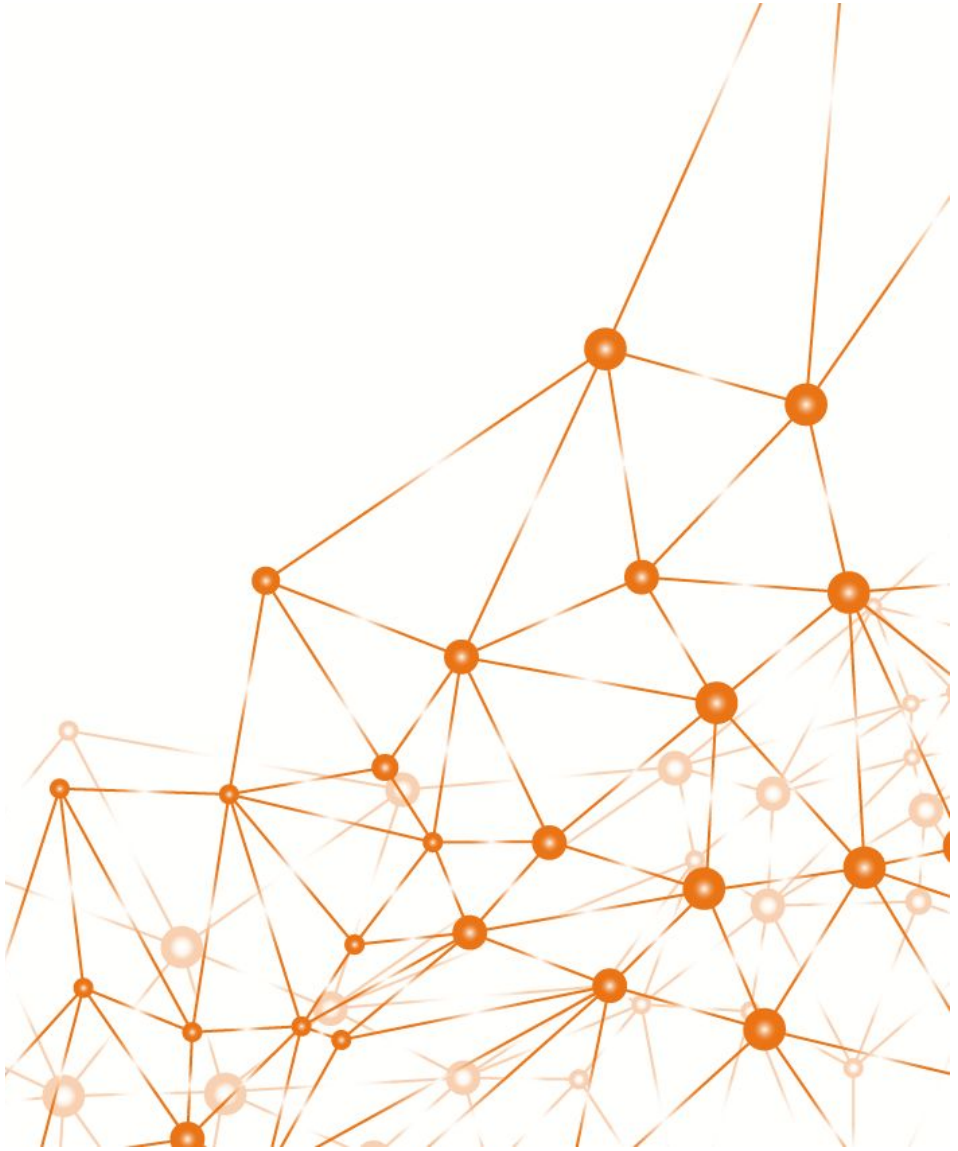
TigerGraph

Select Block Explained

Select Block Explained

1. Overview
2. Tutorial Setup
3. The "Hello world" of SELECT Statement
4. WHERE Clause
5. Accumulators
6. ACCUM Clause
7. POST-ACCUM Clause
8. HAVING Clause
9. ORDER BY + LIMIT



TigerGraph

Overview

Overview

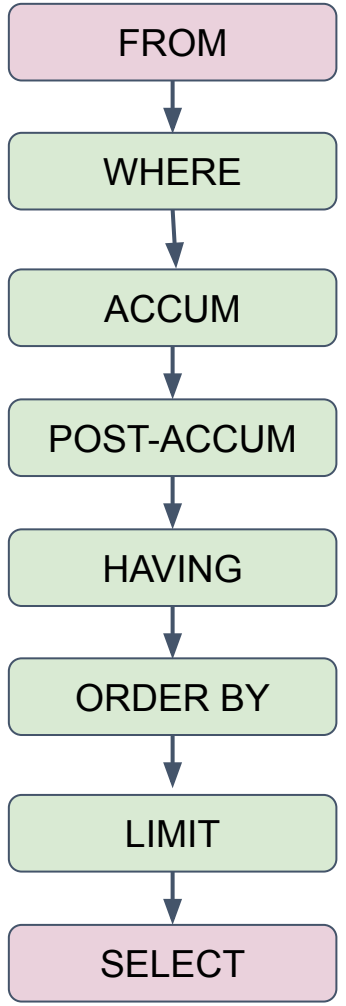
A SELECT Statement is the most fundamental block to build a query. It handles the logic of traversing from one set of vertices to another. This tutorial will guide you through all the essential mechanisms of a SELECT statement you need to know. Everything is explained in plain language and animation charts. In the end you will gain a clear understanding of how a SELECT statement works and know exactly how to implement your traversal logic in GSQL.

TigerGraph

Overview

```
resultSet = SELECT  vSet
    FROM ( edgeSet | vertexSet )
    [whereClause] [accumClause]
    [postAccumClause] [havingClause]
    [orderClause] [limitClause] ;
```

- **FROM**: select active vertices & edges.
- **WHERE**: conditionally filter the active sets
- **ACCUM**: iterate on edge set; compute with accumulators
- **POST-ACCUM**: iterate on vertex sets; compute with accumulators
- **HAVING**: conditionally filter the result set
- **ORDER BY**: sort
- **LIMIT**: max number of items
- **SELECT**: result from source or target set

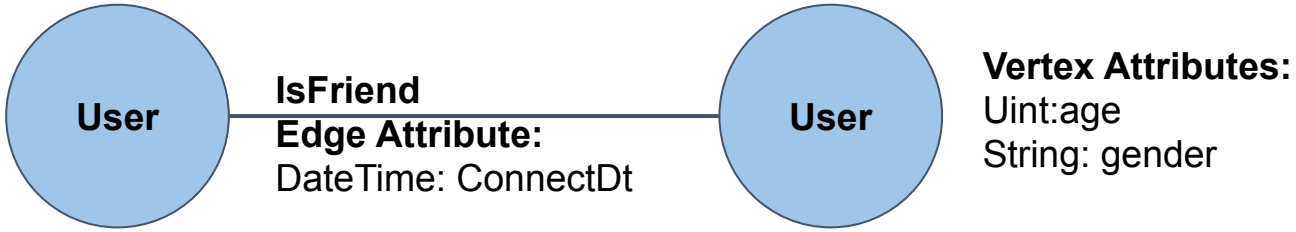


TigerGraph

Tutorial Setup

Tutorial Setup

In this tutorial the example will be given based on schema below. This is a social network of people who are connected as friends.



