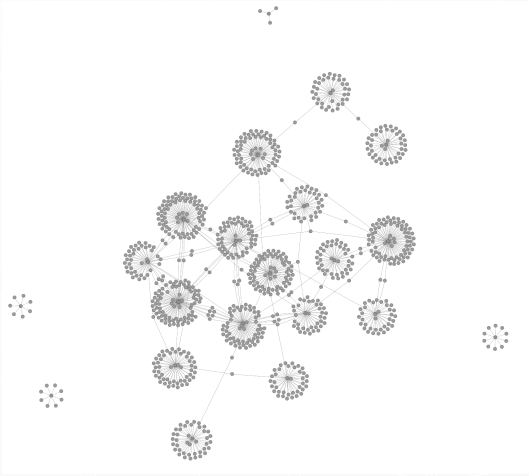
Chapter #1

今一番詳しいApache AGEのはなし

Author | rioriost
X/Twitter | @rioriost

概要 | Apache AGE をハックして Python パッケージを書きました



▲ 図 1： グラフデータの例

Apache AGE とは何か

Apache AGE は、Apache Graph Extension という名前の通り、グラフデータを扱うための PostgreSQL の拡張機能です。グラフデータはノードとエッジの組み合わせで表現され、ノードはデータの要素を表し、エッジはノード間の関係を表します。ノードやエッジにはプロパティ（属性）を付与することできます。AGE は Cypher というクエリ言語を使ってグラフデータを操作します。Cypher に対応するグラフデータベースとして最も有名なのは Neo4j[*1]ですが、他にも Apache TinkerPop というクエリ言語もあります。

変化し続ける PostgreSQL

PostgreSQL は通常、SQL（PL/pgSQL）を用いてクエリを記述します。拡張機能が無くても、配列型や CIDR 型など豊富な型表現を扱うことが出来ましたが、近年、PostgreSQL でも JSON やベクトル型のデータも扱えるようになり[*2]、さらにグラフデータも扱えるようになった、という状況にあります。

シナリオ 1・航空路

PostgreSQL に航空路線のデータを格納する例を考えてみましょう。現在世界には約 3,500 の空港があり、それらの空港を結ぶ航空路線は数万に及びますが、羽田空港からシアトルタコマ空港への直行便は、航空会社の別を考慮しなければ 1 路線しかありません。条件は「羽田発-シアトルタコマ着」のみです。

しかし乗り継ぎを考慮すると、羽田からシアトルタコマへの経路は複数通り存在します。クエリの条件は「羽田発-空港 (A) 着」「空港 (A) 発-シアトルタコマ着」です。さらに乗り継ぎが 2 回になると、「羽田発-空港 (A) 着」「空港 (A) 発-空港 (B) 着」「空港 (B) 着-シアトルタコマ着」となり、経路のパターンが増えることが分かります。空港（ノード）の数より、路線（エッジ）の数がかなり多くなるのはこういう理由です。

航空路線の場合、乗り継ぎ回数がそれほど増えることはないので、クエリもあまり難しくなるとは考えにくく、SQL で十分対応できるでしょう。

シナリオ 2・道路ネットワーク

では、さらに道路のネットワークを考えてみましょう。

ある地点（ノード）と隣接する別の地点を結ぶ道路（エッジ）は 1 本です。しかしあるノードが交差点である場合、そのノードに接続するエッジは複数本になります。

[*1] Apache AGE そのものではないですが、グラフデータを使った Neo4j と PostgreSQL の性能比較の例。https://courses.cs.washington.edu/courses/csed516/20au/projects/p06.pdf

[*2] 恐ろしいことに MongoDB のプロトコルを扱える pgMongo という拡張機能も存在します。https://github.com/thomas4019/pgmongo

日本国内の交差点は約 100 万あるとされているので、そこに接続しているエッジは T 字路で 3 本、2 本の道路が交差していれば 4 本、さらに五叉路や側道があるパターンでは、1 つのノードに 20 ぐらいのエッジが接続しているパターンも実在します。

航空路線に例えると、ハブ空港が日本国内だけで約 100 万あり、エッジがその 10 倍・約 1,000 万存在する、みたいなものです[*3]。そしてその巨大なネットワーク内で乗り継ぎを数十回、数百回するというのが、私たちが日常行っている、自動車の運転を含めた地上での移動です。

