



Query Writing Best Practices

Query Writing Best Practices

1. Overview
2. Data Structure Optimization
3. A Better Traversal Plan
4. Parallelization
5. Memory Saving
6. Preprocessing
7. Miscellaneous

Overview

GSQL gives users unlimited capability to implement their complex business logic. With so much capability, users can sometimes make non-optimal choices.

Drawing from the experience of tuning hundreds of queries and applications, this best practice guide shows ways of improving query performance and reducing memory usage. These practices make the TigerGraph system utilize 100% of its power.

Data Structure Optimization

1. Use fewer container operations **(8 out of 10 for impact)**
2. Use fewer vertex-attached container types **(10/10)**
3. ListAccum is faster than SetAccum **(3/10)**
4. Use bit vector for condition matching **(6/10)**
5. Use fewer Strings **(9/10)**



Data Structure Optimization

1. Use fewer container operations

Container operations are more expensive.

Containers (Lists, Arrays, Sets, Bags, Maps, Heaps, and accumulators of these types) are more complex than scalar elements. While the GSQL operations may look simple, the complex is still there, at the machine level.

Container operations such as populating a Map with key-value pairs, checking existence in a Set takes more time than other non-container operations.

Each statement in the ACCUM and POST-ACCUM clauses is executed for each selected edge or vertex, so a container operation here could be executed up to millions of times. Therefore, **avoiding the use of those operations can significantly improve the query performance.**

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

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

Data Structure Optimization

2. Avoid using container type vertex-attached accumulators

Container type vertex-attached accumulators are expensive, even when they are empty, due to memory allocation. You can observe the query slowing down by just defining them, since the data structure will be created on all vertices involved in the query traversal.

Container type accumulators include: ListAccum, SetAccum, BagAccum, MapAccum, ArrayAccum, GroupByAccum, HeapAccum