Vertex Attributes:
Uint:age
String: gender

TigerGraph

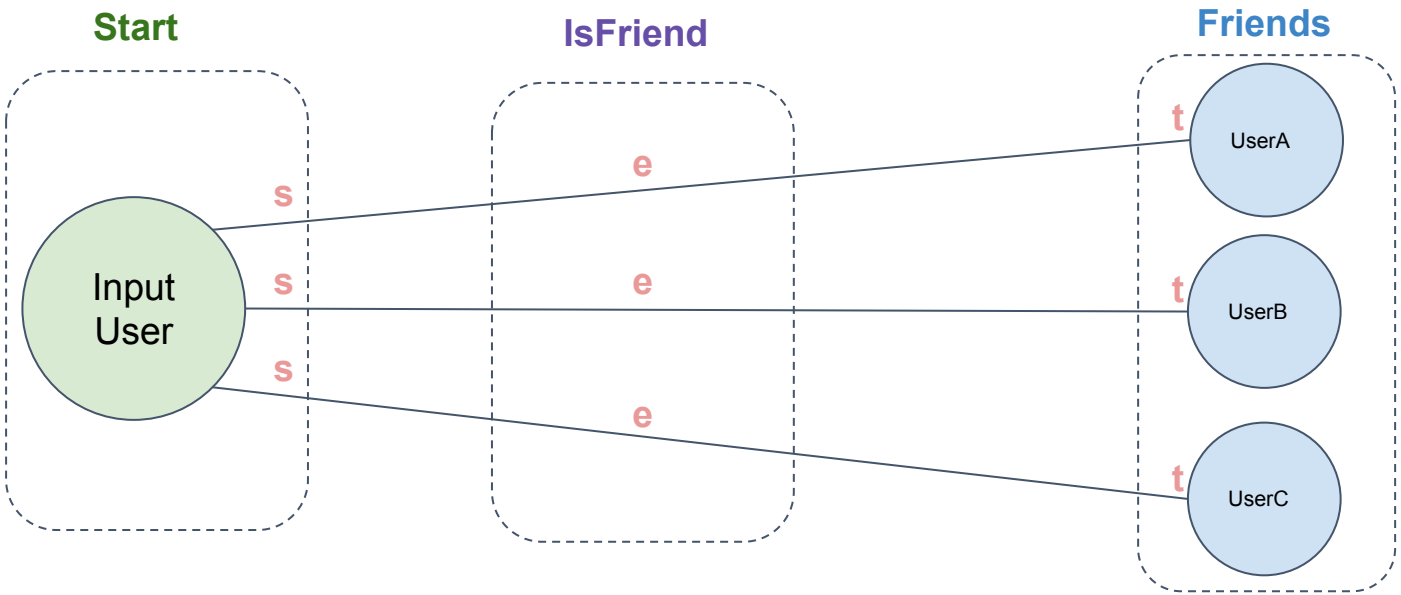
The “Hello world” of SELECT Statement

A Select Statement uses a set of edges to traverse from one set of vertices to another.

The “Hello world” of SELECT Statement

Find all the friends of a user.

```
CREATE QUERY GetFriends(vertex<User> inputUser) FOR GRAPH Social {
    Start = {inputUser};
    Friends = SELECT t FROM Start:s-(IsFriend:e)-User:t;
    PRINT Friends;
}
```



- **Start** is a vertex set initialized by the input vertex **inputUser**
- **FROM** clause finds edges which match the pattern: source vertex is in **Start**, edge type is **IsFriend**, and target vertex is restricted to User type.
- For each edge satisfies the conditions in the **FROM** clauses, **s**, **e** and **t** are aliases of source vertex, edge and target vertex.
- **s**, **e** and **t** are not keywords, you can rename them.
- **Friends** is a new vertex set equal to **t**.

TigerGraph

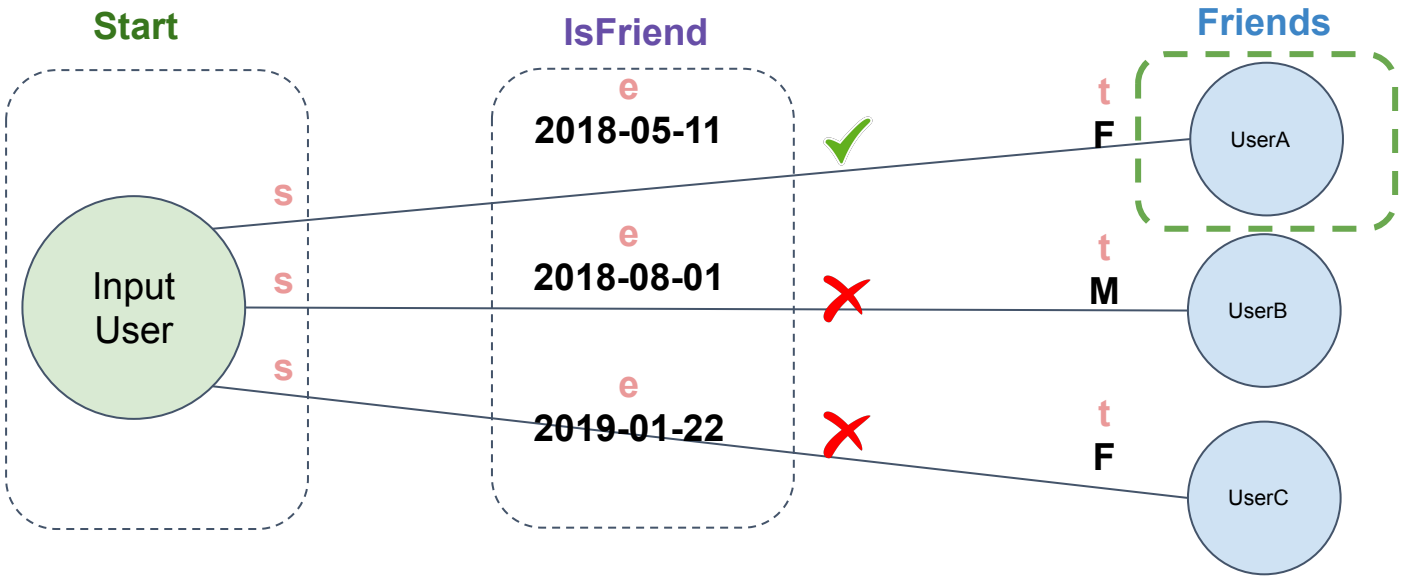
WHERE Clause

Where clause checks additional conditions for the selected set.