道路ネットワークよりもさらに大規模なネットワークとしては、SNS やウェブのリンクなども考えられます。

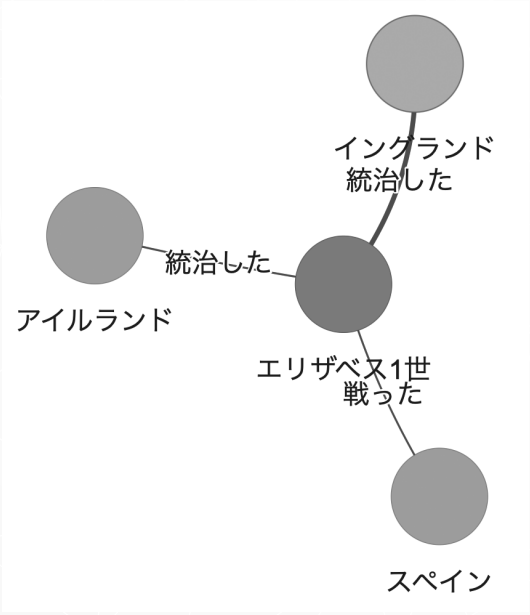
ここに挙げたいずれのシナリオでも共通するのは、ノードの総数に対してエッジの数が非常に多いということです。SQL でこのようなネットワーク内の経路を検索するのは非常に難しくなるため、「ノード A から最短距離でノード B に至る経路」や「ノード A からノード B を経由しノード C に至る経路」といった、より単純に記述できる Cypher や TinkerPop のようなクエリ言語が必要になります。

グラフデータと生成 AI

そしてご多分に漏れず、グラフデータにも生成 AI の波が訪れています。RAG[*4]における検索・生成精度の限界が認識されるようになり、次の手法の一つとして Knowledge Graph が注目されています。Knowledge Graph は、ノードが事実や概念を表し、エッジがそれらの関係を表します。例えば「エリザベス 1 世はイングランドとアイルランドを統治し、スペインと闘った」という事実をノードとエッジで表すと、「エリザベス 1 世」「イングランド」「アイルランド」「スペイン」というノードを、「統治した」「戦った」というエッジで結ぶことができます。このようなネットワーク構造を使って、AI による検索・生成結果を改善することが期待されています。

[*3] 現在、国内の空港は 97

[*4] Retrieval Augmented Generation



▲ 図 2： エリザベス 1 世と国家

AGE の使い方

前置きが長くなりました。ここからは紙面では分かりにくいと思うので、是非、手元で AGE を動かしながら読んでみてください。公式ドキュメントを参照して Docker で動かせば秒ですね[*5]。Azure で試す場合は 筆者のブログ[*6]を参照してください。

動いたら psql で接続し、search_path を設定します。

```
psql -h localhost -p 5455 -U postgresUser -d postgresDB
postgres=# SET search_path TO ag_catalog, "$user", public;
```

メタコマンド\d を実行して、ag_catalog スキーマのテーブルが 2 つ表示されれば、AGE が利用出来る状態です。

```
\d
                リレーション一覧
  スキーマ  |   名前   |  タイプ  |  所有者
------------+----------+----------+----------
 ag_catalog | ag_graph | テーブル | postgres
 ag_catalog | ag_label | テーブル | postgres
```

グラフを作成し、ノードとエッジを追加します。

```
SELECT create_graph('mygraph');
NOTICE:  graph "mygraph" has been created
 create_graph
--------------

SELECT * FROM cypher('mygraph',
  $$ CREATE (a:Person {name: "Alice"}) $$)
```

[*5] https://age.apache.org/age-manual/master/intro/setup.html

[*6] https://rio.st/prhf

```
  AS (a agtype);
 a
---

SELECT * FROM cypher('mygraph',
  $$ CREATE (b:Person {name: "Bob"}) $$)
  AS (b agtype);
 b
---

SELECT * FROM cypher('mygraph',
  $$ MATCH (a:Person {name:'Alice'}), (b:Person {name:'Bob'})
     CREATE (a)-[r:KNOWS]->(b) $$)
  AS (r agtype);
 r
---
```

「Alice は Bob を知っている」というグラフデータを作成しました。まずノードを確認してみましょう。

```
SELECT * FROM cypher('mygraph',
  $$ MATCH (a:Person)
  RETURN (a) $$) AS (a agtype);
                                  a
-----------------------------------------------------------------------
 {"id": 844424930131969, "label": "Person", "properties": {"nam.
.e": "Alice"}}::vertex
 {"id": 844424930131970, "label": "Person", "properties": {"nam.
.e": "Bob"}}::vertex
```