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

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;
```

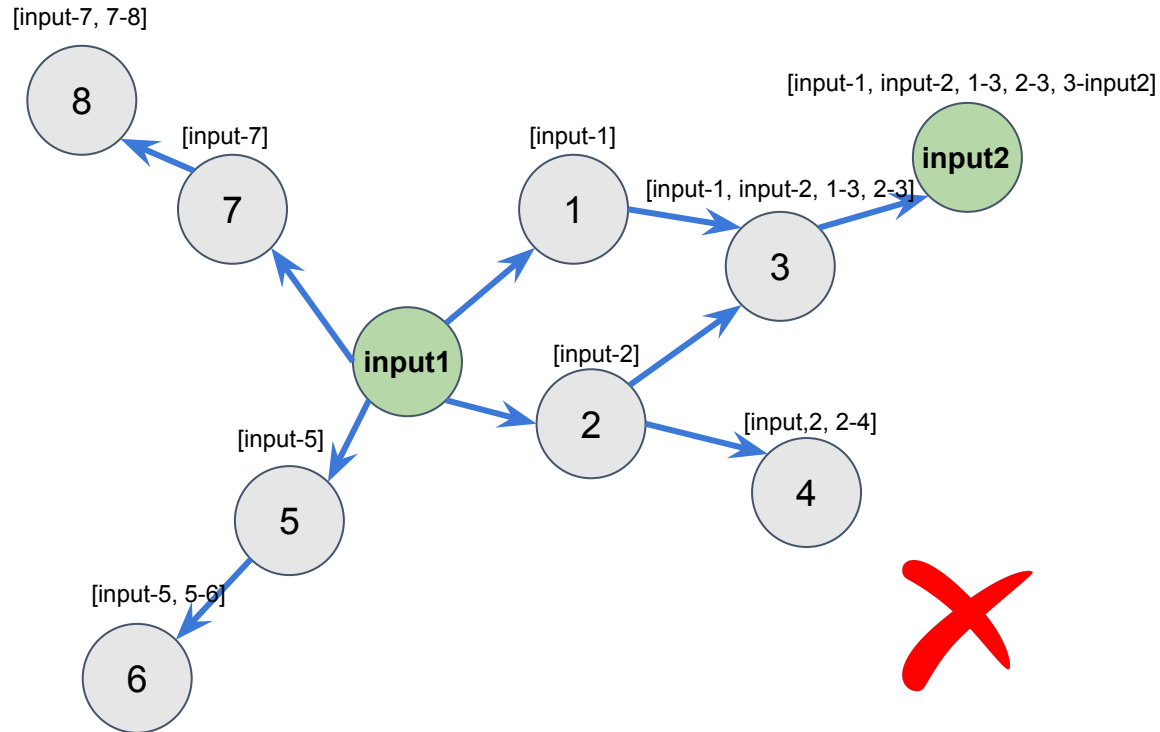


- Use MaxAccum<INT> instead of SetAccum<EDGE>
- Mark each vertex traversed with the distance from input1, until input2 is found
- When input2 is found, start from input2 and traverse in the reverse direction. If a vertex having a distance equal to the distance of from the vertex minus one, then it must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

Data Structure Optimization

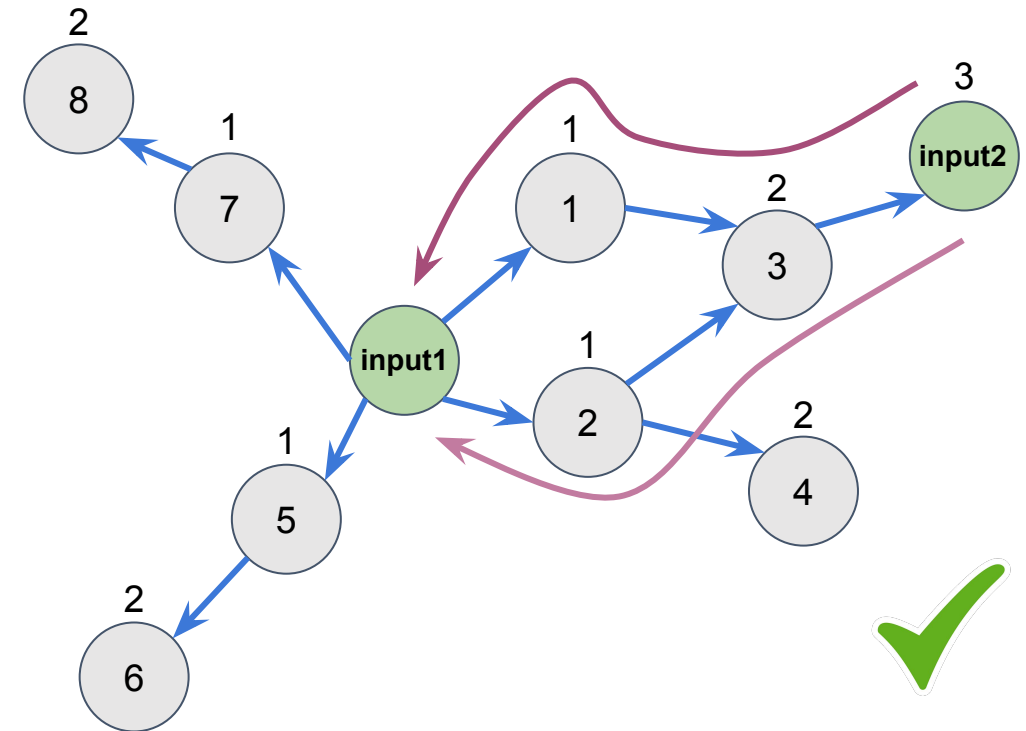
2. Avoid using container type vertex-attached accumulators

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

Data Structure Optimization

3. ListAccum is faster than SetAccum

ListAccum is faster for insertion

SetAccum is a little faster for lookups, but slower for insertion

For the case above, in the solution with MaxAccum<INT>, it used **ListAccum<EDGE>** to store the result instead of **SetAccum<EDGE>**. Since there is no duplication in the collected edges, using ListAccum can provide better performance.

Data Structure Optimization

4. Use bit vector for condition matching.

Example 1: Find companies that are controlled by all input companies.