WHERE Clause

Find all the female friends that connected in 2018

```
CREATE QUERY GetFriends(vertex<User> inputUser) FOR GRAPH Social {
    Start = {inputUser};
    Friends = SELECT t FROM Start:s-(IsFriend:e)-:t
              WHERE e.connectDt BETWEEN to_datetime("2018-01-01") AND to_datetime("2019-01-01")
              AND   t.gender == "F";
    PRINT Friends;}
```

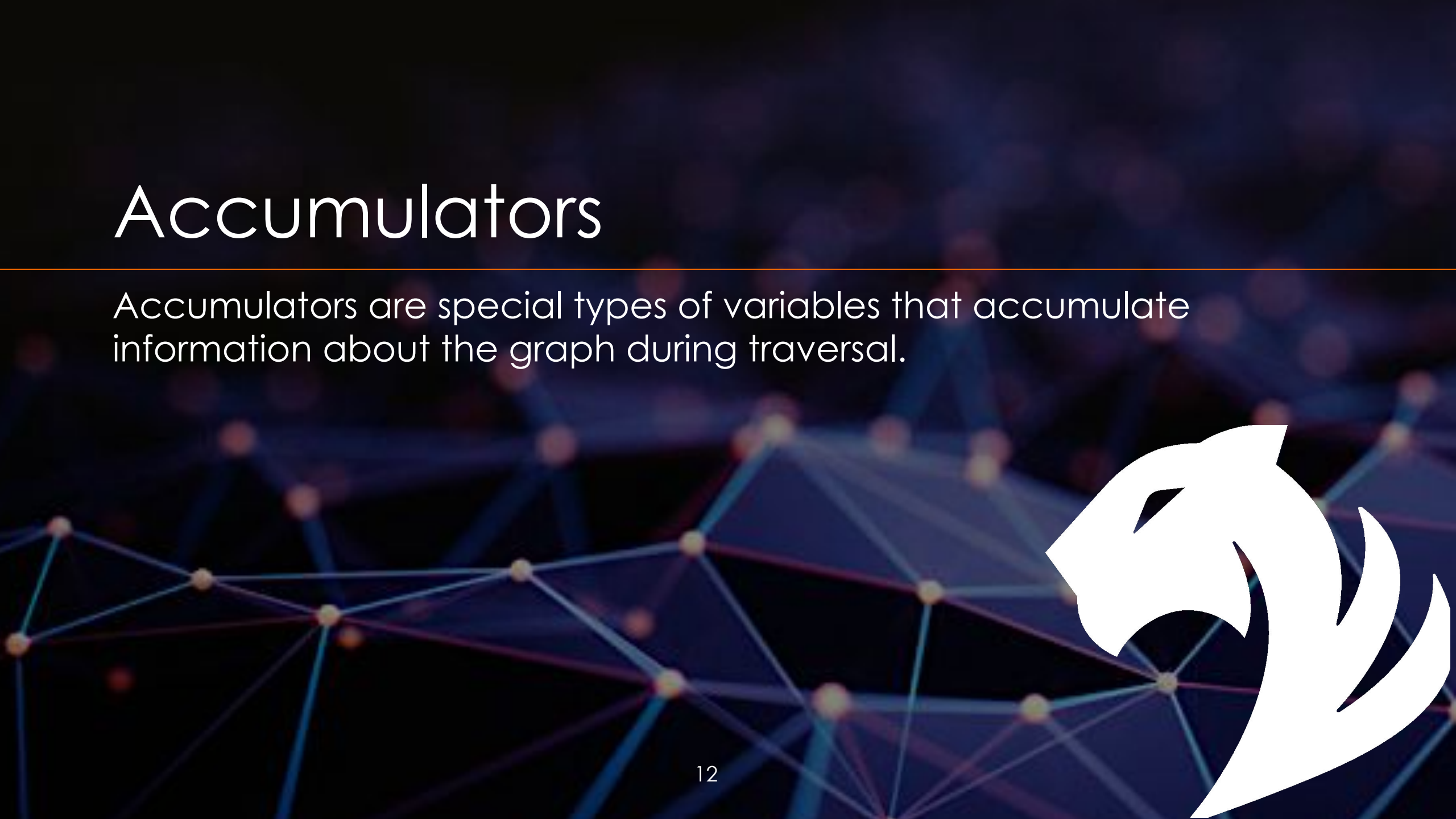


- All selected edges does checking defined in **WHERE** clause in parallel
- **WHERE** clause filter out edges not needed.
- **WHERE** clause has access to **s**, **e** and **t**.
- From **s**, **t** you can get vertex attributes and local accumulators. From **e** you can get edge attributes.
- Only **UserA** will be stored in **Friends**

TigerGraph

Accumulators

Accumulators are special types of variables that accumulate information about the graph during traversal.

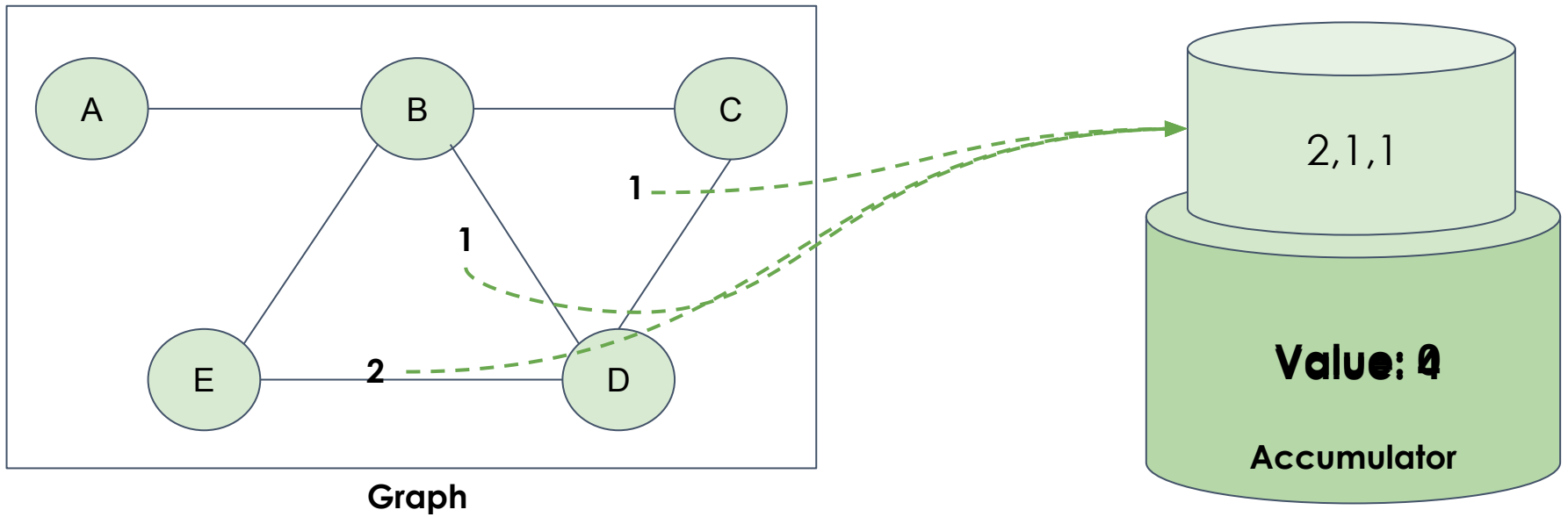


Accumulators

Accumulators are special type of variables that accumulate information about the graph during the traversal.

Accumulating phase 1: receiving messages, the messages received will be temporarily put to a bucket that belongs to the accumulator.

Accumulating phase 2: The accumulator will aggregate the messages it received based on its accumulator type. The aggregated value will become the accumulator's value, and its value can be accessed.



TigerGraph

Accumulators

For example:

The teacher collects test papers from all students and calculates an average score.

