Best Practices

Xinyu Chang, Director of Customer Solutions

Agenda

1. Schema Design Best Practices
2. Native Storage and MPP Architecture
3. How does a SELECT statement work
4. Query Writing Best Practices

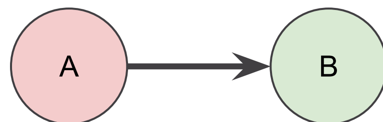
Schema Design Best Practices

Choosing an Edge Type: Undirected? Directed? Reversed?

Undirected Edge

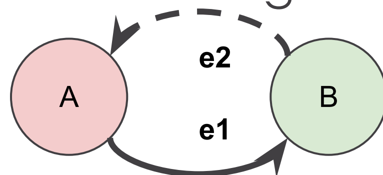
A can traverse to B, and B can traverse to A

Directed Edge



A can traverse to B, but B **cannot** traverse to A

Directed Edge + Reverse Edge



A can traverse to B via e1, B can traverse to A via e2
(e2 is automatically created upon creation of e1.
e2's attributes have the same values as e1's)

- What is the difference between undirected edge and directed edge + reverse edge?
- What to know when making edge type choices ?

© 2019 TigerGraph. All Rights Reserved

TigerGraph

Choosing an Edge Type: Undirected? Directed? Reversed?

Undirected Edge

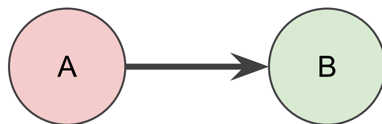
A can traverse to B, and B can traverse to A

Pros: Simple when working with undirected (symmetric) or bidirectional relationships.

Example: "A friend_of B" ⇔ "B friend_of A"

Cons: Does not carry directional info.

Directed Edge



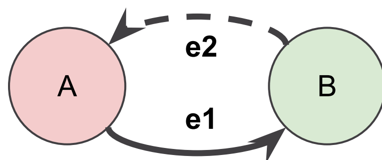
A can traverse to B, but B **cannot** traverse to A

Pros: Saves memory and correctly describes a direction-restricted relationship

Example: "A parent_of B" ⇏ "B parent_of A"

Cons: Can not traverse back from target to source.

Directed Edge + Reverse Edge



A can traverse to B via e1, B can traverse to A via e2

Pros: Flexibility to traverse in either designated direction.

Example: e1 type is "parent_of" and e2 type is "child_of"

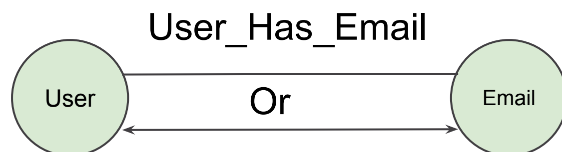
Cons: Need to remember two edge types.

© 2019 TigerGraph. All Rights Reserved

TigerGraph

Choosing an Edge Type: Undirected? Directed? Reversed?

Given Schema:



Find users share the same email

with undirected edge:

```
user_share_email = SELECT t FROM start-(User_Has_Email*2)-:t;
```

with directed edge + reverse edge

```
user_share_email = SELECT t FROM start-(User_Has_Email>)-email-(reverse_User_Has_Email>)-:t;
```


In this case, it is more concise to use undirected edge

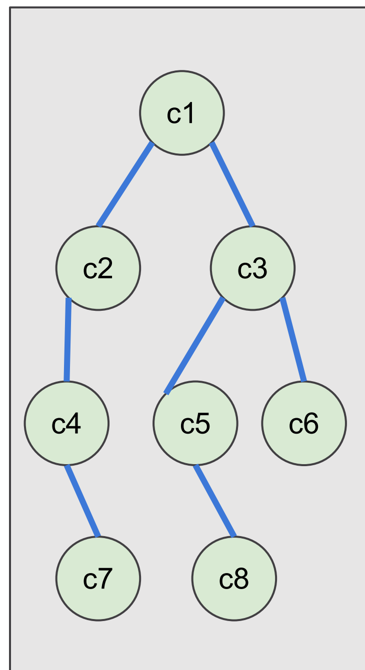
© 2019 TigerGraph. All Rights Reserved

TigerGraph

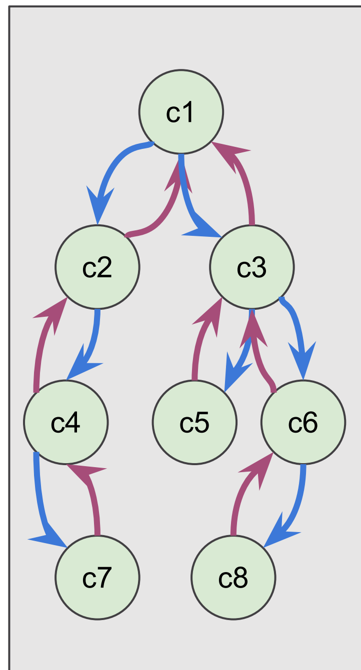
Choosing an Edge Type: Undirected? Directed? Reversed?

Use Case:
Given an enterprise graph and an input company, find its ultimate parent company and the ultimate child branches

Undirected Edge



Directed Edge +
Reverse Edge



In this use case, the question is hard to answer by using undirected edges, since they do not provide any directional info (parent vs. child)

However it can be easily solved by using directed edge + reversed edge.

When querying for parent companies it can use the red edge, and when querying for child companies it can use the reverse edge (blue).

© 2019 TigerGraph. All Rights Reserved

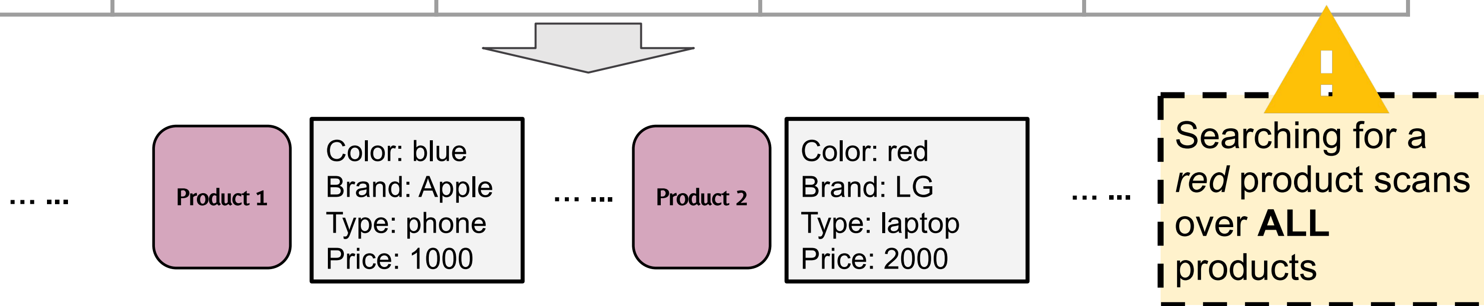
TigerGraph

Attribute or Vertex?

Given a column, should it be defined as an attribute or a vertex?

Product Table

Product Name	Color	Brand	Type	Price
product 1	blue	Apple	phone	1000
product 2	red	LG	laptop	2000
...	...	...	...	...

