*, *azure_embedding_deployment_name: str | None = None*, *azure_resource_name: str | None = None*)

Azure specific attributes

Parameters

- **azure_deployment_name** (*str*) – Name of the Azure OpenAI Service deployed model.
- **azure_embedding_deployment_name** (*str*) – Name of the Azure OpenAI deployed embedding model.
- **azure_resource_name** (*str*) – Name of the Azure OpenAI Service resource

3.4 AWSProvider

class select_ai.**AWSProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'aws'*, *provider_endpoint: str | None = None*, *region: str | None = None*, *aws_apiformat: str | None = None*)

AWS specific attributes

3.5 CohereProvider

class select_ai.**CohereProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'cohere'*, *provider_endpoint: str | None = None*, *region: str | None = None*)

Cohere AI specific attributes

3.6 OpenAIProvider

class select_ai.**OpenAIProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'openai'*, *provider_endpoint: str | None = 'api.openai.com'*, *region: str | None = None*)

OpenAI specific attributes

3.7 OCIGenAIProvider

class select_ai.**OCIGenAIProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'oci'*, *provider_endpoint: str | None = None*, *region: str | None = None*, *oci_apiformat: str | None = None*, *oci_compartment_id: str | None = None*, *oci_endpoint_id: str | None = None*, *oci_runtimetype: str | None = None*)

OCI Gen AI specific attributes

Parameters

- **oci_apiformat** (*str*) – Specifies the format in which the API expects data to be sent and received. Supported values are 'COHERE' and 'GENERIC'
- **oci_compartment_id** (*str*) – Specifies the OCID of the compartment you are permitted to access when calling the OCI Generative AI service
- **oci_endpoint_id** (*str*) – This attributes indicates the endpoint OCID of the Oracle dedicated AI hosting cluster
- **oci_runtimetype** (*str*) – This attribute indicates the runtime type of the provided model. The supported values are 'COHERE' and 'LLAMA'

3.8 GoogleProvider

class select_ai.**GoogleProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'google'*, *provider_endpoint: str | None = None*, *region: str | None = None*)

Google AI specific attributes

3.9 HuggingFaceProvider

class select_ai.**HuggingFaceProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'huggingface'*, *provider_endpoint: str | None = None*, *region: str | None = None*)

HuggingFace specific attributes

3.10 Enable AI service provider

> **Note**
>
> All sample scripts in this documentation read Oracle database connection details from the environment. Create a dotenv file `.env`, export the the following environment variables and source it before running the scripts.

```
export SELECT_AI_ADMIN_USER=<db_admin>
export SELECT_AI_ADMIN_PASSWORD=<db_admin_password>
export SELECT_AI_USER=<select_ai_db_user>
export SELECT_AI_PASSWORD=<select_ai_db_password>
export SELECT_AI_DB_CONNECT_STRING=<db_connect_string>
export TNS_ADMIN=<path/to/dir_containing_tnsnames.ora>
```

3.10.1 Sync API

This method grants execute privilege on the packages `DBMS_CLOUD`, `DBMS_CLOUD_AI` and `DBMS_CLOUD_PIPELINE`. It also enables the database user to invoke the AI(LLM) endpoint

```
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")

select_ai.connect(user=admin_user, password=password, dsn=dsn)
select_ai.enable_provider(
    users=select_ai_user, provider_endpoint="*.openai.azure.com"
)
print("Enabled AI provider for user: ", select_ai_user)
```

output:

```
Enabled AI provider for user: <select_ai_db_user>
```

3.10.2 Async API

```
import asyncio
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")


async def main():
    await select_ai.async_connect(user=admin_user, password=password, dsn=dsn)
    await select_ai.async_enable_provider(
        users=select_ai_user, provider_endpoint="*.openai.azure.com"
    )
    print("Enabled AI provider for user: ", select_ai_user)


asyncio.run(main())
```

output:

```
Enabled AI provider for user: <select_ai_db_user>
```

3.11 Disable AI service provider

3.11.1 Sync API

```
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")

select_ai.connect(user=admin_user, password=password, dsn=dsn)
select_ai.disable_provider(
    users=select_ai_user, provider_endpoint="*.openai.azure.com"
)
print("Disabled AI provider for user: ", select_ai_user)
```

output:

```
Disabled AI provider for user:  <select_ai_db_user>
```

3.11.2 Async API

```
import asyncio
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")


async def main():
    await select_ai.async_connect(user=admin_user, password=password, dsn=dsn)
    await select_ai.async_disable_provider(
        users=select_ai_user, provider_endpoint="*.openai.azure.com"
    )
    print("Disabled AI provider for user: ", select_ai_user)


asyncio.run(main())
```

output:

```
Disabled AI provider for user:  <select_ai_db_user>
```

CHAPTER

FOUR

CREDENTIAL

Credential object securely stores API key from your AI provider for use by Oracle Database. The following table shows AI Provider and corresponding credential object format

Table 1: AI Provider and expected credential format

AI Provider	Credential format
Anthropic	`{"username": "anthropic", "password": "sk-xxx"}`
HuggingFace	`{"username": "hf", "password": "hf_xxx"}`
OCI Gen AI	`{"user_ocid": "", "tenancy_ocid": "", "private_key": "", "fingerprint": ""}`
OpenAI	`{"username": "openai", "password": "sk-xxx"}`

4.1 Create credential

In this example, we create a credential object to authenticate to OCI Gen AI service provider:

4.1.1 Sync API

```
import os

import oci
import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)

# Default config file and profile
default_config = oci.config.from_file()
oci.config.validate_config(default_config)
with open(default_config["key_file"]) as fp:
    key_contents = fp.read()
credential = {
    "credential_name": "my_oci_ai_profile_key",
    "user_ocid": default_config["user"],
    "tenancy_ocid": default_config["tenancy"],
    "private_key": key_contents,
    "fingerprint": default_config["fingerprint"],
}
select_ai.create_credential(credential=credential, replace=True)
print("Created credential: ", credential["credential_name"])
```

output:

```
Created credential:  my_oci_ai_profile_key
```

4.1.2 Async API

```
import asyncio
import os

import oci
import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    default_config = oci.config.from_file()
    oci.config.validate_config(default_config)
    with open(default_config["key_file"]) as fp:
        key_contents = fp.read()
    credential = {
        "credential_name": "my_oci_ai_profile_key",
        "user_ocid": default_config["user"],
        "tenancy_ocid": default_config["tenancy"],
        "private_key": key_contents,
        "fingerprint": default_config["fingerprint"],
    }
    await select_ai.async_create_credential(
        credential=credential, replace=True
    )
    print("Created credential: ", credential["credential_name"])


asyncio.run(main())
```

output:

```
Created credential:  my_oci_ai_profile_key
```

CHAPTER

FIVE

PROFILE ATTRIBUTES

5.1 ProfileAttributes

This class defines attributes to manage and configure the behavior of the AI profile. The `ProfileAttributes` objects are created by `select_ai.ProfileAttributes()`.

class select_ai.**ProfileAttributes**(*annotations: str | None = None*, *case_sensitive_values: bool | None = None*, *comments: bool | None = None*, *constraints: str | None = None*, *conversation: bool | None = None*, *credential_name: str | None = None*, *enable_sources: bool | None = None*, *enable_source_offsets: bool | None = None*, *enforce_object_list: bool | None = None*, *max_tokens: int | None = 1024*, *object_list: List[Mapping] | None = None*, *object_list_mode: str | None = None*, *provider:* Provider *| None = None*, *seed: str | None = None*, *stop_tokens: str | None = None*, *streaming: str | None = None*, *temperature: float | None = None*, *vector_index_name: str | None = None*)

Use this class to define attributes to manage and configure the behavior of an AI profile