Teacher: accumulator

Student: vertex/edge

Test paper: message sent to accumulator

Average Score: final value of accumulator

Phase 1: teacher collects all the test paper

Phase 2: teacher grades it and calculate the average score.

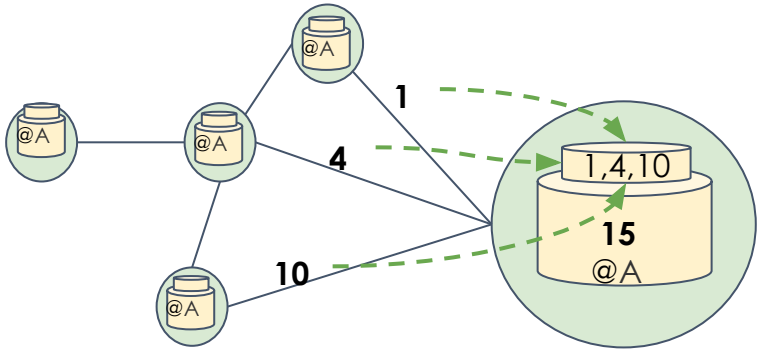
TigerGraph

Accumulators

Local Accumulators:

- Each selected vertex has its own accumulator.
- Local means per vertex. Each vertex does its own processing and considers what it can see/read/write.

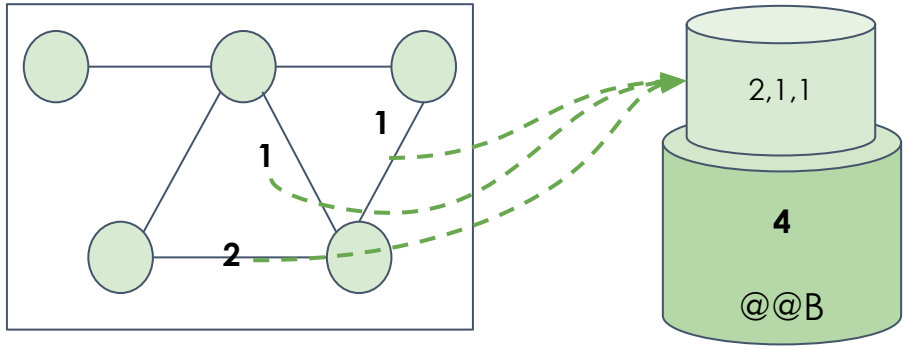
e.x. SumAccum @A;



Global Accumulators:

- Stored in stored globally, visible to all.
- All vertices and edges have access.

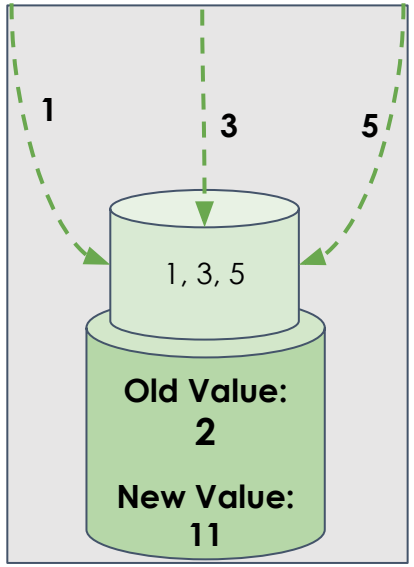
e.x. SumAccum @@B;



TigerGraph

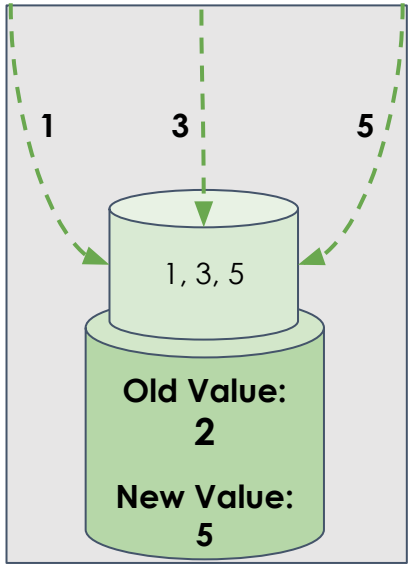
Accumulators

The GSQL language provides many different accumulators, which follow the same rules for receiving and accessing data. However each of them has their unique way of **aggregating values.**



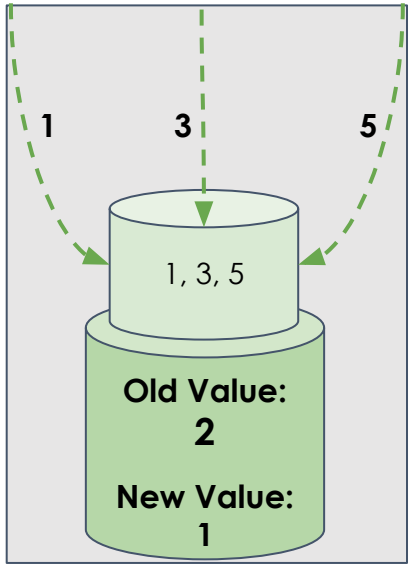
SumAccum<int>

Computes and stores the cumulative sum of numeric values or the cumulative concatenation of text values.



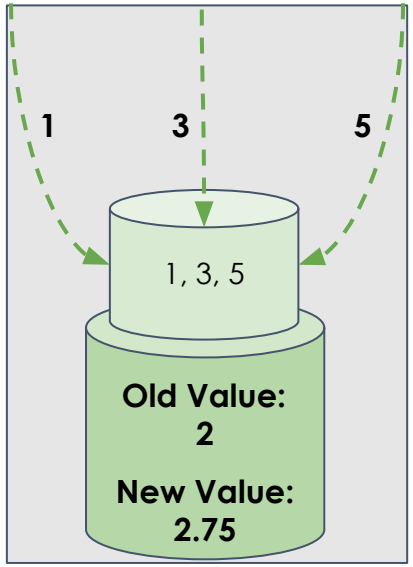
MaxAccum<int>

The MaxAccum types calculate and store the cumulative maximum of a series of values.



MinAccum<int>

The MinAccum types calculate and store the cumulative minimum of a series of values.



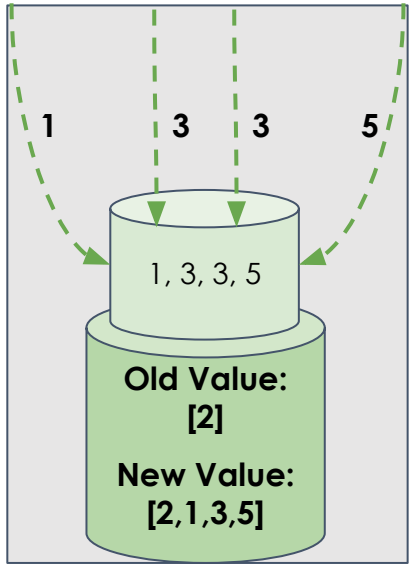
AvgAccum

Calculates and stores the cumulative mean of a series of numeric values.