```
CREATE QUERY example (set<vertex<Enterprise>> set)
FOR GRAPH exampleGraph {

  SetAccum<VERTEX> @set, @@result;

  start = {set};
  start = SELECT s FROM start:s ACCUM s.@set += s;

  WHILE start.size() > 0 LIMIT 30 DO
    start = SELECT t FROM start-(reverse_controlled_by:e)-:t
      ACCUM t.@set += s.@set
    POST-ACCUM
      CASE WHEN s.@set == set.size() THEN
        @@result += t
      END;
  END;

  PRINT @@result;
}
```



```
CREATE QUERY example (set<vertex<Enterprise>> set)
FOR GRAPH exampleGraph {

  MapAccum<vertex, uint> @@bitMap;
  SetAccum<VERTEX> @@result;
  OrAccum<uint> @bit;
  int i = 1;
  FOREACH v IN set DO
    @@bitMap += (v -> i);
    i = i * 2;
  END;

  start = {set};
  start = SELECT s FROM start:s ACCUM s.@bit += @@bitMap.get(s);
  WHILE start.size() > 0 LIMIT 30 DO
    start = SELECT t FROM start-(reverse_controlled_by:e)-:t
      ACCUM t.@bit += s.@bit
    POST-ACCUM
      CASE WHEN s.@bit == pow(2, set.size())-1 THEN
        @@result += t
      END;
  END;

  PRINT @@result;
}
```



Data Structure Optimization

5. Use fewer Strings

String is a lot more expensive than other primitive types. If possible, integer or vertex type is preferred over string type. Since vertex type basically is a 64 bit integer.

performance: vertex > string > container of vertex > container of string

For example: find a set of companies and store them in a result ListAccum

```
CREATE QUERY example (set<vertex<Enterprise>> set)
FOR GRAPH exampleGraph {

  SetAccum<string> @@list;
  ...
  v_set = SELECT t FROM start-(:e)-:t
  ...
  POST-ACCUM @@list += t.company_name
  ...
  PRINT @@list;
}
```