Parameters

- **comments** (*bool*) – True to include column comments in the metadata used for generating SQL queries from natural language prompts.
- **constraints** (*bool*) – True to include referential integrity constraints such as primary and foreign keys in the metadata sent to the LLM.
- **conversation** (*bool*) – Indicates if conversation history is enabled for a profile.
- **credential_name** (*str*) – The name of the credential to access the AI provider APIs.
- **enforce_object_list** (*bool*) – Specifies whether to restrict the LLM to generate SQL that uses only tables covered by the object list.
- **max_tokens** (*int*) – Denotes the number of tokens to return per generation. Default is 1024.
- **object_list** (*List[Mapping]*) – Array of JSON objects specifying the owner and object names that are eligible for natural language translation to SQL.
- **object_list_mode** (*str*) – Specifies whether to send metadata for the most relevant tables or all tables to the LLM. Supported values are - 'automated' and 'all'
- **provider** (select_ai.Provider) – AI Provider
- **stop_tokens** (*str*) – The generated text will be terminated at the beginning of the earliest stop sequence. Sequence will be incorporated into the text. The attribute value must be a valid array of string values in JSON format

- **temperature** (*float*) – Temperature is a non-negative float number used to tune the degree of randomness. Lower temperatures mean less random generations.
- **vector_index_name** (*str*) – Name of the vector index

CHAPTER

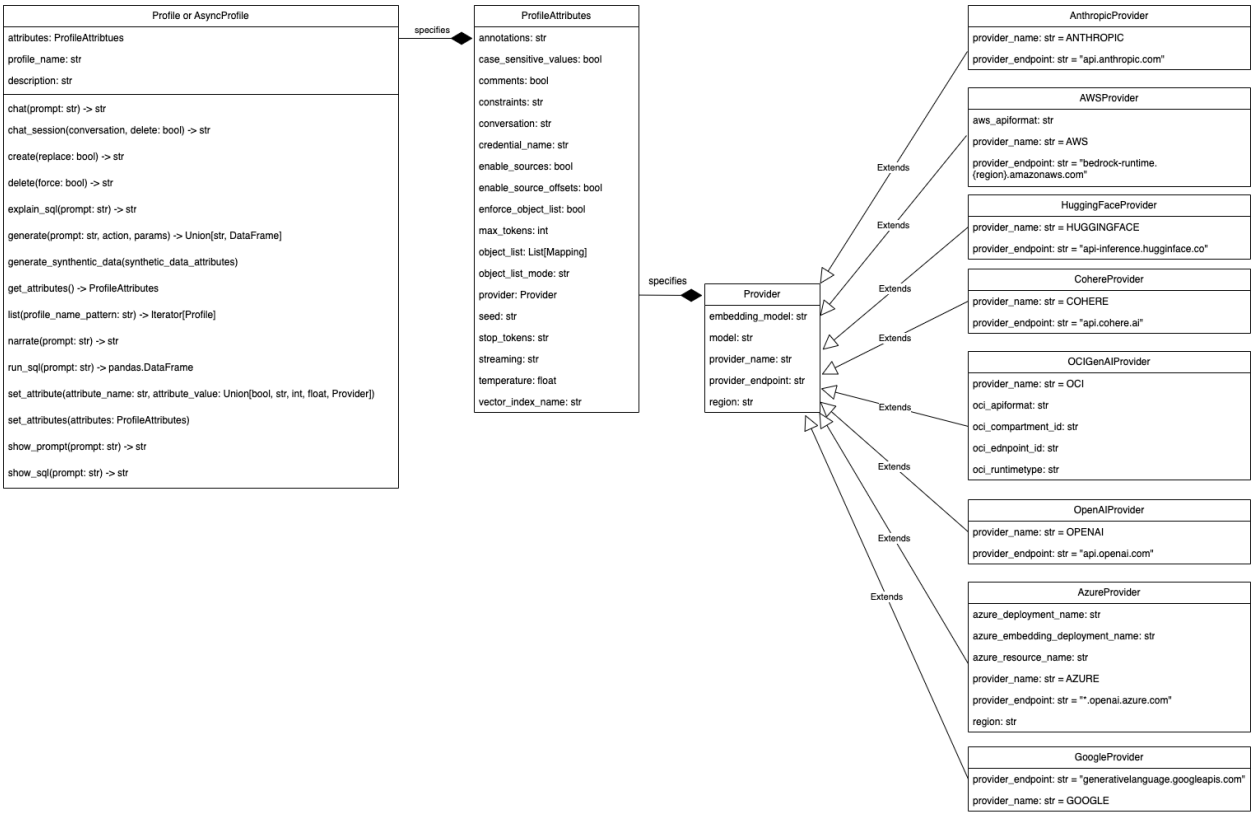
SIX

PROFILE

An AI profile is a specification that includes the AI provider to use and other details regarding metadata and database objects required for generating responses to natural language prompts.

An AI profile object can be created using `select_ai.Profile()`

6.1 Profile Object Model



6.2 Base `Profile` API

class select_ai.**BaseProfile**(*profile_name: str | None = None*, *attributes: ProfileAttributes | None = None*, *description: str | None = None*, *merge: bool | None = False*, *replace: bool | None = False*, *raise_error_if_exists: bool | None = True*)

BaseProfile is an abstract base class representing a Profile for Select AI's interactions with AI service providers (LLMs). Use either select_ai.Profile or select_ai.AsyncProfile to instantiate an AI profile object.

:param str profile_name : Name of the profile

Parameters

- **attributes** (*select_ai.ProfileAttributes*) – Object specifying AI profile attributes
- **description** (*str*) – Description of the profile
- **merge** (*bool*) – Fetches the profile from database, merges the non-null attributes and saves it back in the database. Default value is False
- **replace** (*bool*) – Replaces the profile and attributes in the database. Default value is False
- **raise_error_if_exists** (*bool*) – Raise ProfileExistsError if profile exists in the database and replace = False and merge = False. Default value is True

6.3 Profile API

class select_ai.**Profile**(**args*, ***kwargs*)

Profile class represents an AI Profile. It defines attributes and methods to interact with the underlying AI Provider. All methods in this class are synchronous or blocking

chat(*prompt: str*, *params: Mapping = None*) → str

Chat with the LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

Returns
str

chat_session(*conversation:* Conversation, *delete: bool = False*)

Starts a new chat session for context-aware conversations

Parameters

- **conversation** (Conversation) – Conversation object to use for this chat session
- **delete** (*bool*) – Delete conversation after session ends

Returns

create(*replace: int | None = False*) → None

Create an AI Profile in the Database

Parameters
replace (*bool*) – Set True to replace else False

Returns
None

Raises
oracledb.DatabaseError

delete(*force=False*) → None

Deletes an AI profile from the database

Parameters
force (*bool*) – Ignores errors if AI profile does not exist.

Returns
None

Raises
oracledb.DatabaseError

explain_sql(*prompt: str*, *params: Mapping = None*) → str

Explain the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

Returns
str

generate(*prompt: str*, *action: Action | None = Action.RUNSQL*, *params: Mapping = None*) → DataFrame | str | None

Perform AI translation using this profile

Parameters

- **prompt** (*str*) – Natural language prompt to translate
- **action** (*select_ai.profile.Action*)
- **params** – Parameters to include in the LLM request. For e.g. conversation_id for context-aware chats

Returns
Union[pandas.DataFrame, str]

generate_synthetic_data(*synthetic_data_attributes:* SyntheticDataAttributes) → None

Generate synthetic data for a single table, multiple tables or a full schema.

Parameters
synthetic_data_attributes (select_ai.SyntheticDataAttributes)

Returns
None

Raises
oracledb.DatabaseError

get_attributes() → *ProfileAttributes*

Get AI profile attributes from the Database

Returns
select_ai.ProfileAttributes

classmethod list(*profile_name_pattern: str = '.*'*) → Iterator[*Profile*]

List AI Profiles saved in the database.

Parameters
profile_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all AI profiles.