TigerGraph

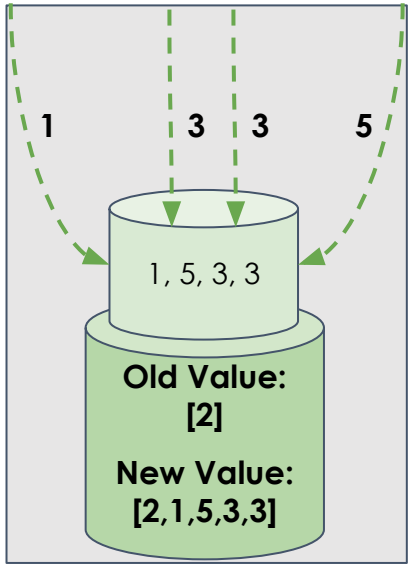
Accumulators

The GSQL language provides many different accumulators, which follow the same rules for receiving and accessing data. However each of them has their unique way of **aggregating values.**



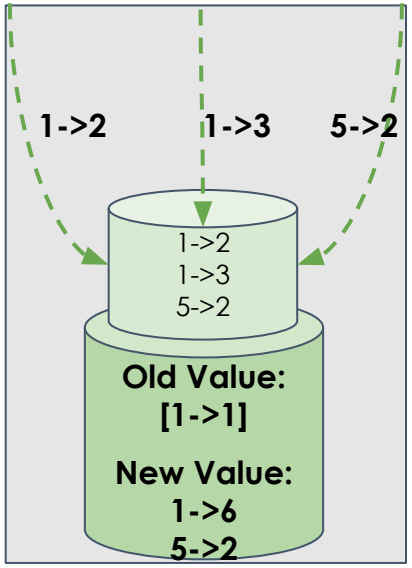
SetAccum<int>

Maintains a collection of unique elements.



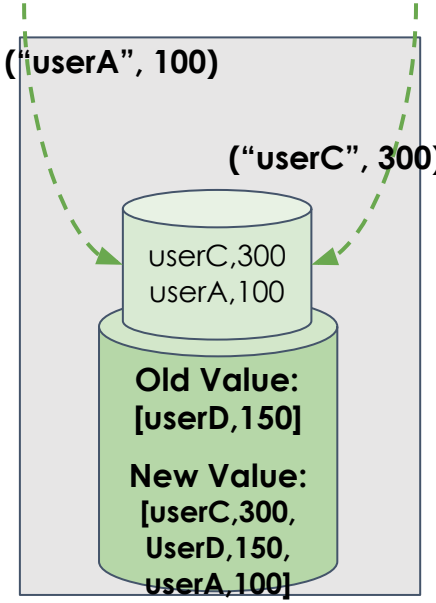
ListAccum<int>

Maintains a sequential collection of elements.



MapAccum<int,SumAccum<int>>

Maintains a collection of (key -> value) pairs.



HeapAccum<Tuple>

Maintains a sorted collection of tuples and enforces a maximum number of tuples in the collection

TigerGraph

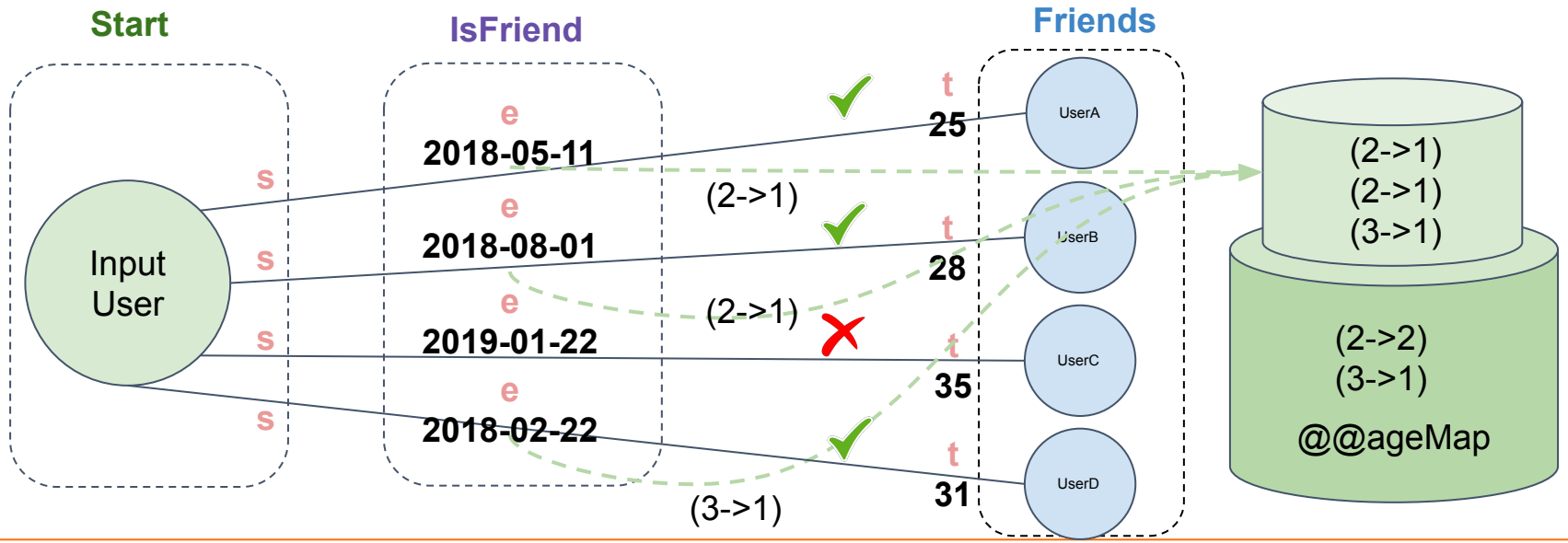
ACCUM Clause

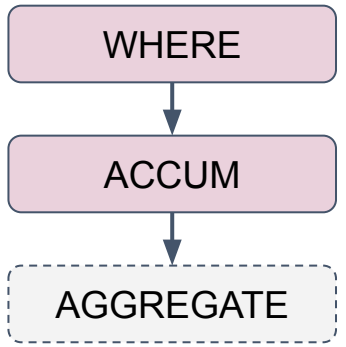
Logic defined in ACCUM clause executes on each edge in parallel.

ACCUM Clause

What is the age distribution of friends that were registered in 2018?

```
CREATE QUERY GetFriends(vertex<User> inputUser) FOR GRAPH Social {
    MapAccum<uint, SumAccum<uint>> @@ageMap;
    Start = {inputUser};
    Friends = SELECT t FROM Start:s-(IsFriend:e)-:t
              WHERE e.connectDt BETWEEN to_datetime("2018-01-01")
                AND to_datetime("2019-01-01")
              ACCUM @@ageMap += (t.age/10->1);
    PRINT @@ageMap;
}
```





Select the matching edges

Local compute + send messages

Aggregate the messages to accumulator

- Each edge satisfying the FROM & WHERE clauses performs the **ACCUM** clause statements.
- **ACCUM** has access to **s**, **e** and **t.**
- In **ACCUM,** vertices do not see each other's updates b/c updates aren't processed until the **AGGREGATE** step.
- The **AGGREGATE** phase is done automatically after **ACCUM**. After that, the updated accumulator value can be accessed
- **+=** means sending message to accumulator

