باسمه تعالی

با توجه به کدهاب داخل MAS\_system میخوام این سناریو را پیاده سازی کنی.

کدها را داخل یک فولدر JDL\_system پیاده سازی کن. من از رویکرد چندعامله نمیخوام استفاده بشه.

من میخوام دو حالت centralize agent و distribute agents را درنظر بگیری.

در حالت centralize، تنها یک عامل وجود دارد که میخواهد دو کوئری q1 و q2 را به صورت متوالی اجرا کند به صورتی که نتایج q1 به q2 به عنوان ورودی وارد شود.

در حالت distribute agents ما دو عامل worker و master داریم. در عامل worker، کوئری q1 اجرا می شود و در عامل master کوئری q2 اجرا می گردد. لذا این دو کوئری روی دو عامل مجزا اجرا می گردد.

در هر دو سیستم (متمرکز و توزیعی) نتایج کوئری q1 روی یک صف (queue) به نام BUS قرار داده می شوند. یک فضای مشخصی برای BUS باید درنظر بگیری یعنی اینکه چند تا triple گنجایش دارد.

من میخوام نشان دهم که رویکرد توزیعی از رویکرد متمرکز بهتر است از منظر زمان اجرای کوئری، فضای حافظه مصرفی .

در حالت اول : من فقط میخوام که به تعداد 60 هزار triple، داده خام ارسال شود و عملیات در هر دو سیستم شروع به کار کند (اول متمرکز بعد توزیعی) و q1 اجرا و بعدش q2 اجرا شود. و نتایج کارایی (زمان پایان کار) مشخص شود. کارایی هم ارزیابی شود در هر دو روش.

در حالت دوم: من میخوام ...

**JDL\_SYStem (1)**

- دلیل اصلی بهتر شدن زمان در توزیعی: هم‌پوشانی مراحل. در روش توزیعی، تولید خروجی q1 (worker) و ساخت/انباشت گراف مشاهدات برای q2 (master) به‌صورت هم‌زمان پیش می‌رود. این پایپ‌لاین کردن، زمان بیکاری بین مراحل را حذف می‌کند؛ در متمرکز، q1 کاملاً تمام می‌شود و سپس q2 اجرا می‌شود.

- دلیل کاهش پیک حافظه: BUS با ظرفیت محدود نقش backpressure دارد و master به‌صورت تدریجی خروجی‌های q1 را در گراف وارد می‌کند. چون مصرف هم‌زمان با تولید رخ می‌دهد، داده‌های میانی کمتر در حافظه معطل می‌مانند؛ در نتیجه پیک حافظه پایین‌تر می‌آید.

- نکته فنی در پایتون: با وجود GIL، بخشی از کار rdflib در توابع C/IO-مانند انجام می‌شود و الگوی تولید-مصرف باعث هم‌پوشانی مؤثر می‌شود. بنابراین حتی با threads هم بهبود زمان مشاهده می‌شود.

اگر بخواهید بیشتر بهبود دهید:

- کاهش یا تنظیم ظرفیت BUS برای تعادل بین فشار حافظه و هم‌پوشانی

- افزایش موازی‌سازی واقعی با multiprocessing به‌جای threading برای بخش‌های CPU-bound

- اجرای q2 نهایی روی گراف در یک پردازش مجزا یا استفاده از backendهای سریع‌تر SPARQL

**JDL\_system (2)**

**Prompt:**

حالا میخوام این تغییرات را بدی.  
میخوام هر دو روش توزیعی و متمرکز به صورت هم‌پوشانی انجام شوند (رویکرد متمرکز هم مثل توزیعی باشد و همپوشانی داشته باشد).  
اما میخوام گه اجرای کوئری q2 روی متمرکز سنگین تر شود نسبت به اجرای q2 در توزییع و دیرتر بتواند داده از BUS بردارد و لذا BUS فول و پر بشه و منجر به توقفاجرای q1 بشه در متمرکز و اینطوری تاخیر بخوره (چون q2 نتونسته سریع داده ها بردارد از BUS و لذا BUS توسط خروجی های q1 پر شده و باید یک توقفی روی اجرای q1 انجام شود.)  
از طرفی این سنگین شدن اجرای کوئری q2 در متمرکز میخوام به صورت مصرف شدن زیاد RAM نشون داده بشه و ارزیابی ها که در متمرکز چون q2 سنگین بوده ram زیاد مصرف شده و BUS پر شده.  
اما در توزیعی جون فرض میکنیم هر کوئری روی یک سیستم مجرا است، اجرای q2 منجر به تاخیر در اجزای q1 نمی شود.