次にエッジを確認します。

```
SELECT * FROM cypher('mygraph',
  $$ MATCH (a:Person {name:'Alice'})-[r:KNOWS]-(b:Person {name:'
     Bob'})
  RETURN (r) $$) AS (r agtype);
                                  r
-----------------------------------------------------------------------
 {"id": 1125899906842625, "label": "KNOWS", "end_id": 844424930.
.131970, "start_id": 844424930131969, "properties": {}}::edge
```

良かった、Alice は Bob を知っているようです。

Apache AGE の裏側を覗く

さて、ここで読者諸氏は本稿が書かれている薄い本の名前を思い出してください。そう、ここはらぼちっく；げーとです。ヌルい話題はここには似合わない、あなたもそう思いますよね？では、参りますよ？

使い方で見た通り、Cypher クエリでグラフデータを追加しましたが、「どう見てもただのテーブルやんけ」と思ったそこのあなた。正解です。

上のメタコマンド\d を実行して表示された 2 つのテーブルに、どのようなデータが格納されたのかを見てみましょう。create_graph で作成したグラフの名称が ag_graph テーブルに格納されています。

```
SELECT * FROM ag_graph;
 graphid |  name   | namespace
---------+---------+-----------
   33352 | mygraph | mygraph
```

次に ag_label テーブルを見てみましょう。

```
SELECT * FROM ag_label;
   name   | graph | id | kind |    relation     |   seq_name
----------+-------+----+------+-----------------+--------------
 _ag_labe.| 33352 |  1 | v    | mygraph._ag_lab.| _ag_label_ver.
.l_vertex |       |    |      |.el_vertex       |.tex_id_seq
 _ag_labe.| 33352 |  2 | e    | mygraph._ag_lab.| _ag_label_edg.
.l_edge   |       |    |      |.el_edge         |.e_id_seq
 Person   | 33352 |  3 | v    | mygraph."Person.| Person_id_seq
          |       |    |      |."               |
 KNOWS    | 33352 |  4 | e    | mygraph."KNOWS" | KNOWS_id_seq
```

_ag_label_vertex、_ag_label_edge、Person、KNOWS の 4 つのテーブルが見えますね。では Person・KNOWS テーブルに格納されたデータを見てみましょう。FROM 句は graph_name."LABEL" のように、ダブルクォートが必要です。

```
SELECT * FROM my_graph."Person";
       id        |    properties
-----------------+------------------
 844424930131969 | {"name": "Alice"}
 844424930131970 | {"name": "Bob"}

SELECT id,start_id,end_id FROM mygraph."KNOWS";
        id         |    start_id     |     end_id
-------------------+-----------------+-----------------
 1125899906842625 | 844424930131969 | 844424930131970
```

もう分かってきましたね。そう、vertex には id が割り振られ、edge は start_id と end_id での組み合わせに edge としての id が割り振られているだけです。my_graph."Person" のテーブル定義を見てみましょう。

```
\d mygraph."Person"
                    テーブル"mygraph.Person"
   列    | タイプ  | 照合順序 | Null 値を許容 |   デフォルト
---------+---------+----------+---------------+-----------------
 id      | graphid |          | not null      | _graphid(_label.
         |         |          |               |._id('mygraph'::.
         |         |          |               |.name, 'Person'::.
         |         |          |               |.::name)::integer.
         |         |          |               |., nextval('mygr.
         |         |          |               |.aph."Person_id_.
         |         |          |               |.seq"'::regclass.
         |         |          |               |.))
 propert.| agtype  |          | not null      | agtype_build_ma.
.ies     |         |          |               |.p()
継承元: mygraph._ag_label_vertex
```

id は uint64 型の graphid となっていて、PostgreSQL のシーケンスを利用しています。properties は agtype 型ですが、これは AGE が提供する agtype_build_map() というユーザ定義関数[*7]が返す値がデフォルト値になります。