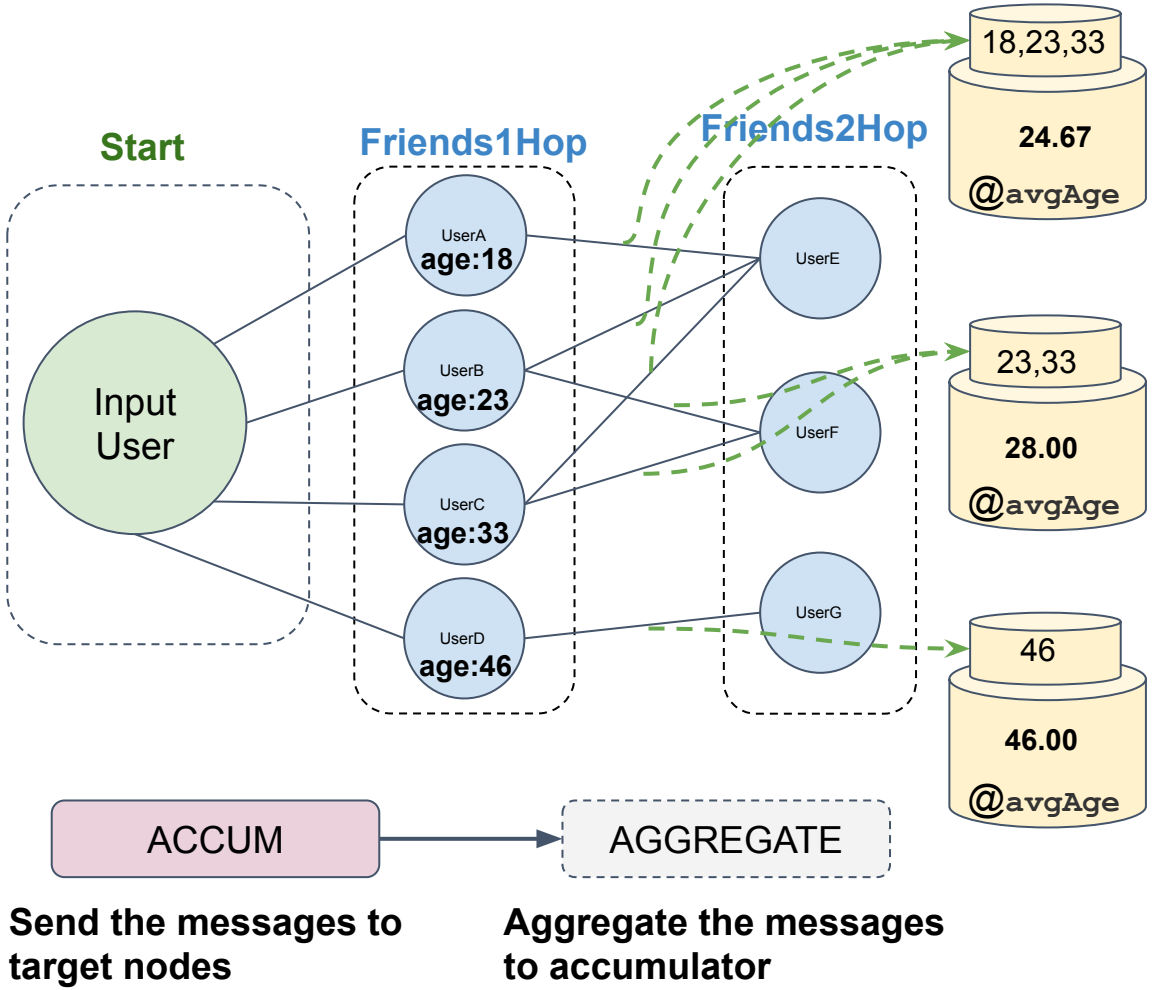
TigerGraph

ACCUM Clause

Given an input user. Output the average age of their common friends.

```
CREATE QUERY GetFriends(vertex<User> inputUser) FOR
GRAPH Social {
    AvgAccum @avgAge;
    Start = {inputUser};
    Friends1Hop = SELECT t FROM Start:s-(IsFriend:e)-:t;
    Friends2Hop = SELECT t
                  FROM Friends1Hop:s-(IsFriend:e)-:t
                  ACCUM t.@avgAge += s.age;
    print Friends2Hop;
}
```

- Update of local accumulator cannot be seen during **ACCUM** phase
- The messages will be aggregated during **AGGREGATE** phase based on accumulator type.



TigerGraph

POST-ACCUM

Logic defined in POST-ACCUM executes on each selected vertex in parallel.