- Centralized

- Peak RAM ≈ 381 MB: مطابق تنظیمات شما، مصرف RAM اضافی 300MB در مصرف‌کننده q2 فعال بود (+ حافظه پایه پردازش). این باعث شد حافظه اوج به‌طور محسوسی بالا برود.

- تاخیر 20ms در مصرف هر batch: این کندی در مصرف باعث شد BUS گاهی پر شود و تولید q1 منتظر بماند (backpressure). با این وجود، هم‌پوشانی q1/q2 حفظ شد و کل زمان به 15.425s رسید.

- خروجی 12 تریپل: q2 یک‌بار روی کل گراف Observations اجرا شده و تعداد خروجی ثابت است.

- Distributed

- Peak RAM ≈ 59 MB: در رویکرد توزیعی، q2 سنگین‌سازی نشده و مصرف RAM پایین مانده است. ضمن اینکه تولید و مصرف روی دو نقش جدا انجام می‌شود و حافظه تجمعی در یک فرآیند کمتر می‌شود.

- زمان 15.903s: با وجود حافظه کمتر، سربار همزمانی/هماهنگی و انتقال داده بین نقش‌ها، و اجرای نهایی q2 روی گراف کامل باعث شده زمان کمی بیشتر از حالت مرکزی شود. هم‌پوشانی کمک کرده که هم‌سطح بماند، اما چون در مرکزی q2 هم‌فرایند است و هزینه‌های بین‌نقشی ندارد، کمی جلو افتاده است.

- خروجی 12 تریپل: همان منطق q2 روی گراف نهایی، پس خروجی برابر است.

- برداشت کلیدی

- تفاوت اصلی: در مرکزی q2 عمداً سنگین شده و RAM بالایی مصرف کرده که به پر شدن BUS و توقف‌های q1 منجر می‌شود، اما به‌خاطر نبود overhead بین‌فرایندی، زمان کل از توزیعی اندکی کمتر شده است. در توزیعی، حافظه بسیار کمتر است ولی اندکی سربار هماهنگی زمان را بالا برده است.

- اگر هدف نشان‌دادن برتری زمانی توزیعی هم هست، می‌توانید:

- ظرفیت BUS را کمتر کنید تا backpressure مؤثرتر شود.

- delay مرکزی را افزایش دهید (مثلاً 40–60ms).

- q2 توزیعی را سبک‌تر یا با chunkهای کوچک‌تر مصرف کنید تا سریع‌تر تخلیه کند.

این پارامتر مربوط به الگوی polling در master thread است:

- `master-poll-ms`: مدت زمان انتظار (به میلی‌ثانیه) که master قبل از تسلیم شدن منتظر می‌ماند

- وقتی master می‌خواهد از BUS داده بگیرد، ابتدا `bus.get(timeout=...)` را با این timeout فراخوانی می‌کند

- اگر در این مدت داده‌ای نرسد، master از حلقه خارج می‌شود (worker تمام شده)

- مقدار کوچک (1ms) یعنی master سریع تسلیم می‌شود و دوباره تلاش می‌کند

- مقدار بزرگ‌تر یعنی master بیشتر منتظر می‌ماند و کمتر polling می‌کند

مثال:

- `--master-poll-ms 1`: master هر 1ms چک می‌کند (CPU usage بالا)

- `--master-poll-ms 50`: master هر 50ms چک می‌کند (CPU usage پایین‌تر، اما latency بیشتر)

برای بهینه‌سازی: مقدار کوچک (1-5ms) برای latency پایین، مقدار بزرگ‌تر (20-50ms) برای CPU usage پایین‌تر.