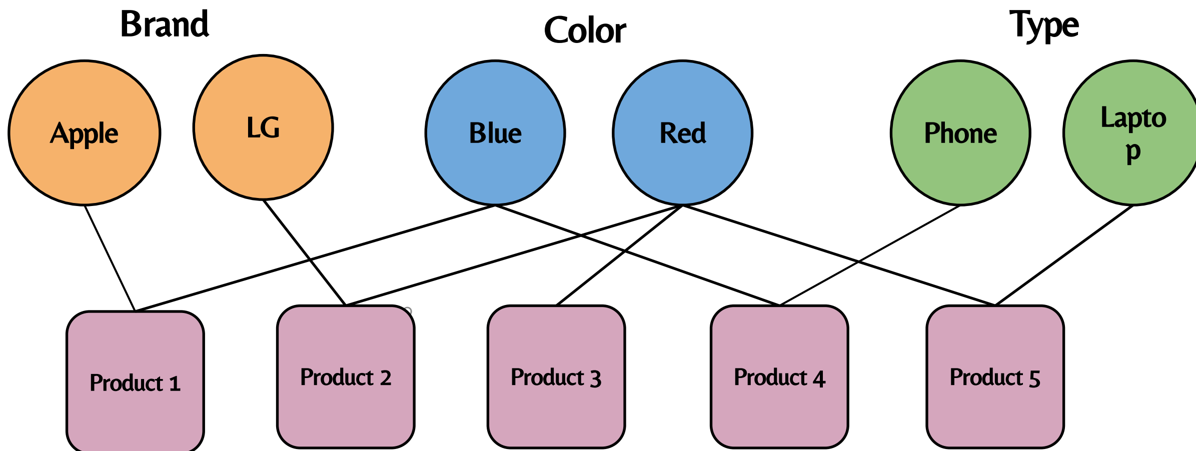


© 2019 TigerGraph. All Rights Reserved

TigerGraph

Attribute or Vertex?

It can be beneficial to represent a column (attribute) as a vertex type if you will frequently need to query for particular values of the property. This way, the vertices act like an search index. E.g., all the red products are connected to the **Red** vertex under **Color** type.



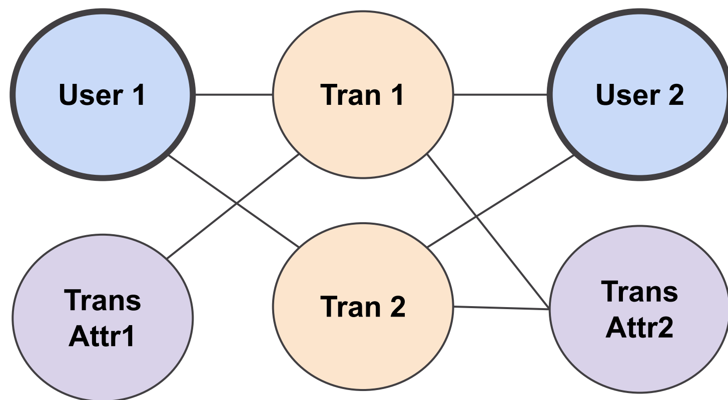
© 2019 TigerGraph. All Rights Reserved

TigerGraph

Multiple Events/Transactions Between Two Entities

Method 1: Each Event as a Vertex

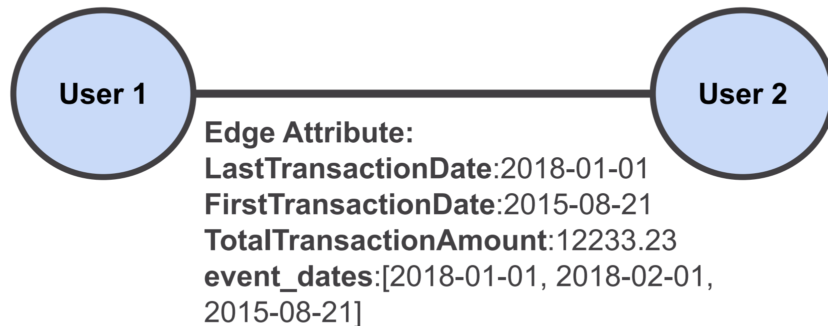
- Create a vertex for each transaction event.
- Connect transactions with the same attributes via attribute vertices.



OR

Method 2: Events aggregated into one Edge

- Connect users who have transactions with a single edge
- Aggregate historical info or use a container to hold a set of values



© 2019 TigerGraph. All Rights Reserved

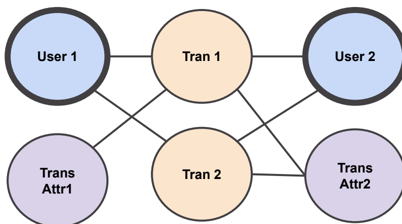
TigerGraph

Multiple Events/Transactions Between Two Entities

- Create vertex for each transaction event.
- Connect transactions with same attribute via attribute vertices.

Pros: Easy to do transaction analytics, such as finding transaction community and similar transactions. Able to do filtering on the transaction vertex attributes.

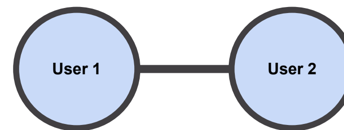
Cons: Uses more memory, takes more steps to traverse between users.



- Connect users who had transactions with a single edge
- Aggregate historical info to edge attributes

Pros: Significantly less memory usage (if without container). Takes fewer steps to traverse between users.

Cons: Searching on transactions is less efficient. Slower update/insert when using a container.



© 2019 TigerGraph. All Rights Reserved

TigerGraph

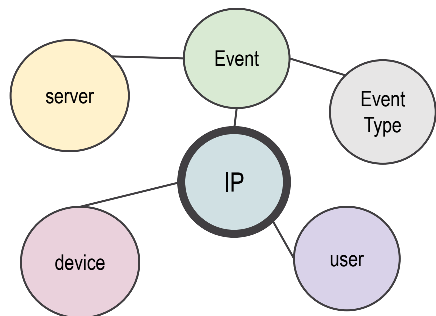
Design Schema Based on Use Case

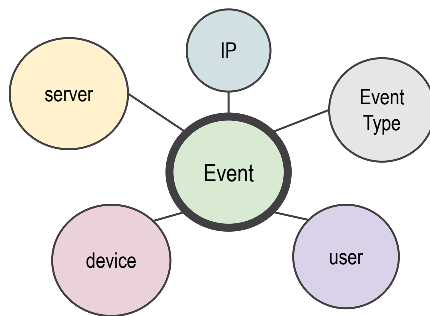
For any given data set, there can be multiple choices for creating a graph schema.

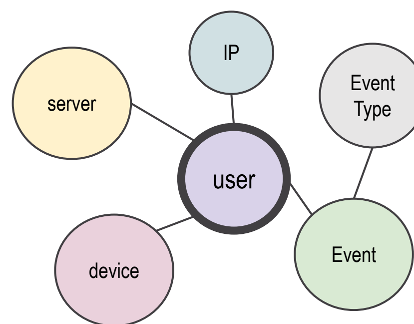
Design the schema that can solve your business problem and provide the best performance.

Event ID	IP	Server	Device	UserId	EventType	Message
001	50.124.11.1	s001	dev001	u001	et1	mmmmmmm
002	50.124.11.2	s002	dev002	u002	et2	mmmmmmm
...	...	...	...	...	...	...

But which one serves **your use case** the best?

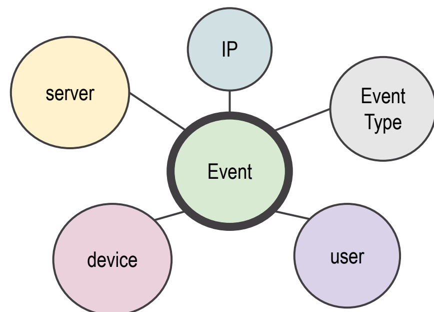






© 2019 TigerGraph. All Rights Reserved

Design Schema Based on Use Case: Two Common Styles



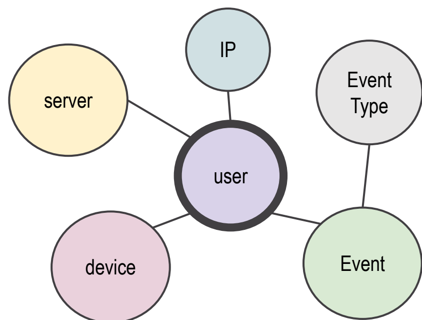
Event-centered schema

Pros: All info of an event is in its 1-hop neighborhood.

Cons: Users are 2 hops away from the device or IP she used

Suitable use cases:

1. Finding communities of events
2. Finding the servers that processed the most events of a given event type
3. Finding the servers visited by a given IP



User-centered schema

Pros: Easy to analyze the connectivities between the users.