```
CREATE QUERY example (set<vertex<Enterprise>> set)
FOR GRAPH exampleGraph {

  SetAccum<vertex> @@list;
  ...
  v_set = SELECT t FROM start-(:e)-:t
  ...
  POST-ACCUM @@list += t
  ...
  PRINT @@list;
}
```



A Better Traversal Plan

1. Design the lightest weight traversal path **(7/10)**
2. Think twice before starting a query with all vertices (of a given type). **(9/10)**
3. Make the algorithm bidirectional **(6/10)**
4. Avoid hub nodes, do the moonwalk **(8/10)**
5. Topology Sort **(-/10)**



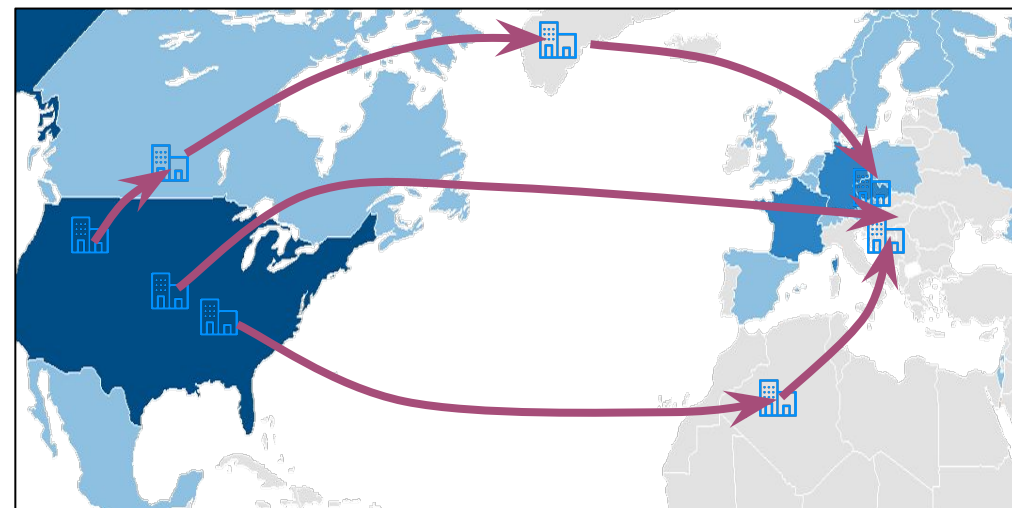
A Better Traversal Plan

1. Design the lightest weight traversal plan

Similar to relational DB query optimization, start with smaller sets, and prune your sets as early as possible.

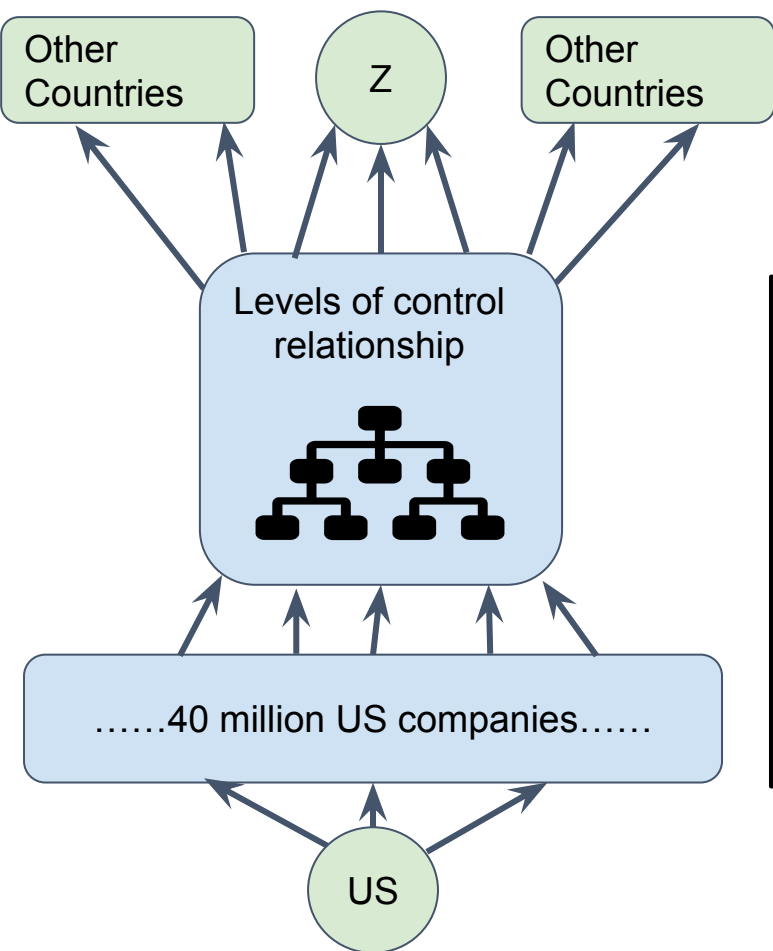
Example: Find the US domiciled companies that have ultimate parent company in country Z. Z has fewer companies than the US.

By starting with the smaller set (Z instead of US), you can reduce the amount of computation.

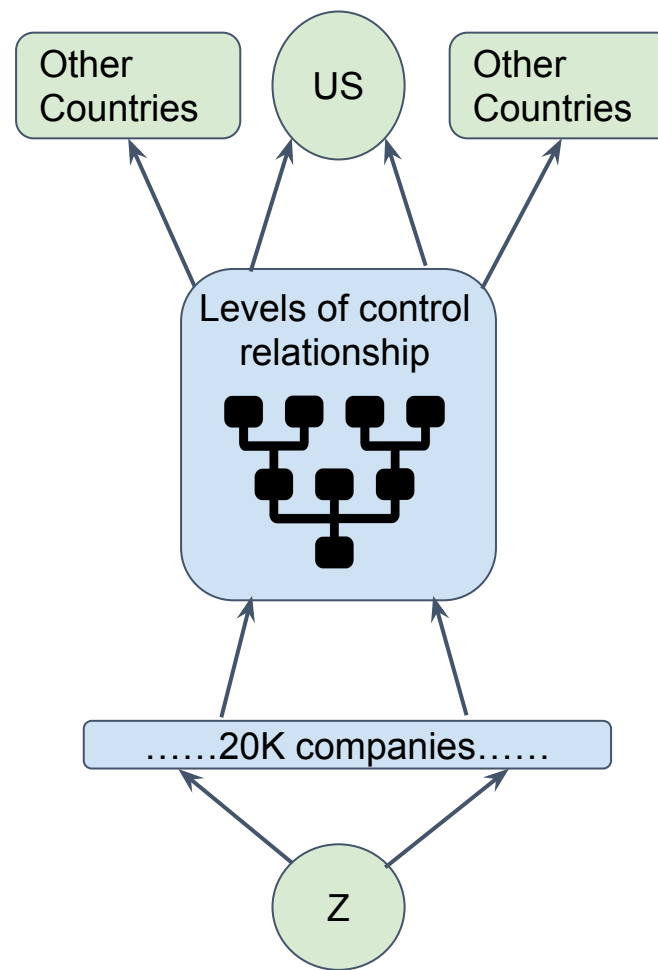


A Better Traversal Plan

1. Design the lightest weight traversal plan



- Starting from US requires finding ultimate parent companies for all 40 million US companies.
- The total number of edges traversed is in the 100s of millions



- Starting from Z, we only need to start with about 20K companies
- The total number of edges traversed is below 1 million

A Better Traversal Plan

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

Start = {TYPEA.*}

Start = {*.}*};

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 () FOR GRAPH g1 {  
  Start = {cities.*};  
  Start = SELECT s FROM Start:s  
  WHERE s.cityName == "Palo Alto";  
  ...  
}
```