در روش متمرکز فعلی:

\*\*بله، تاخیر وجود دارد\*\* به دلایل زیر:

1. \*\*تاخیر عمدی (Simulated Delay)\*\*:

- پارامتر `--central-q2-delay-ms` (پیش‌فرض 20ms)

- در هر batch که master از BUS می‌گیرد، `time.sleep(heavy\_q2\_delay\_ms / 1000.0)` اجرا می‌شود

- این تاخیر برای شبیه‌سازی سنگینی q2 اضافه شده

2. \*\*تاخیر طبیعی (Natural Delay)\*\*:

- \*\*Backpressure\*\*: وقتی BUS پر می‌شود، worker thread در `bus.put(t)` block می‌شود

- \*\*Thread Scheduling\*\*: زمان‌بندی بین worker و master thread

- \*\*Memory Allocation\*\*: تخصیص 300MB RAM اضافی در master

3. \*\*مکانیزم تاخیر\*\*:

```python

# در master function

if heavy\_q2\_delay\_ms > 0:

time.sleep(heavy\_q2\_delay\_ms / 1000.0) # تاخیر عمدی

```

\*\*نتیجه\*\*: q1 باید منتظر بماند تا q2 داده‌ها را از BUS بگیرد و پردازش کند. این تاخیر باعث می‌شود:

- BUS گاهی پر شود

- Worker thread block شود

- کل زمان اجرا افزایش یابد

این تاخیر عمداً طراحی شده تا نشان دهد در سیستم‌های متمرکز، کندی یک بخش روی کل سیستم تاثیر می‌گذارد.

این دو پارامتر مربوط به \*\*batching strategy\*\* در روش توزیعی هستند:

## `--worker-batch` (Worker Batch Size)

- \*\*معنی\*\*: تعداد تریپل‌هایی که worker قبل از ارسال به BUS جمع‌آوری می‌کند

- \*\*عملکرد\*\*: worker خروجی‌های q1 را در batch جمع می‌کند و با `bus.put\_many()` یکجا ارسال می‌کند

- \*\*مثال\*\*: اگر `worker-batch=2000` باشد، worker هر 2000 تریپل را یکجا به BUS می‌فرستد

- \*\*تأثیر\*\*:

- مقدار بزرگ = کمتر فراخوانی BUS = سریع‌تر

- مقدار کوچک = بیشتر فراخوانی BUS = کندتر

## `--master-batch` (Master Batch Size)

- \*\*معنی\*\*: تعداد تریپل‌هایی که master می‌خواهد در هر بار drain از BUS بگیرد

- \*\*عملکرد\*\*: master سعی می‌کند تا این تعداد تریپل از BUS بگیرد و سپس روی گراف اعمال کند

- \*\*مثال\*\*: اگر `master-batch=8000` باشد، master سعی می‌کند تا 8000 تریپل در هر بار drain بگیرد

- \*\*تأثیر\*\*:

- مقدار بزرگ = کمتر فراخوانی append\_triples\_to\_graph = سریع‌تر

- مقدار کوچک = بیشتر فراخوانی append\_triples\_to\_graph = کندتر

## رابطه بین این دو:

- `worker-batch` باید ≤ `master-batch` باشد تا master بتواند batch‌های کامل worker را بگیرد

- اگر `master-batch` خیلی بزرگ باشد، ممکن است منتظر بماند تا worker batch‌های بیشتری تولید کند

- اگر `worker-batch` خیلی کوچک باشد، تعداد فراخوانی‌های BUS زیاد می‌شود

\*\*پیشنهاد برای 300k داده\*\*:

```powershell

--worker-batch 15000 --master-batch 25000

```

این دو پارامتر مربوط به \*\*batching strategy\*\* در روش توزیعی هستند:

## `--worker-batch` (Worker Batch Size)

- \*\*معنی\*\*: تعداد تریپل‌هایی که worker قبل از ارسال به BUS جمع‌آوری می‌کند

- \*\*عملکرد\*\*: worker خروجی‌های q1 را در batch جمع می‌کند و با `bus.put\_many()` یکجا ارسال می‌کند

- \*\*مثال\*\*: اگر `worker-batch=2000` باشد، worker هر 2000 تریپل را یکجا به BUS می‌فرستد

- \*\*تأثیر\*\*:

- مقدار بزرگ = کمتر فراخوانی BUS = سریع‌تر

- مقدار کوچک = بیشتر فراخوانی BUS = کندتر

## `--master-batch` (Master Batch Size)

- \*\*معنی\*\*: تعداد تریپل‌هایی که master می‌خواهد در هر بار drain از BUS بگیرد

- \*\*عملکرد\*\*: master سعی می‌کند تا این تعداد تریپل از BUS بگیرد و سپس روی گراف اعمال کند