Cons: Events are 2 hops away from their related server and IP. It is hard to tell which IP is used for which event.

Suitable use cases:

1. Starting from an input user, detect blacklisted users in k hops.
2. Given a set of blacklisted users, identify the whitelisted users similar to them.
3. Given two input users, are they connected with paths?

© 2019 TigerGraph. All Rights Reserved

TigerGraph

Distributed Native Graph Storage

Distributed Native Graph Storage

“USER123” <---> 1234321

IDS: Bidirectional external ID to Internal ID mapping

1234321, John, 33, john@abc.com
1234322, Tom, 27, tom@abc.com
...

Vertex Partitions: Vertex internal ID and attributes

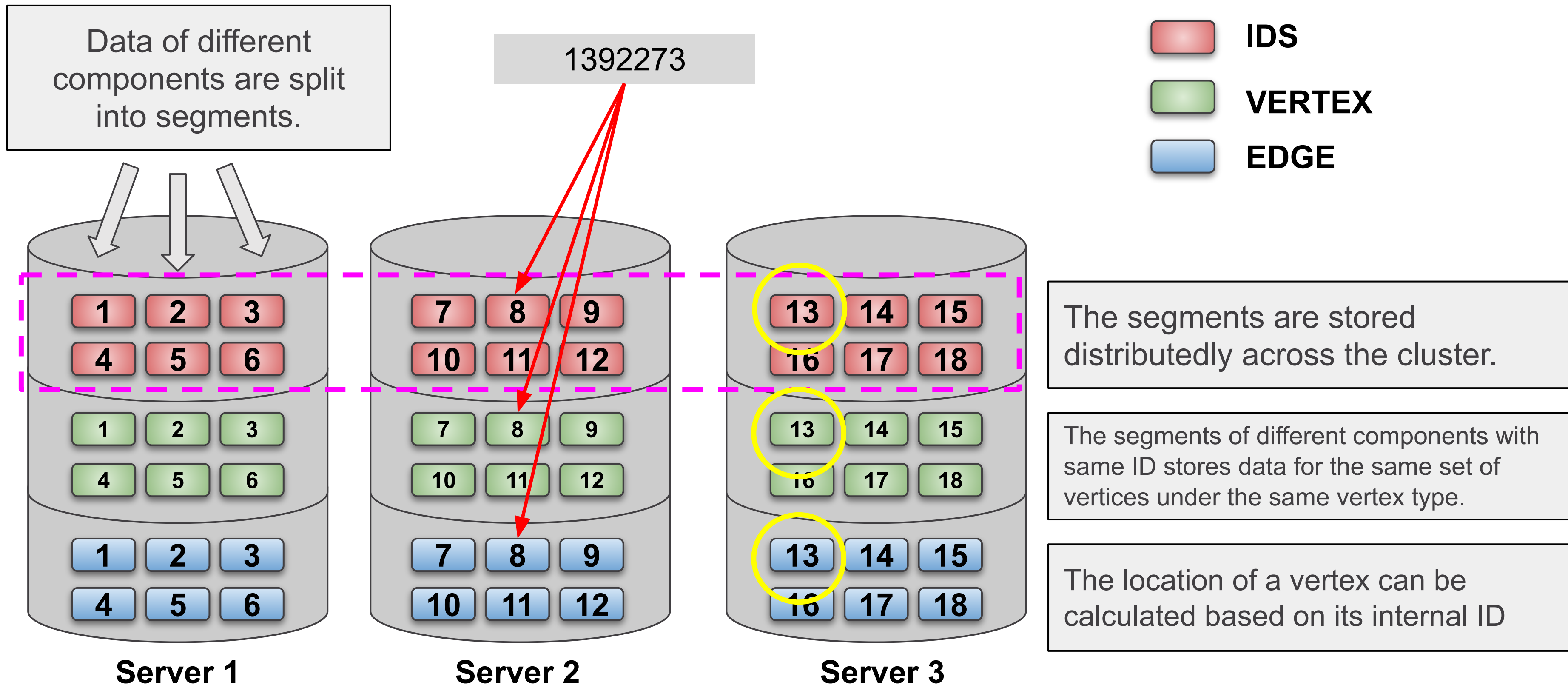
1234321, 1234322, 2020-04-23, 3.3
1234321, 1234324, 2020-02-13, 2.3
...

Edge Partitions: Source vertex internal ID target vertex internal ID, edge attributes

© 2019 TigerGraph. All Rights Reserved

TigerGraph

Distributed Native Graph Storage

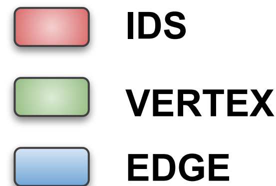


© 2019 TigerGraph. All Rights Reserved

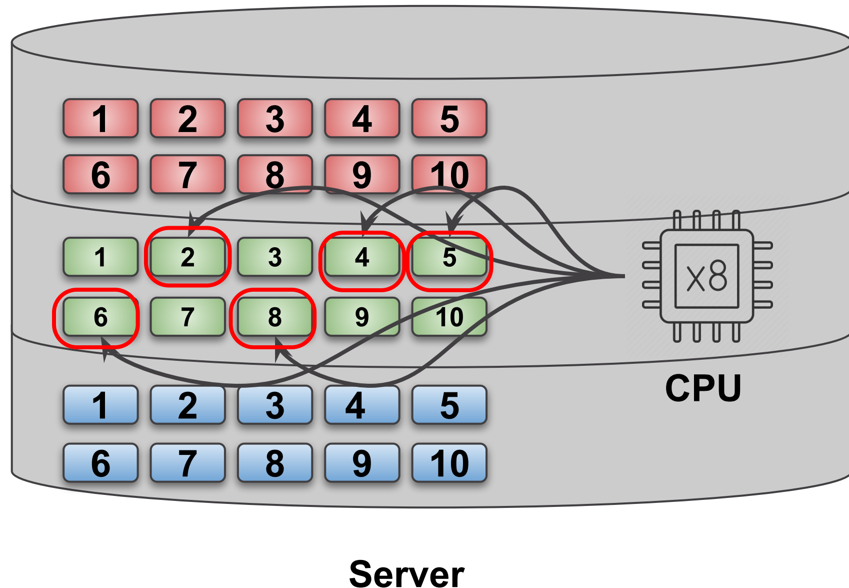
TigerGraph

MPP mechanism of TigerGraph

MPP mechanism of TigerGraph



Processing Vertex-Induced ACCUM/WHERE clause or POST-ACCUM/HAVING clause



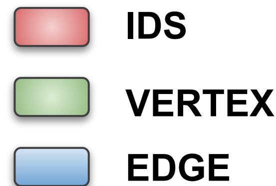

```
user_set = {User.*};
user_set = SELECT s FROM user_set:s
           POST-ACCUM
           …. // some logic;
```

- A thread will be assigned to each vertex segment to perform the logic defined in the **POST-ACCUM** clause in parallel.
- Once the task of one segment is done, the thread move to next unprocessed segment.
- By default, the maximum # of CPU cores of a thread will be assigned.

