Async Programming in Python

If you are new to async programming in Python, this page is for you.

In LlamaIndex specifically, many operations and functions support async execution. This allows you to run multiple operations at the same time without blocking the main thread, which helps improve overall throughput and performance in many cases.

Here are some of the key concepts you should understand:

1. The Basics of `asyncio`

- **Event Loop**: The event loop handles the scheduling and execution of async operations. It continuously checks for and executes tasks (coroutines). All async operations run by this loop, and there can only be one event loop per thread.
- **`asyncio.run()`**: This function is the entry point for running an asynchronous program. It creates and manages the event loop and cleans up after it completes. Remember that it is designed to be called once per thread. Some frameworks like FastAPI will run the event loop for you, others will require you to run it yourself.
- **Async + Python Notebooks**: Python notebooks are a special case where the event loop is already running. This means you don't need to call `asyncio.run()` yourself, and can directly call and await async functions.

2. Async Functions and `await`

- **Defining Async Functions**: Use the `async def` syntax to define an asynchronous function (coroutine). Instead of executing immediately, calling an async function returns a coroutine object that needs to be scheduled and run.
- **Using `await`**: Inside an async function, `await` is used to pause execution of that function until the awaited task is complete. When you write `await some_fn()`, the function yields control back to the event loop so that other tasks can be scheduled and run. Only one async function executes at a time, and they cooperate by yielding with `await`.

3. Concurrency Explained

- **Cooperative Concurrency**: Although you can schedule multiple async tasks, only one task runs at a time. This is different from true parallelism, where multiple tasks run at the same time. When a task hits an `await`, it suspends its execution so that another task may run. This makes async programs excellent for I/O-bound tasks where waiting is common, such as API calls to LLMs and other services.
- **Not True Parallelism**: Asyncio enables concurrency but does not run tasks in parallel. For CPU-bound work requiring parallel execution, consider threading or multiprocessing. LlamaIndex typically avoids multiprocessing in most cases, and leaves it up to the user to implement, as it can be complex to do so in a way that is safe and efficient.

4. Handling Blocking (Synchronous) Code

- **`asyncio.to_thread()`**: Sometimes you need to run synchronous (blocking) code without freezing your async program. `asyncio.to_thread()` offloads the blocking code to a separate thread, allowing the event loop to continue processing other tasks. Use it cautiously, as it adds some overhead and can make debugging more challenging.
- **Alternative: Executors**: You might also encounter the use of `loop.run_in_executor()` to handle blocking functions.

5. A Practical Example

Below is an example demonstrating how to write and run async functions with `asyncio`:

```
python
import asyncio


async def fetch_data(delay):
    print(f"Started fetching data with {delay}s delay")

    # Simulates I/O-bound work, such as network operation
    await asyncio.sleep(delay)

    print("Finished fetching data")
    return f"Data after {delay}s"


async def main():
    print("Starting main")

    # Schedule two tasks concurrently
    task1 = asyncio.create_task(fetch_data(2))
    task2 = asyncio.create_task(fetch_data(3))

    # Wait until both tasks complete
```

```python
    result1, result2 = await asyncio.gather(task1, task2)

    print(result1)
    print(result2)
    print("Main complete")


if name == "main":
    asyncio.run(main())
```