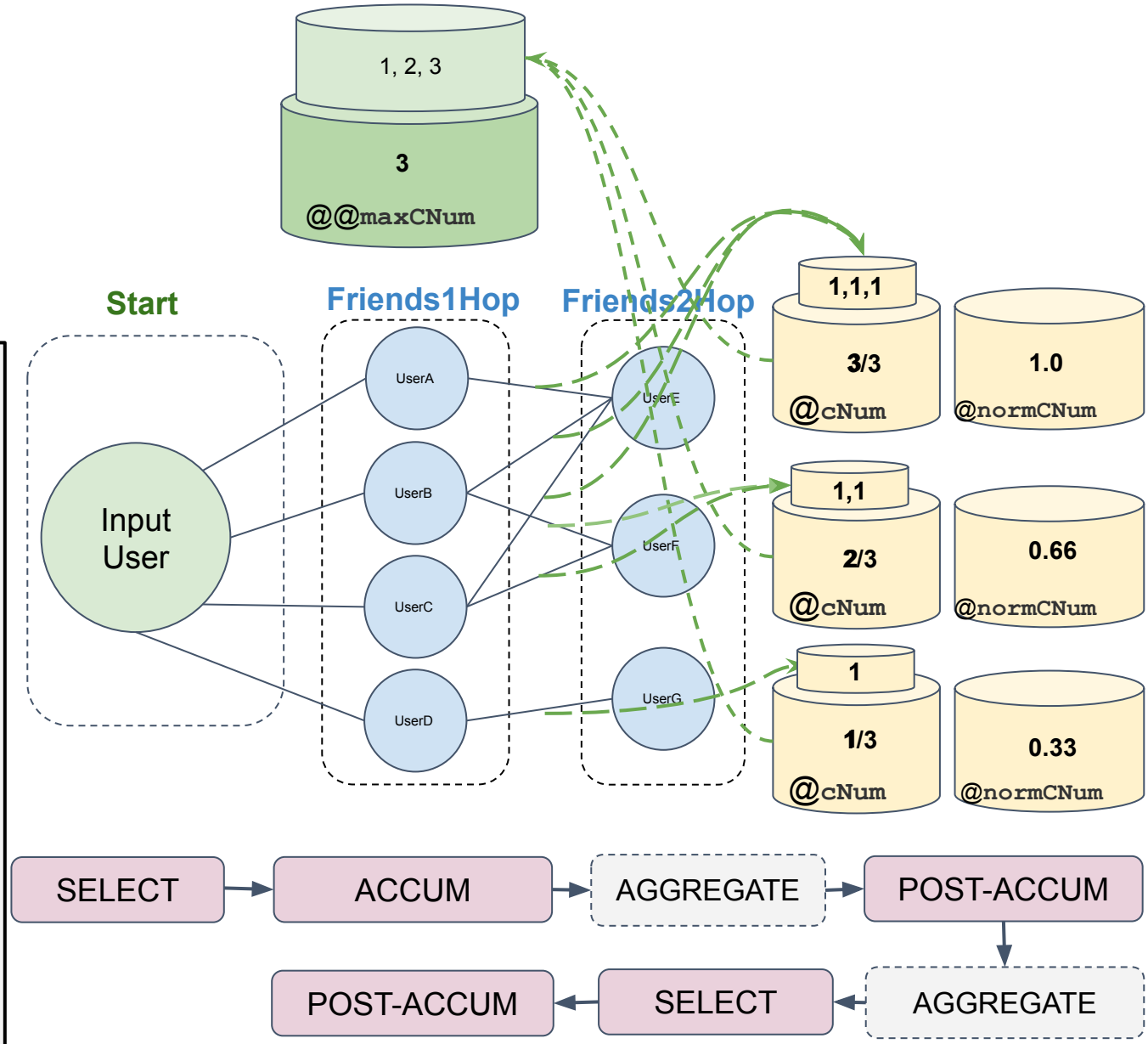
POST-ACCUM

- **POST-ACCUM** always comes after **ACCUM**, but operates independently from the **ACCUM** clause.
- **POST-ACCUM** does not require you to use **ACCUM**.
- **POST-ACCUM's** logic executes on all mentioned vertices in parallel, allowing it to access **s** and **t.**
- Local accumulators can immediately be accessed after updating in the **POST-ACCUM** clause, but global accumulators cannot be accessed in the **POST-ACCUM** until **aggregation** is done.

POST-ACCUM

Given a set of persons (friends of friends of the inputUser), output the normalized number of common friends for each person in the set.

```
1 CREATE QUERY GetFriends(vertex<User> inputUser) FOR
GRAPH Social {
2     SumAccum<uint> @cNum;
3     SumAccum<float> @normCNum;
4     MaxAccum<float> @@maxCNum;
5     Start = {inputUser};
6     Friends1Hop = SELECT t FROM Start:s-(IsFriend:e)-:t;
7     Friends2Hop = SELECT t
8                   FROM Friends1Hop:s-(IsFriend:e)-:t
9                   ACCUM t.@cNum += 1
10                  POST-ACCUM @@maxCNum += t.@cNum;
11    Friends2Hop = select s FROM Friends2Hop:s
12                  POST-ACCUM
13                  s.@normCNum = s.@cNum/@@maxCNum;
14    print Friends2Hop;
15 }
```



TigerGraph

POST-ACCUM

Given a set of persons (friends of friends of the inputUser), output the normalized number of common friends for each person in the set.

```
1 CREATE QUERY GetFriends(vertex<User> inputUser) FOR
GRAPH Social {
2     SumAccum<uint> @cNum;
3     SumAccum<float> @normCNum;
4     MaxAccum<float> @@maxCNum;
5     Start = {inputUser};
6     Friends1Hop = SELECT t FROM Start:s-(IsFriend:e)-:t;
7     Friends2Hop = SELECT t
8                   FROM Friends1Hop:s-(IsFriend:e)-:t
9                   ACCUM t.@cNum += 1
10                  POST-ACCUM @@maxCNum += t.@cNum;
11    Friends2Hop = select s FROM Friends2Hop:s
12                  POST-ACCUM
13                  s.@normCNum = s.@cNum/@@maxCNum;
14    print Friends2Hop;
15 }
```

- Each vertex satisfying the FROM & WHERE clauses performs the block of statements in the **POST-ACCUM** clause.
- The statements with a block are executed in order, but each block execution (per vertex) may be in parallel with other block executions.
- Each **POST-ACCUM** statement may access either the **s** or **t** alias.
- Global accumulator updates are queued but do not take effect until exiting the **POST-ACCUM** clause. So, accumulator values that are seen during the POST-ACCUM clause are those that existed when the preceding ACCUM block (if used) existed.

TigerGraph

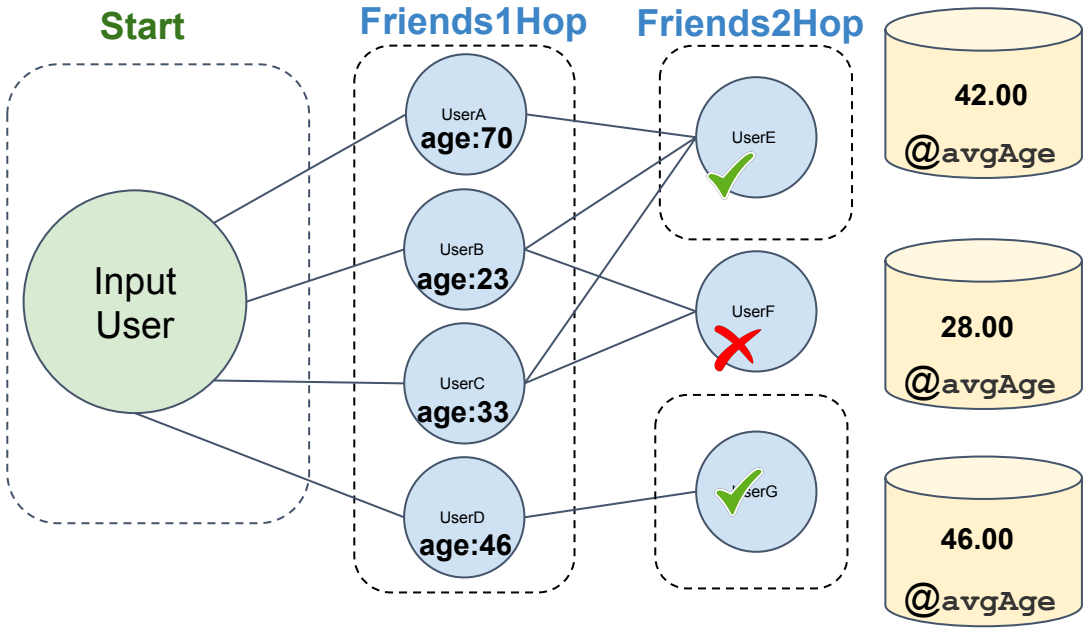
HAVING Clause

Having Clause filters out vertices after ACCUM AND POST-ACCUM, based on attributes and local accumulators.