Returns
Iterator[Profile]

narrate(*prompt: str*, *params: Mapping = None*) → str

Narrate the result of the SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

Returns
str

run_sql(*prompt: str*, *params: Mapping = None*) → DataFrame

Run the generate SQL statement and return a pandas Dataframe built using the result set

Parameters

- **prompt** (*str*) – Natural language prompt

- **params** – Parameters to include in the LLM request

Returns
pandas.DataFrame

set_attribute(*attribute_name: str*, *attribute_value: bool | str | int | float | Provider*)

Updates AI profile attribute on the Python object and also saves it in the database

Parameters

- **attribute_name** (*str*) – Name of the AI profile attribute
- **attribute_value** (*Union[bool, str, int, float, Provider]*) – Value of the profile attribute

Returns
None

set_attributes(*attributes: ProfileAttributes*)

Updates AI profile attributes on the Python object and also saves it in the database

Parameters
attributes (*ProviderAttributes*) – Object specifying AI profile attributes

Returns
None

show_prompt(*prompt: str*, *params: Mapping = None*) → str

Show the prompt sent to LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

Returns
str

show_sql(*prompt: str*, *params: Mapping = None*) → str

Show the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

Returns
str

6.4 Create Profile

```
import os
from pprint import pformat

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
provider = select_ai.OCIGenAIProvider(
    region="us-chicago-1", oci_apiformat="GENERIC"
)
profile_attributes = select_ai.ProfileAttributes(
    credential_name="my_oci_ai_profile_key",
    object_list=[{"owner": "SH"}],
    provider=provider,
)
profile = select_ai.Profile(
    profile_name="oci_ai_profile",
    attributes=profile_attributes,
    description="MY OCI AI Profile",
    replace=True,
)
print("Created profile ", profile.profile_name)
profile_attributes = profile.get_attributes()
print(
    "Profile attributes are: ",
    pformat(profile_attributes.dict(exclude_null=False)),
)
```

output:

```
Created profile  oci_ai_profile
Profile attributes are:  {'annotations': None,
 'case_sensitive_values': None,
 'comments': None,
 'constraints': None,
 'conversation': None,
 'credential_name': 'my_oci_ai_profile_key',
 'enable_source_offsets': None,
 'enable_sources': None,
 'enforce_object_list': None,
 'max_tokens': '1024',
 'object_list': '[{"owner":"SH"}]',
 'object_list_mode': None,
 'provider': OCIGenAIProvider(embedding_model=None,
                              model=None,
                              provider_name='oci',
                              provider_endpoint=None,
                              region='us-chicago-1',
```

(continues on next page)

(continued from previous page)

```
                              oci_apiformat='GENERIC',
                              oci_compartment_id=None,
                              oci_endpoint_id=None,
                              oci_runtimetype=None),
 'seed': None,
 'stop_tokens': None,
 'streaming': None,
 'temperature': None,
 'vector_index_name': None}
```

6.5 Narrate

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(
    profile_name="oci_ai_profile",
)
narration = profile.narrate(prompt="How many promotions?")
print(narration)
```

output:

```
There are 503 promotions in the database.
```

6.6 Show SQL

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
sql = profile.show_sql(prompt="How many promotions ?")
print(sql)
```

output:

```
SELECT
COUNT("p"."PROMO_ID") AS "Number of Promotions"
FROM "SH"."PROMOTIONS" "p"
```

6.7 Run SQL

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
df = profile.run_sql(prompt="How many promotions ?")
print(df.columns)
print(df)
```

output:

```
Index(['Number of Promotions'], dtype='object')
    Number of Promotions
0                    503
```

6.8 Chat

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
response = profile.chat(prompt="What is OCI ?")
print(response)
```

output:

```
OCI stands for Oracle Cloud Infrastructure. It is a comprehensive cloud computing␣
↪platform provided by Oracle Corporation that offers a wide range of services for␣
↪computing, storage, networking, database, and more.
...
...
OCI competes with other major cloud providers, including Amazon Web Services (AWS),␣
↪Microsoft Azure, Google Cloud Platform (GCP), and IBM Cloud.
```

6.9 List profiles

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile()

# matches all the profiles
for fetched_profile in profile.list():
    print(fetched_profile.profile_name)
```

output:

```
ASYNC_OCI_AI_PROFILE
OCI_VECTOR_AI_PROFILE
ASYNC_OCI_VECTOR_AI_PROFILE
OCI_AI_PROFILE
```

6.10 Async Profile

An AsyncProfile object can be created with `select_ai.AsyncProfile()` `AsyncProfile` support use of concurrent programming with asyncio. Unless explicitly noted as synchronous, the `AsyncProfile` methods should be used with `await`.

6.10.1 `AsyncProfile` API

class select_ai.**AsyncProfile**(**args*, ***kwargs*)

AsyncProfile defines methods to interact with the underlying AI Provider asynchronously.

async chat(*prompt*, *params: Mapping = None*) → str

Asynchronously chat with the LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

Returns
str

chat_session(*conversation:* AsyncConversation, *delete: bool = False*)

Starts a new chat session for context-aware conversations

Parameters

- **conversation** (AsyncConversation) – Conversation object to use for this chat session
- **delete** (*bool*) – Delete conversation after session ends

async create(*replace: int | None = False*, *description: str | None = None*) → None

Asynchronously create an AI Profile in the Database

Parameters

- **replace** (*bool*) – Set True to replace else False
- **description** – The profile description

Returns
None

Raises
oracledb.DatabaseError

async delete(*force=False*) → None

Asynchronously deletes an AI profile from the database

Parameters
force (*bool*) – Ignores errors if AI profile does not exist.

Returns
None

Raises
oracledb.DatabaseError

async explain_sql(*prompt: str*, *params: Mapping = None*)

Explain the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

Returns
str

async generate(*prompt: str*, *action=Action.SHOWSQL*, *params: Mapping = None*) → DataFrame | str | None

Asynchronously perform AI translation using this profile

Parameters

- **prompt** (*str*) – Natural language prompt to translate
- **action** (*select_ai.profile.Action*)
- **params** – Parameters to include in the LLM request. For e.g. conversation_id for context-aware chats

Returns
Union[pandas.DataFrame, str]

async generate_synthetic_data(*synthetic_data_attributes:* SyntheticDataAttributes) → None

Generate synthetic data for a single table, multiple tables or a full schema.

Parameters
synthetic_data_attributes (select_ai.SyntheticDataAttributes)

Returns
None

Raises
oracledb.DatabaseError

async get_attributes() → *ProfileAttributes*

Asynchronously gets AI profile attributes from the Database

Returns
select_ai.provider.ProviderAttributes

Raises
ProfileNotFoundError

classmethod list(*profile_name_pattern: str = '.*'*) → AsyncGenerator[*AsyncProfile*, None]

Asynchronously list AI Profiles saved in the database.

Parameters
profile_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all AI profiles.

Returns
Iterator[Profile]

async narrate(*prompt*, *params: Mapping = None*) → str

Narrate the result of the SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

Returns
str

async run_pipeline(*prompt_specifications: List[Tuple[str, Action]]*, *continue_on_error: bool = False*) → List[str | DataFrame]

Send Multiple prompts in a single roundtrip to the Database

Parameters

- **prompt_specifications** (*List[Tuple[str, Action]]*) – List of 2-element tuples. First element is the prompt and second is the corresponding action
- **continue_on_error** (*bool*) – True to continue on error else False

Returns
List[Union[str, pandas.DataFrame]]

async run_sql(*prompt*, *params: Mapping = None*) → DataFrame

Explain the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

Returns
pandas.DataFrame

async set_attribute(*attribute_name: str*, *attribute_value: bool | str | int | float | Provider*)

Updates AI profile attribute on the Python object and also saves it in the database

Parameters

- **attribute_name** (*str*) – Name of the AI profile attribute
- **attribute_value** (*Union[bool, str, int, float]*) – Value of the profile attribute

Returns
None

async set_attributes(*attributes:* ProfileAttributes)

Updates AI profile attributes on the Python object and also saves it in the database

Parameters
attributes (ProfileAttributes) – Object specifying AI profile attributes

Returns
None

async show_prompt(*prompt: str*, *params: Mapping = None*)

Show the prompt sent to LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

Returns
str

async **show_sql**(*prompt*, *params: Mapping = None*)

Show the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

Returns

str

6.10.2 Async Profile creation