- \*\*مثال\*\*: اگر `master-batch=8000` باشد، master سعی می‌کند تا 8000 تریپل در هر بار drain بگیرد

- \*\*تأثیر\*\*:

- مقدار بزرگ = کمتر فراخوانی append\_triples\_to\_graph = سریع‌تر

- مقدار کوچک = بیشتر فراخوانی append\_triples\_to\_graph = کندتر

## رابطه بین این دو:

- `worker-batch` باید ≤ `master-batch` باشد تا master بتواند batch‌های کامل worker را بگیرد

- اگر `master-batch` خیلی بزرگ باشد، ممکن است منتظر بماند تا worker batch‌های بیشتری تولید کند

- اگر `worker-batch` خیلی کوچک باشد، تعداد فراخوانی‌های BUS زیاد می‌شود

\*\*پیشنهاد برای 300k داده\*\*:

```powershell

--worker-batch 15000 --master-batch 25000

```

این عملیات یعنی \*\*برای اجرای q2 که از BUS داده می‌خوانیم تاخیر زیاد دارد\*\*.

## توضیح دقیق:

### این کد در master thread (q2) اجرا می‌شود:

```python

# Apply delay per triple to make it much slower

for triple in batch:

if heavy\_q2\_delay\_ms > 0:

time.sleep(heavy\_q2\_delay\_ms / 1000.0) # تاخیر در پردازش q2

start\_t = time.perf\_counter()

append\_triples\_to\_graph(obs\_graph, [triple]) # اضافه کردن به گراف برای q2

transfer\_seconds += time.perf\_counter() - start\_t

```

### جریان:

1. \*\*Worker (q1)\*\*: خروجی‌ها را سریع به BUS اضافه می‌کند

```python

bus.put(t) # سریع

```

2. \*\*Master (q2)\*\*: از BUS می‌خواند اما \*\*کند پردازش می‌کند\*\*

```python

batch = bus.drain\_all() # خواندن از BUS

for triple in batch:

time.sleep(1.0) # تاخیر 1 ثانیه برای هر تریپل

append\_triples\_to\_graph(...) # پردازش کند

```

### نتیجه:

- \*\*q1 سریع\*\* داده تولید می‌کند

- \*\*q2 کند\*\* داده مصرف می‌کند

- BUS پر می‌شود

- q1 مجبور می‌شود منتظر بماند (backpressure)