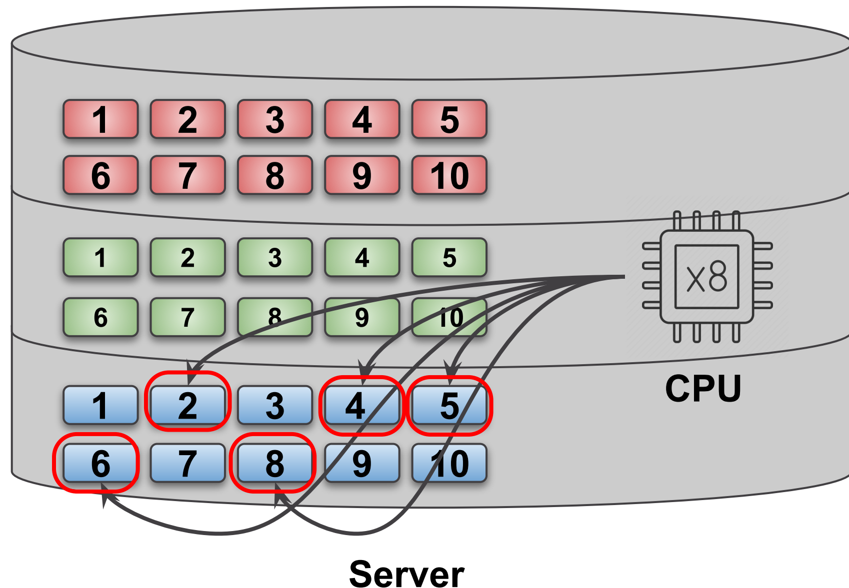
© 2019 TigerGraph. All Rights Reserved

TigerGraph

MPP mechanism of TigerGraph



Processing Edge-Induced WHERE/ACCUM clause




```
user_set = {User.*};
user_set = SELECT s FROM user_set:s-(:e)->:t
           ACCUM
           …. // some logic;
```

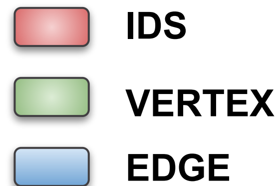
- A thread will be assigned to each edge segment to perform the logic defined in **ACCUM** clause in parallel.
- Once the task of a segment is done, the thread move to next unprocessed segment.
- By default, the maximum # of CPU cores of a thread will be assigned