```
import asyncio
import os
from pprint import pformat

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


# This example shows how to asynchronously generate SQLs nad run SQLs
async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    provider = select_ai.OCIGenAIProvider(
        region="us-chicago-1", oci_apiformat="GENERIC"
    )
    profile_attributes = select_ai.ProfileAttributes(
        credential_name="my_oci_ai_profile_key",
        object_list=[{"owner": "SH"}],
        provider=provider,
    )
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
        attributes=profile_attributes,
        description="MY OCI AI Profile",
        replace=True,
    )
    print("Created async profile ", async_profile.profile_name)
    profile_attributes = await async_profile.get_attributes()
    print(
        "Profile attributes: ",
        pformat(profile_attributes.dict(exclude_null=False)),
    )


asyncio.run(main())
```

output:

```
Created async profile  async_oci_ai_profile
Profile attributes:  {'annotations': None,
 'case_sensitive_values': None,
 'comments': None,
 'constraints': None,
 'conversation': None,
 'credential_name': 'my_oci_ai_profile_key',
 'enable_source_offsets': None,
 'enable_sources': None,
 'enforce_object_list': None,
 'max_tokens': '1024',
 'object_list': '[{"owner":"SH"}]',
```

(continues on next page)

(continued from previous page)

```
'object_list_mode': None,
'provider': OCIGenAIProvider(embedding_model=None,
                             model=None,
                             provider_name='oci',
                             provider_endpoint=None,
                             region='us-chicago-1',
                             oci_apiformat='GENERIC',
                             oci_compartment_id=None,
                             oci_endpoint_id=None,
                             oci_runtimetype=None),
'seed': None,
'stop_tokens': None,
'streaming': None,
'temperature': None,
'vector_index_name': None}
```

6.10.3 Async explain SQL

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    response = await async_profile.explain_sql("How many promotions ?")
    print(response)


asyncio.run(main())
```

output:

```
To answer the question "How many promotions", we need to write a SQL query that counts␣
↪the number of rows in the "PROMOTIONS" table. Here is the query:

```sql
SELECT
  COUNT("p"."PROMO_ID") AS "Number of Promotions"
FROM
  "SH"."PROMOTIONS" "p";
```

Explanation:

* We use the `COUNT` function to count the number of rows in the table.
* We use the table alias `"p"` to refer to the `"PROMOTIONS"` table.
* We enclose the table name and column name in double quotes to make them case-sensitive.
* We use the `AS` keyword to give an alias to the count column, making it easier to read.

This query will return the total number of promotions in the `"PROMOTIONS"` table.
```

6.10.4 Async run SQL

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


# This example shows how to asynchronously run sql
async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    # run_sql returns a pandas df
    df = await async_profile.run_sql("How many promotions?")
    print(df)


asyncio.run(main())
```

output:

```
   PROMOTION_COUNT
0              503
```

6.10.5 Async show SQL

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    response = await async_profile.show_sql("How many promotions?")
    print(response)


asyncio.run(main())
```

output:

```
SELECT COUNT("p"."PROMO_ID") AS "PROMOTION_COUNT" FROM "SH"."PROMOTIONS" "p"
```

6.10.6 Async concurrent SQL

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    sql_tasks = [
        async_profile.show_sql(prompt="How many customers?"),
        async_profile.run_sql(prompt="How many promotions?"),
        async_profile.explain_sql(prompt="How many promotions?"),
    ]

    # Collect results from multiple asynchronous tasks
    for sql_task in asyncio.as_completed(sql_tasks):
        result = await sql_task
        print(result)


asyncio.run(main())
```

output:

```
SELECT COUNT("c"."CUST_ID") AS "customer_count" FROM "SH"."CUSTOMERS" "c"

To answer the question "How many promotions", we need to write a SQL query that counts␣
↪the number of rows in the "PROMOTIONS" table. Here is the query:

```sql
SELECT
  COUNT("p"."PROMO_ID") AS "number_of_promotions"
FROM
  "SH"."PROMOTIONS" "p";
```

Explanation:

* We use the `COUNT` function to count the number of rows in the table.
* We use the table alias `"p"` to refer to the `"PROMOTIONS"` table.
* We specify the schema name `"SH"` to ensure that we are accessing the correct table.
* We enclose the table name, schema name, and column name in double quotes to make them␣
↪case-sensitive.
* The `AS` keyword is used to give an alias to the calculated column, in this case, `
↪"number_of_promotions"`.
```

(continues on next page)

(continued from previous page)

```
This query will return the total number of promotions in the `"PROMOTIONS"` table.

   PROMOTION_COUNT
0              503
```

6.10.7 Async chat

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile"
    )

    # Asynchronously send multiple chat prompts
    chat_tasks = [
        async_profile.chat(prompt="What is OCI ?"),
        async_profile.chat(prompt="What is OML4PY?"),
        async_profile.chat(prompt="What is Autonomous Database ?"),
    ]
    for chat_task in asyncio.as_completed(chat_tasks):
        result = await chat_task
        print(result)


asyncio.run(main())
```

output:

```
OCI stands for several things depending on the context:

1. **Oracle Cloud Infrastructure (OCI)**: This is a cloud computing service offered by␣
↪Oracle Corporation. It provides a range of services including computing, storage,␣
↪networking, database, and more, allowing businesses to build, deploy, and manage␣
↪applications and services in a secure and scalable manner.

...
..
OML4PY provides a Python interface to OML, allowing users to create, manipulate, and␣
↪analyze models using Python scripts. It enables users to leverage the power of OML and␣
↪OMF from within Python, making it easier to integrate modeling and simulation into␣
↪larger workflows and applications.
...
...

An Autonomous Database is a type of database that uses artificial intelligence (AI) and␣
↪machine learning (ML) to automate many of the tasks typically performed by a database␣
↪administrator (DBA)
...
...
```

6.10.8 Async pipeline

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile"
    )
    prompt_specifications = [
        ("What is Oracle Autonomous Database?", select_ai.Action.CHAT),
        ("Generate SQL to list all customers?", select_ai.Action.SHOWSQL),
        (
            "Explain the query: SELECT * FROM sh.products",
            select_ai.Action.EXPLAINSQL,
        ),
        ("Explain the query: SELECT * FROM sh.products", "INVALID ACTION"),
    ]

    # 1. Multiple prompts are sent in a single roundtrip to the Database
    # 2. Results are returned as soon as Database has executed all prompts
    # 3. Application doesn't have to wait on one response before sending
    #    the next prompts
    # 4. Fewer round trips and database is kept busy
    # 5. Efficient network usage
    results = await async_profile.run_pipeline(
        prompt_specifications, continue_on_error=True
    )
    for i, result in enumerate(results):
        print(
            f"Result {i} for prompt '{prompt_specifications[i][0]}' is: {result}"
        )


asyncio.run(main())
```

output:

```
Result 0 for prompt 'What is Oracle Autonomous Database?' is: Oracle Autonomous Database␣
↪is a cloud-based database service that uses artificial intelligence (AI) and machine␣
↪learning (ML) to automate many of the tasks associated with managing a database. It is␣
↪a self-driving, self-securing, and self-repairing database that eliminates the need␣
↪for manual database administration, allowing users to focus on higher-level tasks.