HAVING

Given a set of selected vertices (friends of friends of the inputUser), output those whose friends in common with the inputUser have an average age over 30.

```
CREATE QUERY GetFriends(vertex<User> inputUser) FOR GRAPH
Social {
    AvgAccum @avgAge;
    Start = {inputUser};
    Friends1Hop = SELECT t FROM Start:s-(IsFriend:e)-:t;
    Friends2Hop = SELECT t
                  FROM Friends1Hop:s-(IsFriend:e)-:t
                  ACCUM t.@avgAge += s.age
                  HAVING t.@avgAge > 30;
    print Friends2Hop;
}
```



TigerGraph

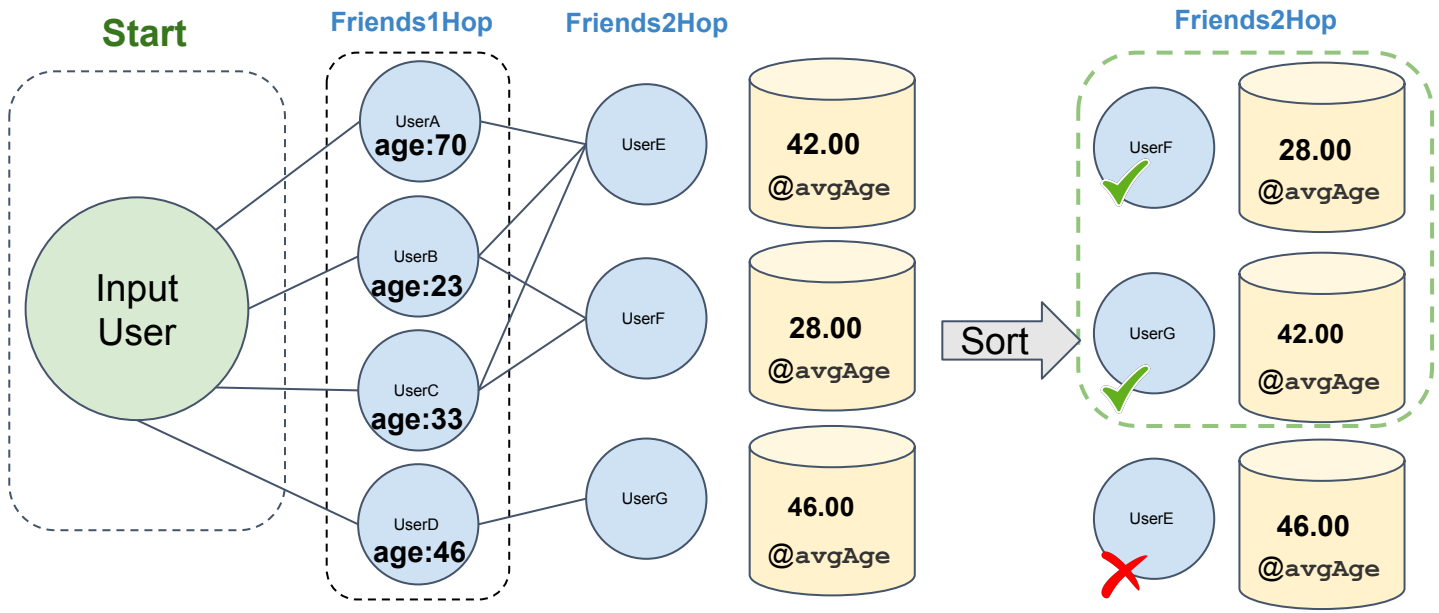
ORDER BY + LIMIT

After all other operations are done, order the selected vertices by the specified fields, then keep the first k.

ORDER BY + LIMIT

Given a set of selected vertices (friends of friends of the inputUser), output the first 2, ordered by increasing average age of the friends in common with the inputUser.

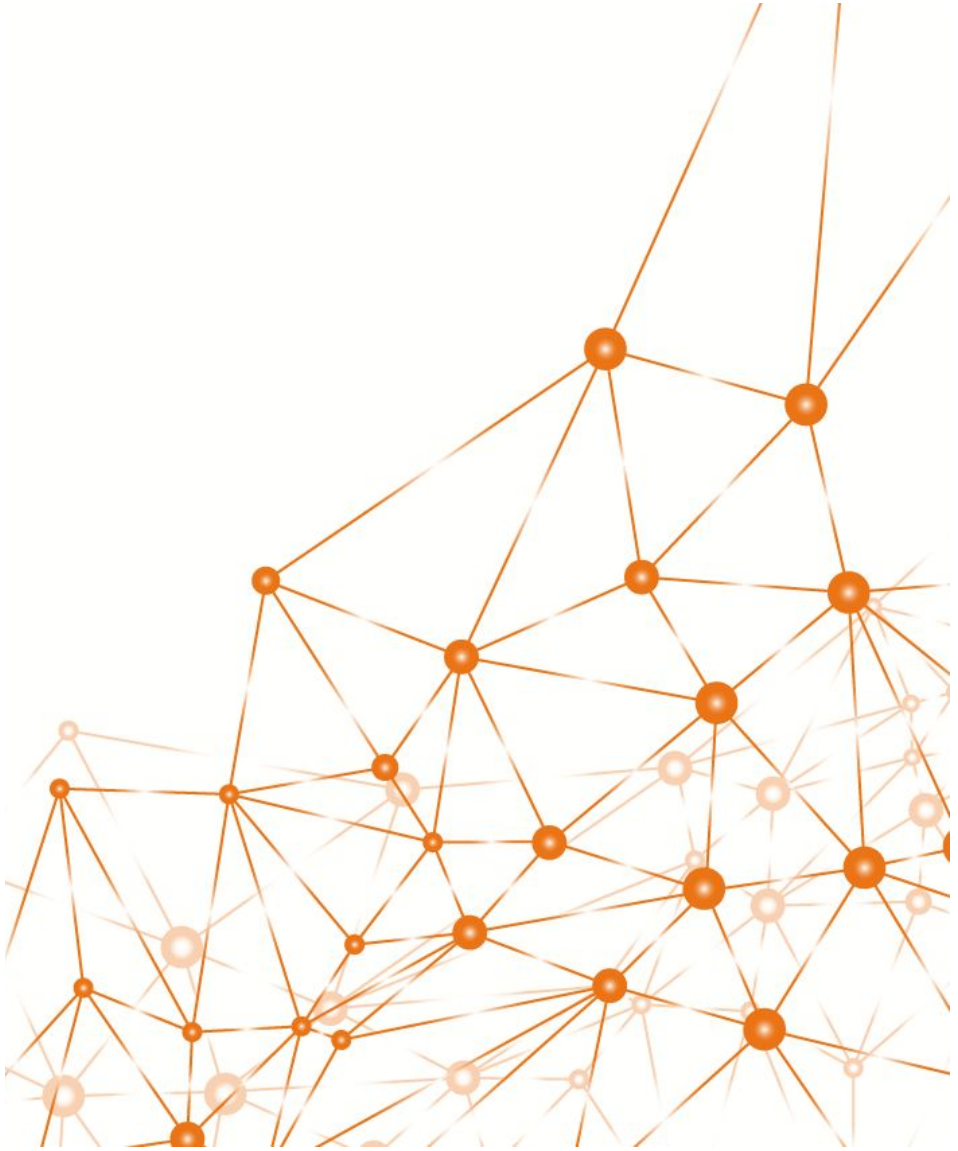
```
CREATE QUERY GetFriends(vertex<User> inputUser) FOR
GRAPH Social {
    AvgAccum @avgAge;
    Start = {inputUser};
    Friends1Hop = SELECT t FROM
                  Start:s-(IsFriend:e)-:t;
    Friends2Hop = SELECT t FROM
                  Friends1Hop:s-(IsFriend:e)-:t
                  ACCUM t.@avgAge += s.age
                  ORDER BY t.@avgAge ASC
                  LIMIT 2;
    print Friends2Hop;
}
```



TigerGraph

Summary

Through this tutorial now you have learned the mechanism of a SELECT block. Including WHERE clause, Accumulators, ACCUM clause, POST-ACCUM clause, HAVING clause and ORDER BY + LIMIT. Now you should be able to manipulate the traversal logic and do graph analytic easily. For more features of GSQL and details please refer to our user documentation.



TigerGraph