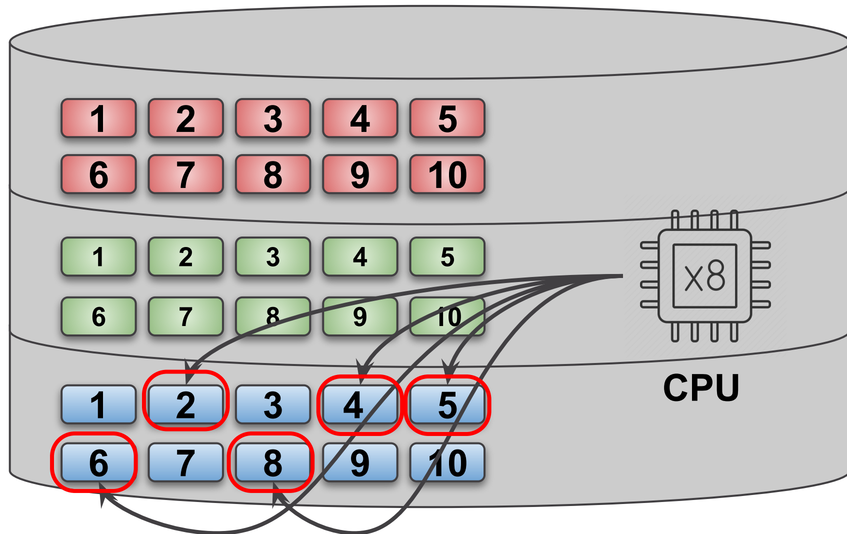
© 2019 TigerGraph. All Rights Reserved

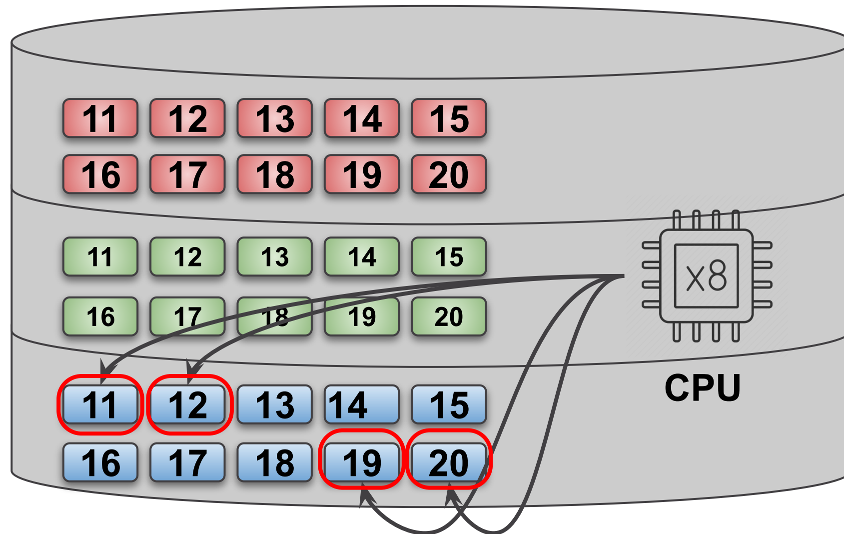
TigerGraph

MPP mechanism of TigerGraph

Processing Edge-Induced WHERE/ACCUM clause distributedly



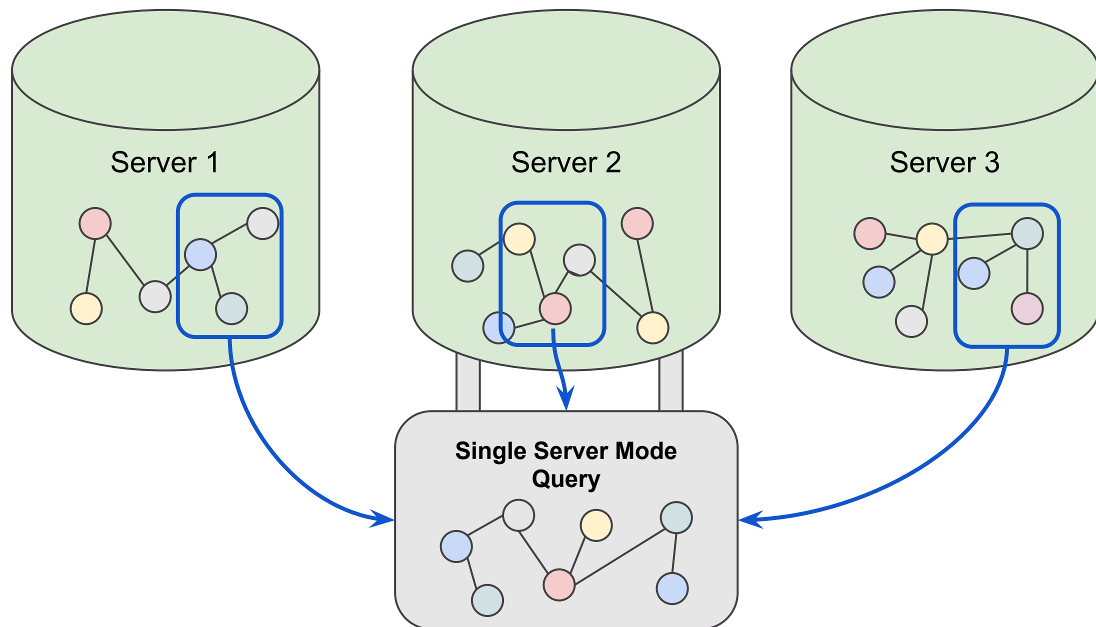




© 2019 TigerGraph. All Rights Reserved

TigerGraph

Single Server mode



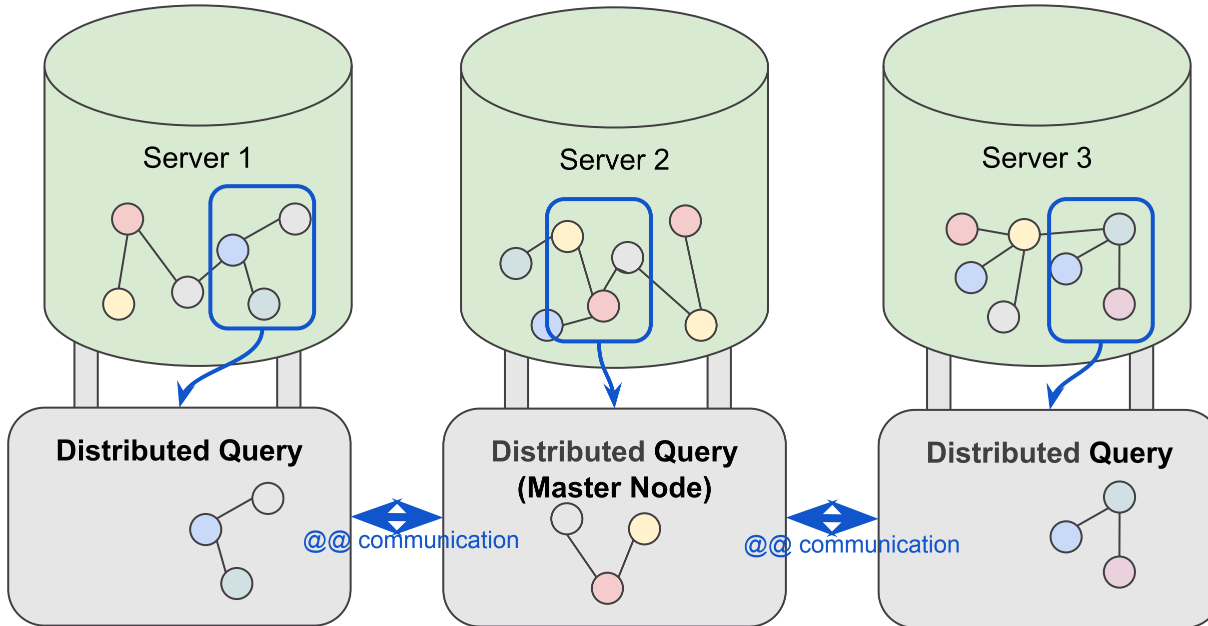
Single Server Mode

- The cluster elects one server to be master for that query.
- All query computation takes place on query master.
- Vertex and edge data are copied to the query master as needed.
- **Best for queries with one or a few starting vertices.**
- **If your query starts from all vertices, don't use this mode.**

© 2019 TigerGraph. All Rights Reserved

TigerGraph

Distributed mode



Distributed Mode

- The server that received the query becomes the master.
- Computation executes on **all** servers in parallel.
- Accumulators are transferred across the cluster.
- **If your query starts from all or most vertices, use this mode.**

© 2019 TigerGraph. All Rights Reserved

TigerGraph

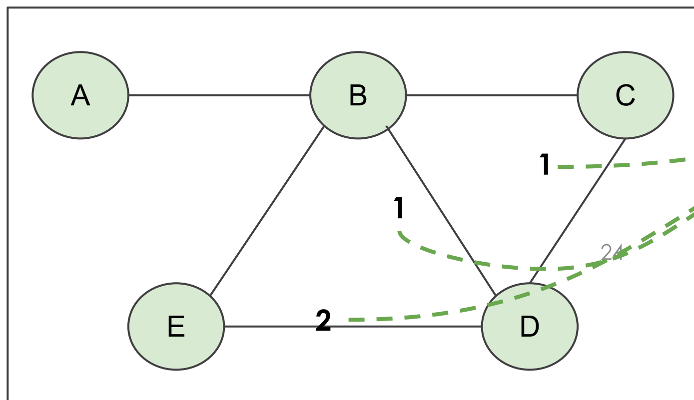
How Does a SELECT Statement Work

Accumulators

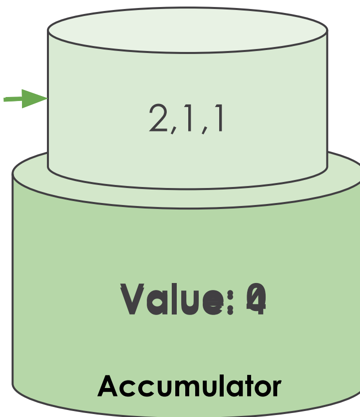
Accumulators are special type of variables that accumulate information about the graph during the traversal.

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.



Graph



© 2019 TigerGraph. All Rights Reserved

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.

© 2019 TigerGraph. All Rights Reserved

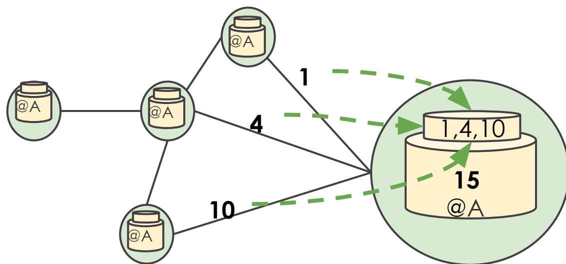
TigerGraph

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.

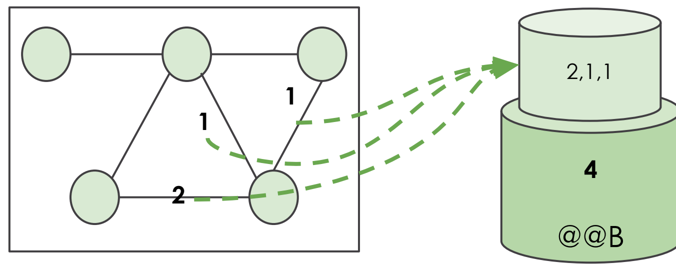
e.x. SumAccum @A;



Global Accumulators:

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

e.x. SumAccum @@B;

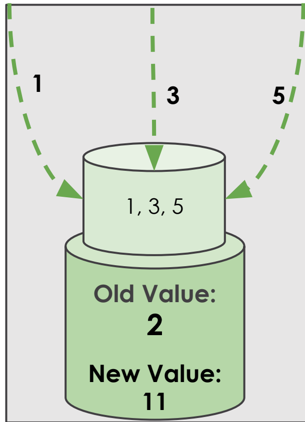


© 2019 TigerGraph. All Rights Reserved

TigerGraph

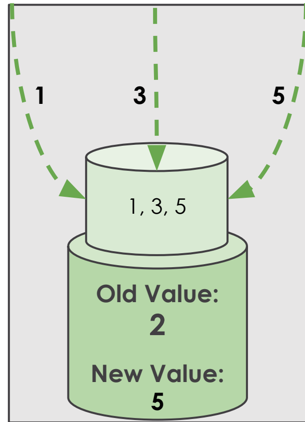
Accumulators

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.**



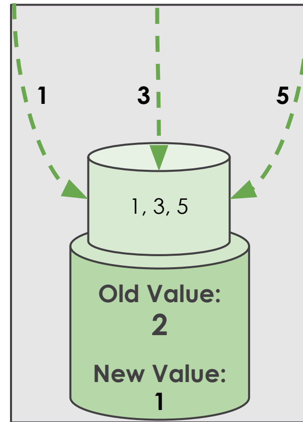
SumAccum<int>

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



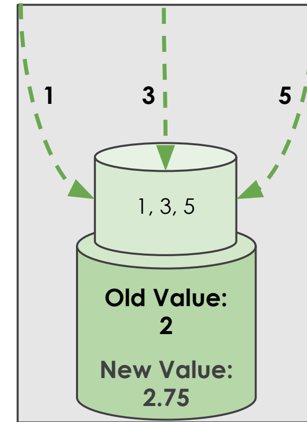
MaxAccum<int>

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

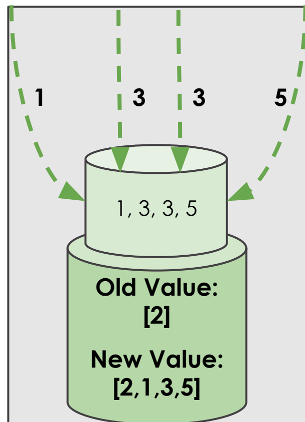
Calculates and stores the cumulative mean of a series of numeric values.