Result 1 for prompt 'Generate SQL to list all customers?' is: SELECT "c"."CUST_ID" AS
```

(continues on next page)

(continued from previous page)

```
↪"Customer ID", "c"."CUST_FIRST_NAME" AS "First Name", "c"."CUST_LAST_NAME" AS "Last␣
↪Name", "c"."CUST_EMAIL" AS "Email" FROM "SH"."CUSTOMERS" "c"

Result 2 for prompt 'Explain the query: SELECT * FROM sh.products' is: ```sql
SELECT
  p.*
FROM
  "SH"."PRODUCTS" p;
```

Explanation:

This query is designed to retrieve all columns (`*`) from the `"SH"."PRODUCTS"` table.

Here's a breakdown of the query components:

Result 3 for prompt 'Explain the query: SELECT * FROM sh.products' is: ORA-20000:␣
↪Invalid action - INVALID ACTION
```
```

6.10.9 List profiles asynchronously

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile()
    # matches the start of string
    async for fetched_profile in async_profile.list(
        profile_name_pattern="^oci"
    ):
        p = await fetched_profile
        print(p.profile_name)


asyncio.run(main())
```

output:

```
OCI_VECTOR_AI_PROFILE
OCI_AI_PROFILE
```

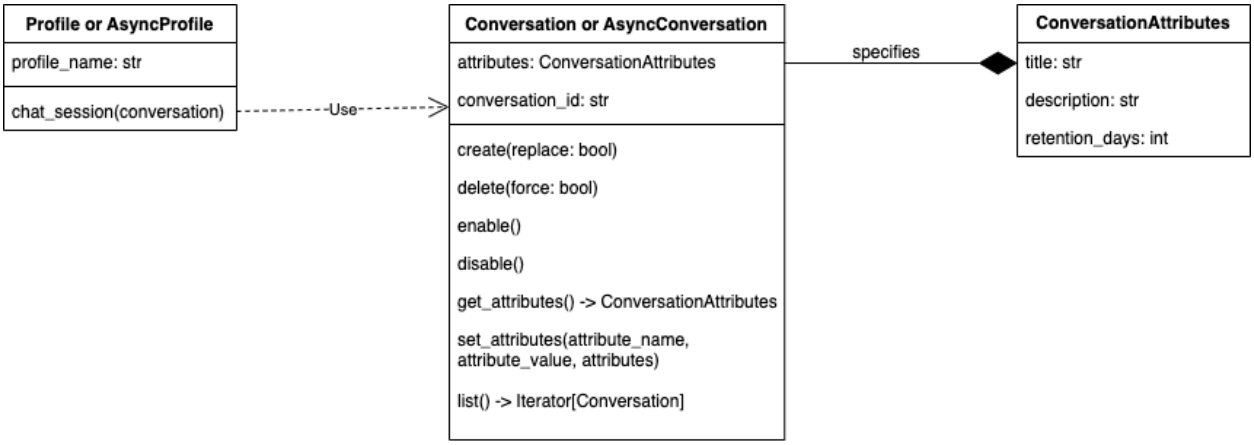
CHAPTER

SEVEN

CONVERSATION

Conversations in Select AI represent an interactive exchange between the user and the system, enabling users to query or interact with the database through a series of natural language prompts.

7.1 Conversation Object model



7.2 ConversationAttributes

class select_ai.**ConversationAttributes**(*title: str | None = 'New Conversation'*, *description: str | None = None*, *retention_days: timedelta | None = datetime.timedelta(days=7)*, *conversation_length: int | None = 10*)

Conversation Attributes

Parameters

- **title** (*str*) – Conversation Title
- **description** (*str*) – Description of the conversation topic
- **retention_days** (*datetime.timedelta*) – The number of days the conversation will be stored in the database from its creation date. If value is 0, the conversation will not be removed unless it is manually deleted by delete
- **conversation_length** (*int*) – Number of prompts to store for this conversation

7.3 Conversation API

class select_ai.**Conversation**(*conversation_id: str | None = None*, *attributes:* ConversationAttributes *| None = None*)

Conversation class can be used to create, update and delete conversations in the database

Typical usage is to combine this conversation object with an AI Profile.chat_session() to have context-aware conversations with the LLM provider

Parameters

- **conversation_id** (*str*) – Conversation ID
- **attributes** (ConversationAttributes) – Conversation attributes

create() → str

Creates a new conversation and returns the conversation_id to be used in context-aware conversations with LLMs

Returns
conversation_id

delete(*force: bool = False*)

Drops the conversation

get_attributes() → *ConversationAttributes*

Get attributes of the conversation from the database

classmethod list() → Iterator[*Conversation*]

List all conversations

Returns
Iterator[VectorIndex]

set_attributes(*attributes:* ConversationAttributes)

Updates the attributes of the conversation in the database

7.3.1 Create conversion

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
conversation_attributes = select_ai.ConversationAttributes(
    title="History of Science",
    description="LLM's understanding of history of science",
)
conversation = select_ai.Conversation(attributes=conversation_attributes)
conversation_id = conversation.create()

print("Created conversation with conversation id: ", conversation_id)
```

output:

```
Created conversation with conversation id:  3AB2ED3E-7E52-8000-E063-BE1A000A15B6
```

7.3.2 Chat session

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
conversation_attributes = select_ai.ConversationAttributes(
    title="History of Science",
    description="LLM's understanding of history of science",
)
conversation = select_ai.Conversation(attributes=conversation_attributes)
with profile.chat_session(conversation=conversation, delete=True) as session:
    print(
        "Conversation ID for this session is:",
        conversation.conversation_id,
    )
    response = session.chat(
        prompt="What is importance of history of science ?"
    )
    print(response)
    response = session.chat(
        prompt="Elaborate more on 'Learning from past mistakes'"
    )
    print(response)
```

output:

```
Conversation ID for this session is: 380A1910-5BF2-F7A1-E063-D81A000A3FDA

The importance of the history of science lies in its ability to provide a comprehensive␣
↪understanding of the development of scientific knowledge and its impact on society.␣
↪Here are some key reasons why the history of science is important:

1. **Contextualizing Scientific Discoveries**: The history of science helps us␣
↪understand the context in which scientific discoveries were made, including the social,
↪ cultural, and intellectual climate of the time. This context is essential for␣
↪appreciating the significance and relevance of scientific findings.

..
..

The history of science is replete with examples of mistakes, errors, and misconceptions␣
↪that have occurred over time. By studying these mistakes, scientists and researchers␣
↪can gain valuable insights into the pitfalls and challenges that have shaped the␣
↪development of scientific knowledge. Learning from past mistakes is essential for␣
↪several reasons:
...
...
```

7.3.3 List conversations

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
for conversation in select_ai.Conversation().list():
    print(conversation.conversation_id)
    print(conversation.attributes)
```

output:

```
5275A80-A290-DA17-E063-151B000AD3B4
ConversationAttributes(title='History of Science', description="LLM's understanding of␣
↪history of science", retention_days=7)

37DF777F-F3DA-F084-E063-D81A000A53BE
ConversationAttributes(title='History of Science', description="LLM's understanding of␣
↪history of science", retention_days=7)
```

7.3.4 Delete conversation

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
conversation = select_ai.Conversation(
    conversation_id="37DDC22E-11C8-3D49-E063-D81A000A85FE"
)
conversation.delete(force=True)
print(
    "Deleted conversation with conversation id: ",
    conversation.conversation_id,
)
```

output:

```
Deleted conversation with conversation id:  37DDC22E-11C8-3D49-E063-D81A000A85FE
```

7.4 AsyncConversation API

class select_ai.**AsyncConversation**(*conversation_id: str | None = None*, *attributes:* ConversationAttributes | *None = None*)

AsyncConversation class can be used to create, update and delete conversations in the database in an async manner

Typical usage is to combine this conversation object with an AsyncProfile.chat_session() to have context-aware conversations

Parameters

- **conversation_id** (*str*) – Conversation ID
- **attributes** (ConversationAttributes) – Conversation attributes

async create() → str

Creates a new conversation and returns the conversation_id to be used in context-aware conversations with LLMs

Returns
conversation_id

async delete(*force: bool = False*)

Delete the conversation

async get_attributes() → *ConversationAttributes*

Get attributes of the conversation from the database

classmethod list() → AsyncGenerator[*AsyncConversation*, None]

List all conversations

Returns
Iterator[VectorIndex]

async set_attributes(*attributes:* ConversationAttributes)

Updates the attributes of the conversation

7.4.1 Async chat session

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile"
    )
    conversation_attributes = select_ai.ConversationAttributes(
        title="History of Science",
        description="LLM's understanding of history of science",
    )
    async_conversation = select_ai.AsyncConversation(
        attributes=conversation_attributes
    )

    async with async_profile.chat_session(
        conversation=async_conversation, delete=True
    ) as async_session:
        response = await async_session.chat(
            prompt="What is importance of history of science ?"
        )
        print(response)
        response = await async_session.chat(
            prompt="Elaborate more on 'Learning from past mistakes'"
        )
        print(response)


asyncio.run(main())
```

output:

```
Conversation ID for this session is: 380A1910-5BF2-F7A1-E063-D81A000A3FDA

The importance of the history of science lies in its ability to provide a comprehensive␣
↪understanding of the development of scientific knowledge and its impact on society.␣
↪Here are some key reasons why the history of science is important:

1. **Contextualizing Scientific Discoveries**: The history of science helps us␣
↪understand the context in which scientific discoveries were made, including the social,
↪ cultural, and intellectual climate of the time. This context is essential for␣
↪appreciating the significance and relevance of scientific findings.