**این شبیه‌سازی می‌کند که در سیستم متمرکز، کندی q2 روی کل سیستم تاثیر می‌گذارد و q1 را هم کند می‌کند.**

**زمان transfer در هر دو روش به همین شکل محاسبه می‌شود:**

**## محاسبه transfer time:**

**### در هر دو روش (مرکزی و توزیعی):**

**```python**

**start\_t = time.perf\_counter()**

**append\_triples\_to\_graph(obs\_graph, [triple]) # یا batch**

**transfer\_seconds += time.perf\_counter() - start\_t**

**```**

**## تفاوت در مقدار:**

**### مرکزی (transfer=0.001s):**

**- \*\*تاخیر زیاد\*\*: `time.sleep(1000ms)` قبل از هر `append\_triples\_to\_graph`**

**- \*\*تعداد کم\*\*: با BUS کوچک (100)، تعداد batch‌ها کم است**

**- \*\*نتیجه\*\*: زمان واقعی append کم است چون تعداد عملیات کم است**

**### توزیعی (transfer=0.001s):**

**- \*\*بدون تاخیر\*\*: هیچ `time.sleep` ندارد**

**- \*\*تعداد زیاد\*\*: با batch size بزرگ، تعداد عملیات append زیاد است**

**- \*\*نتیجه\*\*: زمان واقعی append کم است چون عملیات سریع است**

**## چرا هر دو کم هستند:**

**1. \*\*`append\_triples\_to\_graph` سریع است\*\*: فقط اضافه کردن تریپل به rdflib Graph**

**2. \*\*تاخیر در `time.sleep` محاسبه نمی‌شود\*\*: فقط زمان append محاسبه می‌شود**

**3. \*\*rdflib بهینه است\*\*: عملیات درون‌حافظه‌ای سریع**

**## اگر می‌خواهید transfer time واقعی ببینید:**

**- تاخیر را در `transfer\_seconds` هم محاسبه کنید**

**- یا عملیات سنگین‌تری در `append\_triples\_to\_graph` اضافه کنید**

**توضیحات:  
## شبیه‌سازی دنیای واقعی:**

**### \*\*BUS محدود در مرکزی (100 تریپل) = منابع محدود\*\***

**\*\*دنیای واقعی:\*\***

**- \*\*حافظه محدود\*\*: سرور مرکزی RAM کم دارد**

**- \*\*CPU محدود\*\*: پردازنده ضعیف یا overload شده**

**- \*\*شبکه محدود\*\*: پهنای باند کم بین اجزا**

**- \*\*I/O محدود\*\*: دیسک کند یا queue محدود**

**\*\*مثال واقعی:\*\***

**```**

**سرور مرکزی با 4GB RAM ← نمی‌تواند همه داده‌های IoT را نگه دارد**

**Database با 1000 connection limit ← نمی‌تواند همه درخواست‌ها را پردازش کند**

**```**

**### \*\*BUS نامحدود در توزیعی = منابع مستقل\*\***

**\*\*دنیای واقعی:\*\***

**- \*\*Edge Computing\*\*: هر سنسور/دستگاه حافظه خودش را دارد**

**- \*\*Microservices\*\*: هر سرویس منابع مستقل دارد**

**- \*\*Distributed Systems\*\*: هر node منابع مجزا دارد**

**\*\*مثال واقعی:\*\***

**```**

**IoT Gateway ← 8GB RAM مستقل**

**Cloud Service ← منابع نامحدود**

**Edge Device ← حافظه محلی خودش**

**```**

**### \*\*q2 کند در مرکزی = bottleneck در سیستم متمرکز\*\***

**\*\*دنیای واقعی:\*\***

**- \*\*Database bottleneck\*\*: query کند روی همه تاثیر می‌گذارد**

**- \*\*API rate limiting\*\*: کندی یک endpoint کل سیستم را کند می‌کند**

**- \*\*Single point of failure\*\*: مشکل یک بخش کل سیستم را متوقف می‌کند**

**\*\*مثال واقعی:\*\***

**```**

**Traffic Analysis Server ← اگر کند شود، همه سنسورها منتظر می‌مانند**

**Central Database ← اگر overload شود، همه سرویس‌ها کند می‌شوند**

**```**

**### \*\*q2 سریع در توزیعی = isolation و parallelism\*\***

**\*\*دنیای واقعی:\*\***

**- \*\*Service isolation\*\*: کندی یک سرویس روی بقیه تاثیر نمی‌گذارد**

**- \*\*Parallel processing\*\*: هر node مستقل کار می‌کند**

**- \*\*Fault tolerance\*\*: مشکل یک بخش بقیه را متوقف نمی‌کند**

**\*\*مثال واقعی:\*\***

**```**

**Smart City ← هر منطقه مستقل تحلیل می‌کند**

**Cloud Functions ← هر function مستقل اجرا می‌شود**

**Edge AI ← هر device مستقل تصمیم می‌گیرد**

**```**

**### \*\*Backpressure در مرکزی = سیستم‌های real-time\*\***

**\*\*دنیای واقعی:\*\***

**- \*\*Streaming systems\*\*: اگر consumer کند باشد، producer منتظر می‌ماند**

**- \*\*Message queues\*\*: اگر پردازش کند باشد، queue پر می‌شود**

**- \*\*Real-time analytics\*\*: اگر تحلیل کند باشد، داده‌ها از دست می‌روند**

**\*\*مثال واقعی:\*\***

**```**

**Stock Trading ← اگر analysis کند باشد، trades از دست می‌روند**

**IoT Monitoring ← اگر processing کند باشد، sensor data از دست می‌رود**

**Video Streaming ← اگر encoding کند باشد، کیفیت کاهش می‌یابد**

**```**

**### \*\*نتیجه‌گیری:\*\***

**این شبیه‌سازی نشان می‌دهد که:**

**- \*\*سیستم‌های متمرکز\*\*: مشکل یک بخش روی کل سیستم تاثیر می‌گذارد**

**- \*\*سیستم‌های توزیعی\*\*: هر بخش مستقل کار می‌کند و مشکل یک بخش بقیه را تحت تاثیر قرار نمی‌دهد**

**این همان دلیلی است که امروزه از microservices، edge computing، و distributed systems استفاده می‌شود.**