© 2019 TigerGraph. All Rights Reserved

TigerGraph

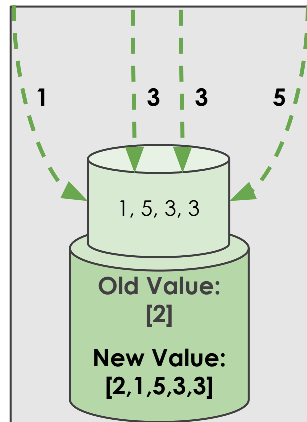
Accumulators

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.**



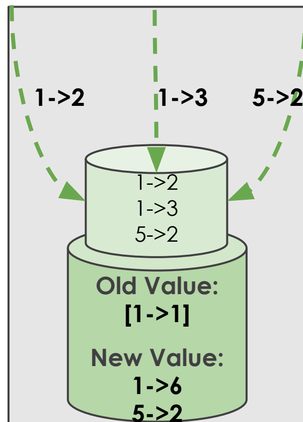
SetAccum<int>

Maintains a collection of unique elements.



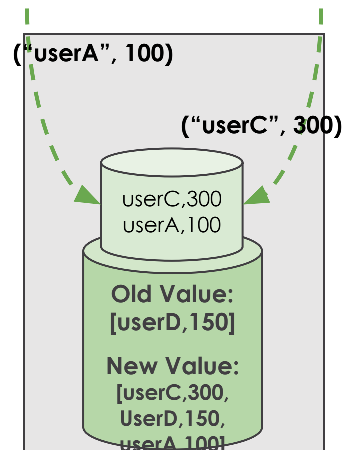
ListAccum<int>

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

© 2019 TigerGraph. All Rights Reserved

TigerGraph

ACCUM Clause

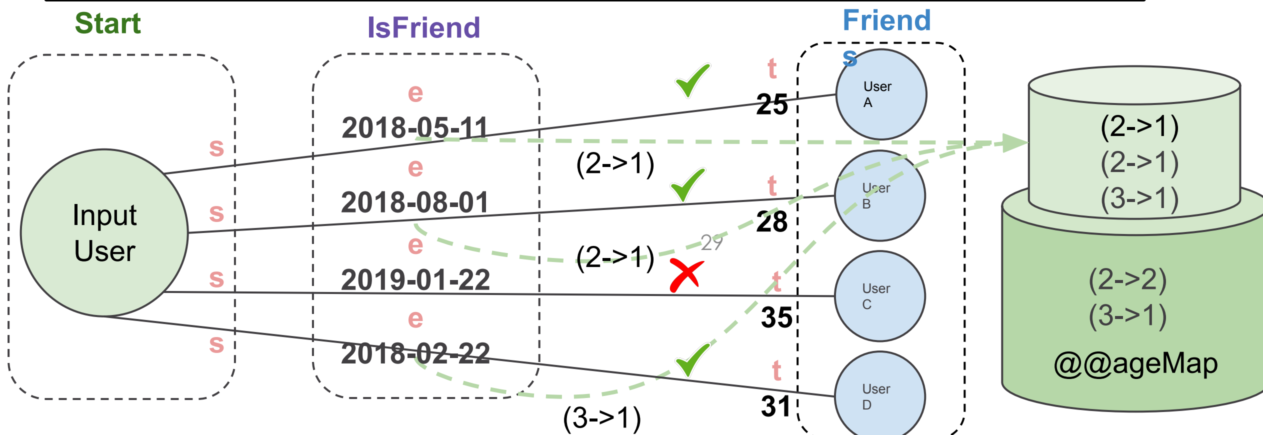
What is the age distribution of friends that were registered in 2018?

```
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;
}
```

WHERE → **Select the matching edges**

ACCUM → **Local compute + send message**

AGGREGATE → **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

© 2019 TigerGraph. All Rights Reserved

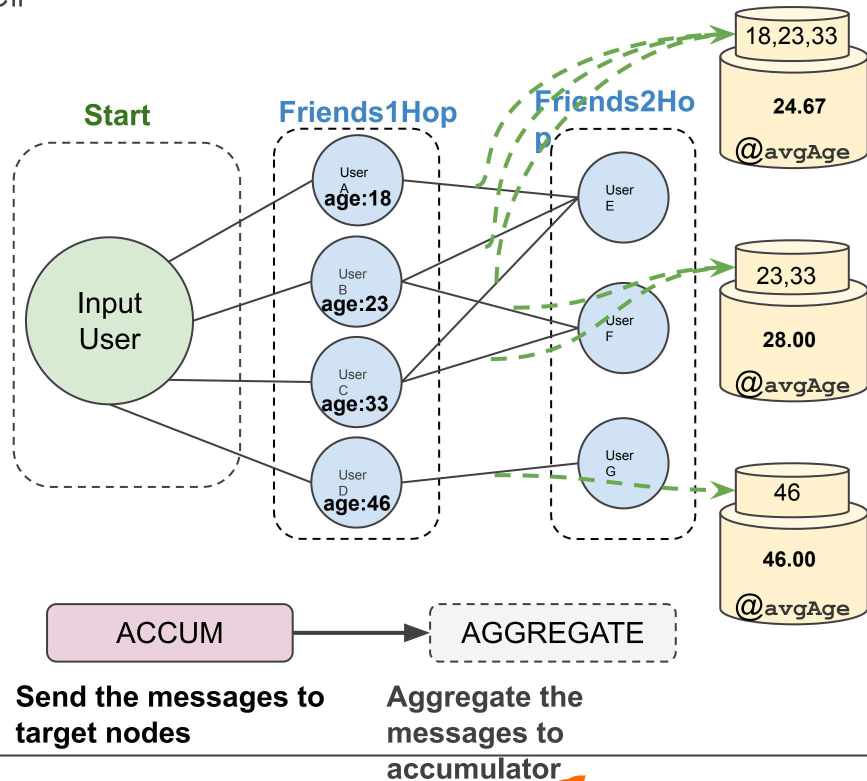
TigerGraph

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.



© 2019 TigerGraph. All Rights Reserved

TigerGraph

Query Writing Best Practices

A Better Traversal Plan

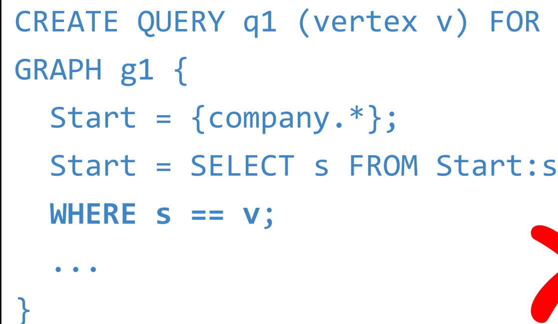
2. Think twice before starting a query with all vertices (of a given type).

Start = {TYPEA.*}

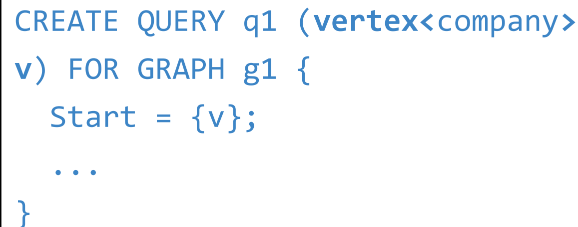
Start = {ANY};

Is it possible to start from a small set of vertex IDs?

Only start from an entire vertex type when you have to.



```
CREATE QUERY q1 (vertex v) FOR
GRAPH g1 {
  Start = {company.*};
  Start = SELECT s FROM Start:s
  WHERE s == v;
  ...
}
```



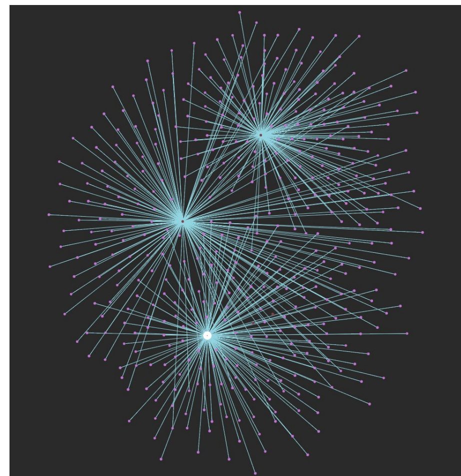
```
CREATE QUERY q1 (vertex<company>
v) FOR GRAPH g1 {
  Start = {v};
  ...
}
```

© 2019 TigerGraph. All Rights Reserved

A Better Traversal Plan

4. Avoid hub nodes

Hub Nodes or **Super Nodes** are vertices having a huge number of neighbors. When traversal encounters such nodes it has to touch a very large portion of the graph, which hinders the query.