..
```

(continues on next page)

(continued from previous page)

```
..

The history of science is replete with examples of mistakes, errors, and misconceptions␣
↪that have occurred over time. By studying these mistakes, scientists and researchers␣
↪can gain valuable insights into the pitfalls and challenges that have shaped the␣
↪development of scientific knowledge. Learning from past mistakes is essential for␣
↪several reasons:
...
...
```

7.4.2 Async list conversations

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async for conversation in select_ai.AsyncConversation().list():
        print(conversation.conversation_id)
        print(conversation.attributes)


asyncio.run(main())
```

output:

```
5275A80-A290-DA17-E063-151B000AD3B4
ConversationAttributes(title='History of Science', description="LLM's understanding of␣
↪history of science", retention_days=7)

37DF777F-F3DA-F084-E063-D81A000A53BE
ConversationAttributes(title='History of Science', description="LLM's understanding of␣
↪history of science", retention_days=7)
```

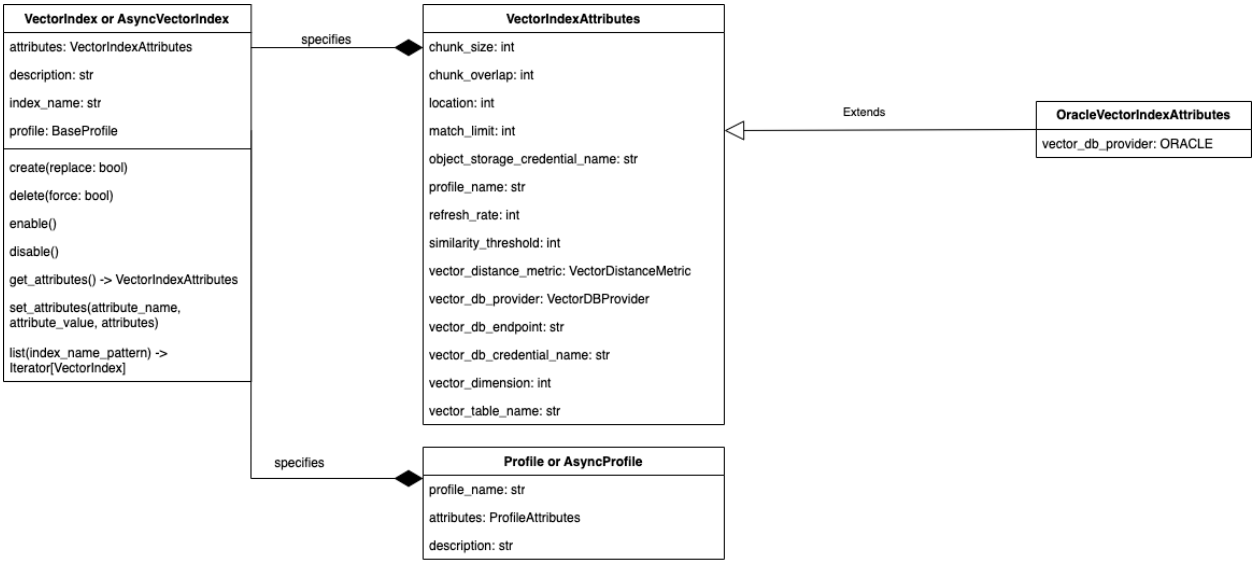
CHAPTER

EIGHT

VECTOR INDEX

`VectorIndex` supports Retrieval Augmented Generation (RAG). For e.g., you can convert text into vector embeddings and store them in a vector store. Select AI will augment the natural language prompt by retrieving content from the vector store using semantic similarity search.

8.1 `VectorIndex` Object Model



8.2 VectorIndexAttributes

A `VectorIndexAttributes` object can be created with `select_ai.VectorIndexAttributes()`. Also check vector index attributes

class select_ai.**VectorIndexAttributes**(*chunk_size: int | None = 1024*, *chunk_overlap: int | None = 128*, *location: str | None = None*, *match_limit: int | None = 5*, *object_storage_credential_name: str | None = None*, *profile_name: str | None = None*, *refresh_rate: int | None = 1440*, *similarity_threshold: float | None = 0*, *vector_distance_metric: VectorDistanceMetric | None = VectorDistanceMetric.COSINE*, *vector_db_endpoint: str | None = None*, *vector_db_credential_name: str | None = None*, *vector_db_provider: VectorDBProvider | None = None*, *vector_dimension: int | None = None*, *vector_table_name: str | None = None*, *pipeline_name: str | None = None*)

Attributes of a vector index help to manage and configure the behavior of the vector index.

Parameters

- **chunk_size** (*int*) – Text size of chunking the input data.
- **chunk_overlap** (*int*) – Specifies the amount of overlapping characters between adjacent chunks of text.
- **location** (*str*) – Location of the object store.
- **match_limit** (*int*) – Specifies the maximum number of results to return in a vector search query
- **object_storage_credential_name** (*str*) – Name of the credentials for accessing object storage.
- **profile_name** (*str*) – Name of the AI profile which is used for embedding source data and user prompts.
- **refresh_rate** (*int*) – Interval of updating data in the vector store. The unit is minutes.
- **similarity_threshold** (*float*) – Defines the minimum level of similarity required for two items to be considered a match
- **vector_distance_metric** (*VectorDistanceMetric*) – Specifies the type of distance calculation used to compare vectors in a database
- **vector_db_provider** (*VectorDBProvider*) – Name of the Vector database provider. Default value is “oracle”
- **vector_db_endpoint** (*str*) – Endpoint to access the Vector database
- **vector_db_credential_name** (*str*) – Name of the credentials for accessing Vector database
- **vector_dimension** (*int*) – Specifies the number of elements in each vector within the vector store
- **vector_table_name** (*str*) – Specifies the name of the table or collection to store vector embeddings and chunked data

8.2.1 OracleVectorIndexAttributes

class select_ai.**OracleVectorIndexAttributes**(*chunk_size: int | None = 1024*, *chunk_overlap: int | None = 128*, *location: str | None = None*, *match_limit: int | None = 5*, *object_storage_credential_name: str | None = None*, *profile_name: str | None = None*, *refresh_rate: int | None = 1440*, *similarity_threshold: float | None = 0*, *vector_distance_metric: VectorDistanceMetric | None = VectorDistanceMetric.COSINE*, *vector_db_endpoint: str | None = None*, *vector_db_credential_name: str | None = None*, *vector_db_provider: VectorDBProvider | None = VectorDBProvider.ORACLE*, *vector_dimension: int | None = None*, *vector_table_name: str | None = None*, *pipeline_name: str | None = None*)

Oracle specific vector index attributes

8.3 VectorIndex API

A VectorIndex object can be created with select_ai.VectorIndex()

class select_ai.**VectorIndex**(*profile:* BaseProfile = *None*, *index_name: str* | *None* = *None*, *description: str* | *None* = *None*, *attributes:* VectorIndexAttributes | *None* = *None*)

VectorIndex objects let you manage vector indexes

Parameters

- **index_name** (*str*) – The name of the vector index
- **description** (*str*) – The description of the vector index
- **attributes** (select_ai.VectorIndexAttributes) – The attributes of the vector index

create(*replace: bool* | *None* = *False*)

Create a vector index in the database and populates the index
with data from an object store bucket using an async scheduler job

Parameters
replace (*bool*) – Replace vector index if it exists

Returns
None

delete(*include_data: bool* | *None* = *True*, *force: bool* | *None* = *False*)

This procedure removes a vector store index

Parameters

- **include_data** (*bool*) – Indicates whether to delete both the customer's vector store and vector index along with the vector index object
- **force** (*bool*) – Indicates whether to ignore errors that occur if the vector index does not exist

Returns
None

Raises
oracledb.DatabaseError

disable()

This procedure disables a vector index object in the current database. When disabled, an AI profile cannot use the vector index, and the system does not load data into the vector store as new data is added to the object store and does not perform indexing, searching or querying based on the index.