ひょっとして Cypher じゃなくても操作できるのでは、と思ったそこのあなた、第 1 問「ただのテーブルやんけ」に続いて正解です。

```
INSERT INTO mygraph."Person" (properties)
  VALUES ('{"name": "Charlie"}'::agtype);
INSERT 0 1

SELECT * FROM mygraph."Person"
  WHERE properties->'"name"' = '"Charlie"';
       id        |     properties
-----------------+--------------------
 844424930131971 | {"name": "Charlie"}

INSERT INTO mygraph."KNOWS" (start_id, end_id, properties)
  VALUES ('844424930131971', '844424930131969', '{}');
INSERT 0 1

SELECT id,start_id,end_id FROM mygraph."KNOWS";
        id         |    start_id     |     end_id
```

[*7] メタコマンド \dx+ age で function、cast、operator 等の一覧を見てみることをオススメします。

```
------------------+-----------------+-----------------
 1125899906842625 | 844424930131969 | 844424930131970
 1125899906842626 | 844424930131971 | 844424930131969
```

はい、「普通の SQL」でノードもエッジも作成することが出来ました。

ご注文は Cypher 抜き AGE ですか？

SQL 縛りゲーは以下です。

```
-- グラフ作成
SELECT create_graph('mygraph');
NOTICE:  graph "mygraph" has been created
 create_graph
--------------

-- ノードラベル作成
SELECT create_vlabel('mygraph', 'Person');
NOTICE:  VLabel "Person" has been created
 create_vlabel
---------------

-- ノード作成
INSERT INTO mygraph."Person" (properties) VALUES ('{"name":"
   Alice"}'), ('{"name":"Bob"}') RETURNING * ;
       id        |    properties
-----------------+------------------
 844424930131971 | {"name": "Alice"}
 844424930131972 | {"name": "Bob"}
INSERT 0 2

-- エッジラベル作成
SELECT create_elabel('mygraph', 'KNOWS');
NOTICE:  ELabel "KNOWS" has been created
 create_elabel
---------------

-- エッジ作成
INSERT INTO mygraph."KNOWS" (start_id, end_id) VALUES
   ('844424930131971', '844424930131972');
INSERT 0 1
```

ひどい、Cypher の欠片もない...

COPY もイケます

実は PostgreSQL の COPY コマンドでデータを投入することも可能ですが、ノードやエッジの id は not null 制約があるため事前に分かっている必要があります。id は次のように生成されています[*8]。

1. ag_label テーブルにおけるそのラベルの id を 48 ビット左シフト
2. そのラベルのシーケンス番号を 0x0000ffffffffffff でマスク
3. 上記 2 つを論理和

例えば、Person というラベルの ag_label テーブルにおける id が 3 であれば、最初に作成したノードの id は常に 844424930131969 になり、そのままエッジも作成した場合、ag_label における id は 4 になるため、最初に作成したエッジの id は常に 1125899906842625 になります。

このことが分かっていれば、Python や PL/pgSQL で COPY コマンドの入力として用いるファイルが簡単に生成できます。

とある AGE のインデックス

数万件程度のデータでは問題ないと思いますが、それを超えるデータになるとインデックスの張り方が重要になります。インデックスについては AGE の公式ドキュメントにも見つからなかったので、EXPLAIN でクエリプランを眺めながら検討した結果は以下です。

```
CREATE INDEX ON graph_name."VLABEL" USING GIN (properties);
CREATE INDEX ON graph_name."VLABEL" USING BTREE (id);
CREATE INDEX ON graph_name."ELABEL" USING BTREE (start_id);
CREATE INDEX ON graph_name."ELABEL" USING BTREE (end_id);
```

ノードを検索する際には properties の中身を検索するため GIN インデックスが必要となります。ただし、執筆時点の AGE v1.5 は部分インデックスを活用する手段が無く、properties の項目数が多い、あるいはデータが増えると GIN インデックスが肥大化することが懸念されます。

また、エッジを作成する際には start_id や end_id を検索します。インデックスが無い場合は Seq Scan となるため、見事なまでに線形に処理時間が増えていきます。つまりノードの id に対する btree インデックスが必要ということになります[*9]。

宣伝

筆者は AgeFreighter という Python パッケージ[*10]を作成しています。本稿を読んで興味が湧いた方は是非！

```
pip install agefreighter
```

著者情報

衛星経由で監視されているハムスター

[*8] 詳細については、筆者のブログ（https://rio.st/gv85）を参照のこと

[*9] \dx+ age の結果を見ても、btree / hash / gin に関する function が定義されています。

[*10] https://pypi.org/project/agefreighter/