Design the traversal plan to avoid starting from the hub nodes.

© 2019 TigerGraph. All Rights Reserved

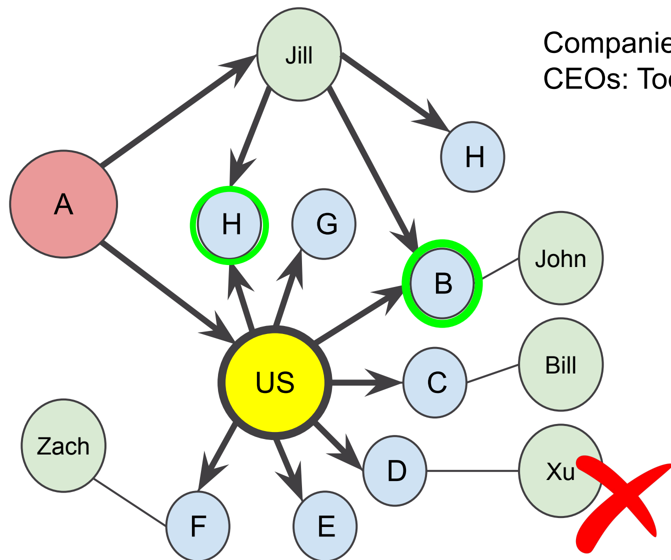
TigerGraph

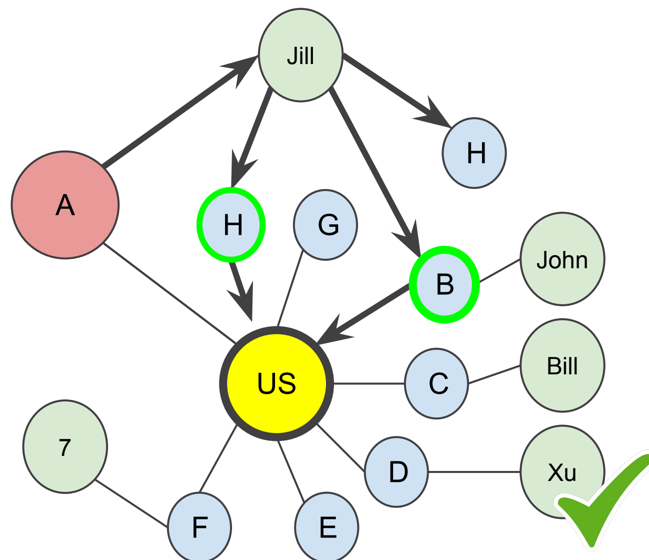
A Better Traversal Plan

4. Avoid hub nodes

Example: Given a company **A**, find all companies that are in the same country and were ran by the same CEO

Companies: A,B,C,...
CEOs: Todd, John, Bill...





© 2019 TigerGraph. All Rights Reserved

TigerGraph

A Better Traversal Plan

4. Avoid hub nodes

Alternatively, when an approximated result is good enough, you can also consider filtering the hub nodes out in your WHERE clause. Or use the SAMPLE clause to sample a subset of the neighbors.

WHERE t.outdegree() < 100000

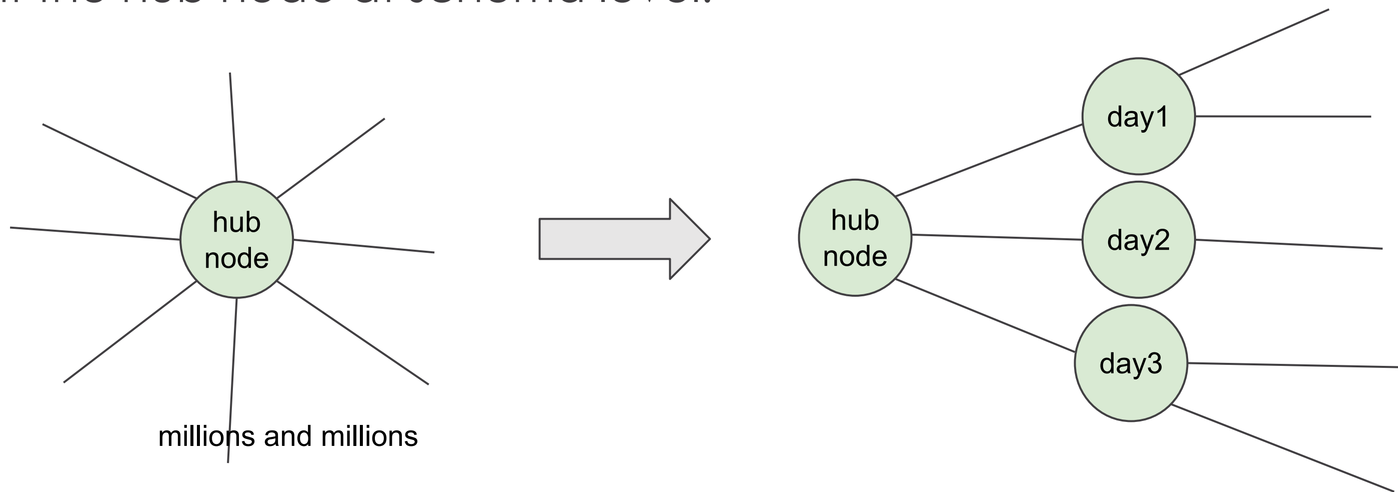
SAMPLE 100 **EDGE WHEN** s.outdegree() > 1000000

© 2019 TigerGraph. All Rights Reserved

A Better Traversal Plan

4. Avoid hub nodes

Split the hub node at schema level.



© 2019 TigerGraph. All Rights Reserved

TigerGraph

Techniques to optimize memory usage

2. Split the query load into batches

When is this needed?

1. The query involves all or a significant fraction of all vertices.
2. Each hop accumulators a significant amount of data in every local accumulator or in global accumulator(s).

The memory usage issue can be mitigated by splitting the query into batches.

© 2019 TigerGraph. All Rights Reserved

TigerGraph

2. Split the query load into batches

```
Foreach company, find majority type of the subsidiaries.
// split the calculation into k batches
CREATE QUERY split (int k) FOR GRAPH exampleGraph {
  MapAccum<string,int> @map;
  SumAccum<string> @majorityType;
  // define local and global accumulators
  comps = {Company.*};
  FOREACH i IN range [0, k-1] DO
        tmp = SELECT s FROM comps:s
                WHERE getvid(s) % k == i;
        tmp = select s from tmp:s-(subsidiary:e)-:t
              accum s.@map += (t.comp_type, 1)
              Post-accum foreach (k,v) in s.@map do
                          …
                         End,
                         s.@map.clear() ...
  END;
}
```

- k is the number of batches that the job will be split into
- Splitting the query into batches increases the execution time, but decreases peak memory usage.

© 2019 TigerGraph. All Rights Reserved

Parallelization

1. Run queries in parallel

```
CREATE QUERY sequentialExample () FOR GRAPH exampleGraph {
  SetAccum<VERTEX> @@verSet;
  Start = {TYPEA.*};
  Start = SELECT s FROM Start:s ACCUM @@verSet +=s;
  FOREACH v in @@verSet DO
     Start = {v};
     Start = SELECT s FROM Start.... # Nested query
  End;
}
```

- FOREACH executes sequentially for each vertex.
- Sequential execution is slower and does not make full use of available CPU resources.