Returns
None

Raises
oracledb.DatabaseError

enable()

This procedure enables or activates a previously disabled vector index object. Generally, when you create a vector index, by default it is enabled such that the AI profile can use it to perform indexing and searching.

Returns
None

Raises
oracledb.DatabaseError

`get_attributes()` → *VectorIndexAttributes*

Get attributes of this vector index

Returns
select_ai.VectorIndexAttributes

Raises
VectorIndexNotFoundError

`classmethod list(index_name_pattern: str = '.*')` → Iterator[*VectorIndex*]

List Vector Indexes

Parameters
index_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all vector indexes.

Returns
Iterator[VectorIndex]

`set_attributes(attribute_name: str, attribute_value: str | int | float, attributes: VectorIndexAttributes = None)`

This procedure updates an existing vector store index with a specified value of the vector index attribute. You can specify a single attribute or multiple attributes by passing an object of type :class *VectorIndexAttributes*

Parameters

- **attribute_name** (*str*) – Custom attribute name
- **attribute_value** (*Union[str, int, float]*) – Attribute Value
- **attributes** (`VectorIndexAttributes`) – Specify multiple attributes to update in a single API invocation

Returns
None

Raises
oracledb.DatabaseError

Check the examples below to understand how to create vector indexes

8.3.1 Create vector index

In the following example, vector database provider is Oracle and objects (to create embedding for) reside in OCI's object store

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


select_ai.connect(user=user, password=password, dsn=dsn)
# Configure an AI provider with an embedding model
# of your choice
provider = select_ai.OCIGenAIProvider(
    region="us-chicago-1",
    oci_apiformat="GENERIC",
    embedding_model="cohere.embed-english-v3.0",
)

# Create an AI profile to use the Vector index with
profile_attributes = select_ai.ProfileAttributes(
    credential_name="my_oci_ai_profile_key",
    provider=provider,
)
profile = select_ai.Profile(
    profile_name="oci_vector_ai_profile",
    attributes=profile_attributes,
    description="MY OCI AI Profile",
    replace=True,
)

# Specify objects to create an embedding for. In this example,
# the objects reside in ObjectStore and the vector database is
# Oracle
vector_index_attributes = select_ai.OracleVectorIndexAttributes(
    location="https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/conda-
↪environment/o/tenant1-pdb3/graph",
    object_storage_credential_name="my_oci_ai_profile_key",
)

# Create a Vector index object
vector_index = select_ai.VectorIndex(
    index_name="test_vector_index",
    attributes=vector_index_attributes,
    description="Test vector index",
    profile=profile,
)
vector_index.create(replace=True)
print("Created vector index: test_vector_index")
```

output:

```
Created vector index: test_vector_index
```

8.3.2 List vector index

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
vector_index = select_ai.VectorIndex()
for index in vector_index.list(index_name_pattern="^test"):
    print("Vector index", index.index_name)
    print("Vector index profile", index.profile)
```

output:

```
Vector index TEST_VECTOR_INDEX
Vector index profile Profile(profile_name=oci_vector_ai_profile,␣
↪attributes=ProfileAttributes(annotations=None, case_sensitive_values=None,␣
↪comments=None, constraints=None, conversation=None, credential_name='my_oci_ai_profile_
↪key', enable_sources=None, enable_source_offsets=None, enforce_object_list=None, max_
↪tokens=1024, object_list=None, object_list_mode=None,␣
↪provider=OCIGenAIProvider(embedding_model=None, model=None, provider_name='oci',␣
↪provider_endpoint=None, region='us-chicago-1', oci_apiformat='GENERIC', oci_
↪compartment_id=None, oci_endpoint_id=None, oci_runtimetype=None), seed=None, stop_
↪tokens=None, streaming=None, temperature=None, vector_index_name='test_vector_index'),␣
↪description=None)
```

8.3.3 RAG using vector index

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_vector_ai_profile")
r = profile.narrate("list the conda environments in my object store")
print(r)
```

output:

```
The conda environments in your object store are:
 1. fccenv
 2. myrenv
 3. fully-loaded-mlenv
 4. graphenv

 These environments are listed in the provided data as separate JSON documents, each␣
↪containing information about a specific conda environment.

 Sources:
   - fccenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/
↪b/conda-environment/o/tenant1-pdb3/graph/fccenv-manifest.json)
   - myrenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/
↪b/conda-environment/o/tenant1-pdb3/graph/myrenv-manifest.json)
   - fully-loaded-mlenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.
↪com/n/dwcsdev/b/conda-environment/o/tenant1-pdb3/graph/fully-loaded-mlenv-manifest.
↪json)
   - graphenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/
↪dwcsdev/b/conda-environment/o/tenant1-pdb3/graph/graphenv-manifest.json)
```

8.3.4 Delete vector index

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
vector_index = select_ai.VectorIndex(index_name="test_vector_index")
vector_index.delete(force=True)
print("Deleted vector index: test_vector_index")
```

output:

```
Deleted vector index: test_vector_index
```

8.4 AsyncVectorIndex API

A AsyncVectorIndex object can be created with select_ai.AsyncVectorIndex()

class select_ai.**AsyncVectorIndex**(*profile:* BaseProfile = *None*, *index_name: str* | *None* = *None*, *description: str* | *None* = *None*, *attributes:* VectorIndexAttributes | *None* = *None*)

AsyncVectorIndex objects let you manage vector indexes using async APIs. Use this for non-blocking concurrent requests

Parameters

- **index_name** (*str*) – The name of the vector index
- **description** (*str*) – The description of the vector index
- **attributes** (VectorIndexAttributes) – The attributes of the vector index

async create(*replace: bool* | *None* = *False*) → None

Create a vector index in the database and populates it with data from an object store bucket using an async scheduler job

Parameters

replace (*bool*) – True to replace existing vector index

async delete(*include_data: bool* | *None* = *True*, *force: bool* | *None* = *False*) → None

This procedure removes a vector store index.

Parameters

- **include_data** (*bool*) – Indicates whether to delete both the customer's vector store and vector index along with the vector index object.
- **force** (*bool*) – Indicates whether to ignore errors that occur if the vector index does not exist.

Returns

None

Raises

oracledb.DatabaseError

async disable() → None

This procedure disables a vector index object in the current database. When disabled, an AI profile cannot use the vector index, and the system does not load data into the vector store as new data is added to the object store and does not perform indexing, searching or querying based on the index.

Returns

None

Raises

oracledb.DatabaseError

async enable() → None

This procedure enables or activates a previously disabled vector index object. Generally, when you create a vector index, by default it is enabled such that the AI profile can use it to perform indexing and searching.

Returns

None

Raises

oracledb.DatabaseError

async get_attributes() → *VectorIndexAttributes*

Get attributes of a vector index

Returns
select_ai.VectorIndexAttributes

Raises
VectorIndexNotFoundError

classmethod list(*index_name_pattern: str = '.*'*) → AsyncGenerator[*VectorIndex*, None]

List Vector Indexes.

Parameters
index_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all vector indexes.