© 2019 TigerGraph. All Rights Reserved

TigerGraph

Parallelization

1. Run queries in parallel

```
CREATE QUERY subQuery (VERTEX input) FOR GRAPH exampleGraph RETURNS …{
  Start = {input};
  Start = SELECT s FROM Start:….;
}

CREATE QUERY parallelExample () FOR GRAPH exampleGraph {
  Start = {TYPEA.*};
  Start = SELECT s FROM Start:s ACCUM subQuery(s);
}
```

- ACCUM executes multiple threads concurrently, so subqueries run in parallel.
- For some use cases, ACCUM and/or subquery can simplify the logic or data structures.

© 2019 TigerGraph. All Rights Reserved

TigerGraph

Data Structure Optimization

1. Use fewer container operations

// given an input vertex set **forbiddenSet**, skip those vertices during the traversal

```
CREATE QUERY example1 (set<vertex> forbiddenSet) FOR GRAPH exampleGraph {
    …
    Start = {ANY};
    Start = SELECT t FROM Start:s-(:e)->:t
            WHERE t NOT IN forbiddenSet; // if alias t is in forbiddenSet, skip this edge
   …
}
```

- Each edge will perform the same set operation to check if vertex **t** is in **forbiddenSet** or not
- Set operation of existence checking is slow

© 2019 TigerGraph. All Rights Reserved

TigerGraph

Data Structure Optimization

1. Use fewer container operations

// given an input vertex set **forbiddenSet**, skip those vertices during the traversal

```
CREATE QUERY example2 (set<vertex> forbiddenSet) FOR GRAPH exampleGraph {
    OrAccum<bool> @isFob;
    Forbid = forbiddenSet;
    Forbid = SELECT s FROM Forbid:s POST_ACCUM s.@isFob = true;
   …
    Start = {ALL};
    Start = SELECT t FROM Start:s-(:e)-> :t
                where t.@isFob == false;
  …
}
```

- Create an OrAccum to mark the forbidden set first
- In each WHERE clause only a boolean check is executed

© 2019 TigerGraph. All Rights Reserved

TigerGraph

Data Structure Optimization

2. Avoid using container type vertex-attached accumulators

```
// This query prints all the shortest paths between two input vertices
CREATE QUERY example (VERTEX input1, VERTEX input2) FOR GRAPH exampleGraph {
  SetAccum<EDGE> @path;
  OrAccum<Bool> @@found;
  Start = {input1};
  WHILE Start.size() > 0 AND @@found == false DO
    Start = SELECT t FROM Start-(directedEdge:e)-> :t
            WHERE t.@path.size() == 0
            ACCUM t.@path += s.@path, t.@path += e
            POST-ACCUM CASE WHEN t==input2 THEN
                       @@found = true END;
  END;
  PRINT Start;
}
```

- Start from input1, accumulate the edges along the path
- Continue doing above till input2 is found

© 2019 TigerGraph. All Rights Reserved

TigerGraph

Data Structure Optimization

```
// This query prints all the shortest paths between two input vertices
CREATE QUERY example (VERTEX input1, VERTEX input2) FOR GRAPH exampleGraph {
  MaxAccum<INT> @dist;
  OrAccum<Bool> @@found1, @@found2;
  ListAccum<EDGE> @@resultPath;
  Start = {input1};
  // mark the vertices along the path with distance from input 1
  While Start.size() > 0 and @@found1 == false do
    Start = SELECT t FROM Start-(directedEdge:e)-> :t
            WHERE t.@dist < 0
            ACCUM t.@dist += s.@dist + 1
            POST_ACCUM CASE WHEN t==input2 THEN @@found1 = true END;
  END;
  Start = {input2};
  // store the vertices along the path in the result
  While Start.size() > 0 and @@found2 == false do
    Start = SELECT t FROM Start-(ReverseDirectedEdge:e)-> :t
            WHERE t.@dist == s.@dist - 1
            ACCUM @@resultPath += e
            POST_ACCUM CASE WHEN t==input1 THEN @@found2 = true END;
  END;
  PRINT @@resultPath;
}
```

- Use MaxAccum<INT> instead of SetAccum<EDGE>
- Mark each vertex traversed with the distance from input1, until input2 is found
- When input2 is found, start from input2 and traverse in the reverse direction. If a vertex having a distance equal to the distance of from the vertex minus one, then it must be on the path.

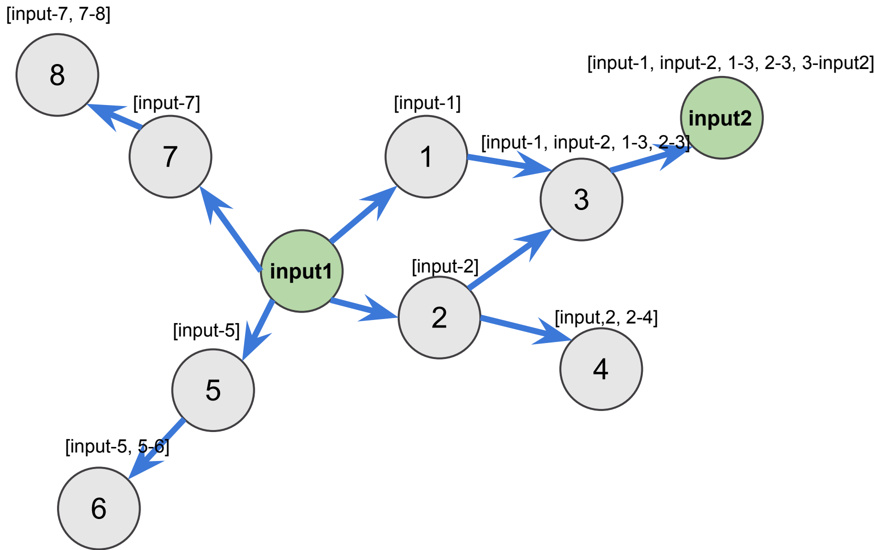
© 2019 TigerGraph. All Rights Reserved

TigerGraph

Data Structure Optimization

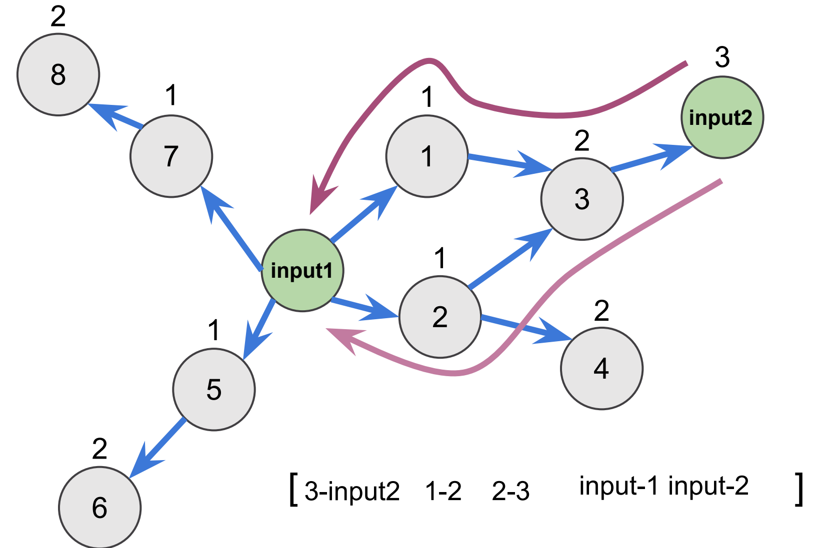
2. Avoid using container type vertex-attached accumulators

Solution With SetAccum<EDGE>



- Every vertex traversed carries the path starting from **input1**
- SetAccum<EDGE> @path is memory consuming
- SetAccum += operation is expensive

Solution With MaxAccum<INT>



- Every vertex carries the distance to **input1**
- MaxAccum<INT> @dist is more memory efficient
- Even when traversal distance is longer, this solution is faster in most cases

Q&A