Returns
AsyncGenerator[VectorIndex]

async set_attributes(*attribute_name: str*, *attribute_value: str | int*, *attributes:* VectorIndexAttributes = *None*) → None

This procedure updates an existing vector store index with a specified value of the vector index attribute. You can specify a single attribute or multiple attributes by passing an object of type :class *VectorIndexAttributes*

Parameters

- **attribute_name** (*str*) – Custom attribute name
- **attribute_value** (*Union[str, int, float]*) – Attribute Value
- **attributes** (VectorIndexAttributes) – Specify multiple attributes to update in a single API invocation

Returns
None

Raises
oracledb.DatabaseError

8.4.1 Async create vector index

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)

    provider = select_ai.OCIGenAIProvider(
        region="us-chicago-1",
        oci_apiformat="GENERIC",
        embedding_model="cohere.embed-english-v3.0",
    )
    profile_attributes = select_ai.ProfileAttributes(
        credential_name="my_oci_ai_profile_key",
        provider=provider,
    )
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_vector_ai_profile",
        attributes=profile_attributes,
        description="MY OCI AI Profile",
        replace=True,
    )

    vector_index_attributes = select_ai.OracleVectorIndexAttributes(
        location="https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/conda-
↪environment/o/tenant1-pdb3/graph",
        object_storage_credential_name="my_oci_ai_profile_key",
    )

    async_vector_index = select_ai.AsyncVectorIndex(
        index_name="test_vector_index",
        attributes=vector_index_attributes,
        description="Vector index for conda environments",
        profile=async_profile,
    )
    await async_vector_index.create(replace=True)
    print("Created vector index: test_vector_index")


asyncio.run(main())
```

output:

```
created vector index: test_vector_index
```

8.4.2 Async list vector index

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    vector_index = select_ai.AsyncVectorIndex()
    async for index in vector_index.list(index_name_pattern="^test"):
        print("Vector index", index.index_name)
        print("Vector index profile", index.profile)


asyncio.run(main())
```

output:

```
Vector index TEST_VECTOR_INDEX
Vector index profile AsyncProfile(profile_name=oci_vector_ai_profile,␣
↪attributes=ProfileAttributes(annotations=None, case_sensitive_values=None,␣
↪comments=None, constraints=None, conversation=None, credential_name='my_oci_ai_profile_
↪key', enable_sources=None, enable_source_offsets=None, enforce_object_list=None, max_
↪tokens=1024, object_list=None, object_list_mode=None,␣
↪provider=OCIGenAIProvider(embedding_model=None, model=None, provider_name='oci',␣
↪provider_endpoint=None, region='us-chicago-1', oci_apiformat='GENERIC', oci_
↪compartment_id=None, oci_endpoint_id=None, oci_runtimetype=None), seed=None, stop_
↪tokens=None, streaming=None, temperature=None, vector_index_name='test_vector_index'),␣
↪description=None)
```

8.4.3 Async RAG using vector index

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_vector_ai_profile"
    )
    r = await async_profile.narrate(
        "list the conda environments in my object store"
    )
    print(r)


asyncio.run(main())
```

output:

```
The conda environments in your object store are:
1. fccenv
2. myrenv
3. fully-loaded-mlenv
4. graphenv

These environments are listed in the provided data as separate JSON documents, each␣
↪containing information about a specific conda environment.

Sources:
  - fccenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/
↪conda-environment/o/tenant1-pdb3/graph/fccenv-manifest.json)
  - myrenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/
↪conda-environment/o/tenant1-pdb3/graph/myrenv-manifest.json)
  - fully-loaded-mlenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/
↪n/dwcsdev/b/conda-environment/o/tenant1-pdb3/graph/fully-loaded-mlenv-manifest.json)
  - graphenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/
↪b/conda-environment/o/tenant1-pdb3/graph/graphenv-manifest.json)
```

CHAPTER

NINE

SYNTHETIC DATA

9.1 SyntheticDataAttributes

class select_ai.**SyntheticDataAttributes**(*object_name: str | None = None*, *object_list: List[Mapping] | None = None*, *owner_name: str | None = None*, *params: SyntheticDataParams | None = None*, *record_count: int | None = None*, *user_prompt: str | None = None*)

Attributes to control generation of synthetic data

Parameters

- **object_name** (*str*) – Table name to populate synthetic data
- **object_list** (*List[Mapping]*) – Use this to generate synthetic data on multiple tables
- **owner_name** (*str*) – Database user who owns the referenced object. Default value is connected user's schema
- **record_count** (*int*) – Number of records to generate
- **user_prompt** (*str*) – User prompt to guide generation of synthetic data For e.g. "the release date for the movies should be in 2019"

9.2 SyntheticDataParams

class select_ai.SyntheticDataParams(*sample_rows: int | None = None*, *table_statistics: bool | None = False*, *priority: str | None = 'HIGH'*, *comments: bool | None = False*)

Optional parameters to control generation of synthetic data

Parameters

- **sample_rows** (*int*) – number of rows from the table to use as a sample to guide the LLM in data generation
- **table_statistics** (*bool*) – Enable or disable the use of table statistics information. Default value is False
- **priority** (*str*) – Assign a priority value that defines the number of parallel requests sent to the LLM for generating synthetic data. Tasks with a higher priority will consume more database resources and complete faster. Possible values are: HIGH, MEDIUM, LOW
- **comments** (*bool*) – Enable or disable sending comments to the LLM to guide data generation. Default value is False

Also, check the generate_synthetic_data PL/SQL API for attribute details

9.3 Single table synthetic data

The below example shows single table synthetic data generation

9.3.1 Sync API

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
synthetic_data_params = select_ai.SyntheticDataParams(
    sample_rows=100, table_statistics=True, priority="HIGH"
)
synthetic_data_attributes = select_ai.SyntheticDataAttributes(
    object_name="MOVIE",
    user_prompt="the release date for the movies should be in 2019",
    params=synthetic_data_params,
    record_count=100,
)
profile.generate_synthetic_data(
    synthetic_data_attributes=synthetic_data_attributes
)
```

output:

```
SQL> select count(*) from movie;

  COUNT(*)
----------
       100
```

9.3.2 Async API

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    synthetic_data_params = select_ai.SyntheticDataParams(
        sample_rows=100, table_statistics=True, priority="HIGH"
    )
    synthetic_data_attributes = select_ai.SyntheticDataAttributes(
        object_name="MOVIE",
        user_prompt="the release date for the movies should be in 2019",
        params=synthetic_data_params,
        record_count=100,
    )
    await async_profile.generate_synthetic_data(
        synthetic_data_attributes=synthetic_data_attributes
    )


asyncio.run(main())
```

output:

```
SQL> select count(*) from movie;

  COUNT(*)
----------
       100
```

9.4 Multi table synthetic data

The below example shows multitable synthetic data generation

9.4.1 Sync API

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
synthetic_data_params = select_ai.SyntheticDataParams(
    sample_rows=100, table_statistics=True, priority="HIGH"
)
object_list = [
    {
        "owner": user,
        "name": "MOVIE",
        "record_count": 100,
        "user_prompt": "the release date for the movies should be in 2019",
    },
    {"owner": user, "name": "ACTOR", "record_count": 10},
    {"owner": user, "name": "DIRECTOR", "record_count": 5},
]
synthetic_data_attributes = select_ai.SyntheticDataAttributes(
    object_list=object_list, params=synthetic_data_params
)
profile.generate_synthetic_data(
    synthetic_data_attributes=synthetic_data_attributes
)
```

output:

```
SQL> select count(*) from actor;

  COUNT(*)
----------
        40

SQL> select count(*) from director;

  COUNT(*)
----------
        13

SQL> select count(*) from movie;

  COUNT(*)
```

(continues on next page)

(continued from previous page)

```
----------
       300
```

9.4.2 Async API

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    synthetic_data_params = select_ai.SyntheticDataParams(
        sample_rows=100, table_statistics=True, priority="HIGH"
    )
    object_list = [
        {
            "owner": user,
            "name": "MOVIE",
            "record_count": 100,
            "user_prompt": "the release date for the movies should be in 2019",
        },
        {"owner": user, "name": "ACTOR", "record_count": 10},
        {"owner": user, "name": "DIRECTOR", "record_count": 5},
    ]
    synthetic_data_attributes = select_ai.SyntheticDataAttributes(
        object_list=object_list, params=synthetic_data_params
    )
    await async_profile.generate_synthetic_data(
        synthetic_data_attributes=synthetic_data_attributes
    )


asyncio.run(main())
```

output:

```
SQL> select count(*) from actor;

  COUNT(*)
----------
    40

SQL> select count(*) from director;
```

(continues on next page)

(continued from previous page)

```
  COUNT(*)
----------
    13

SQL> select count(*) from movie;

  COUNT(*)
----------
       300
```

INDEX

R

S

V