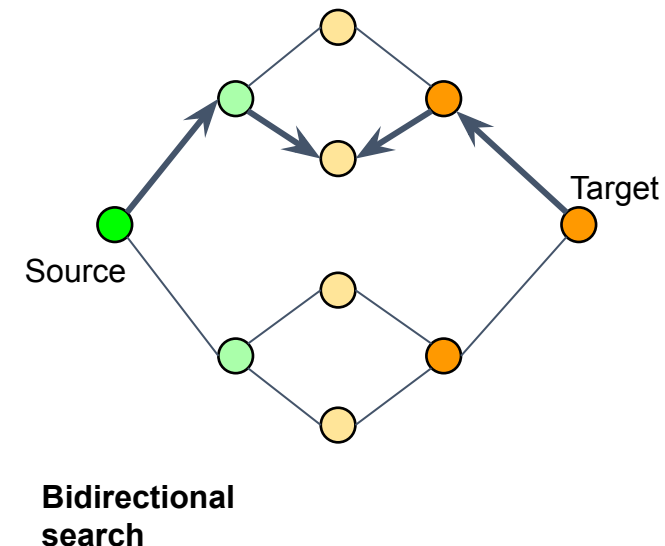
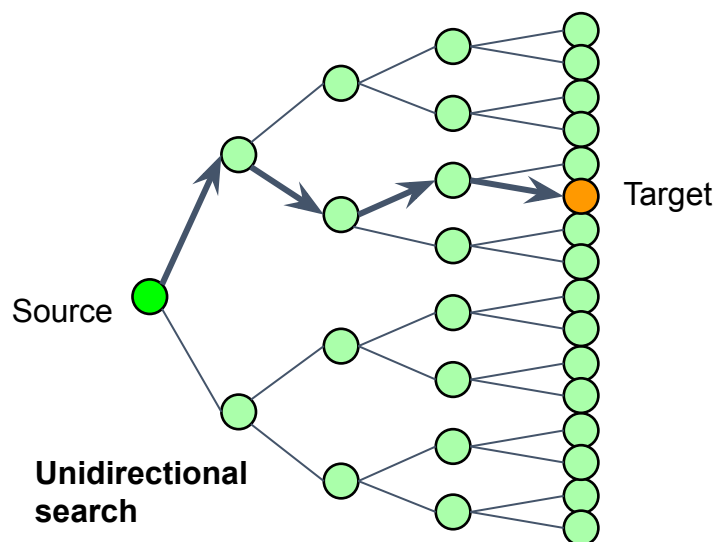
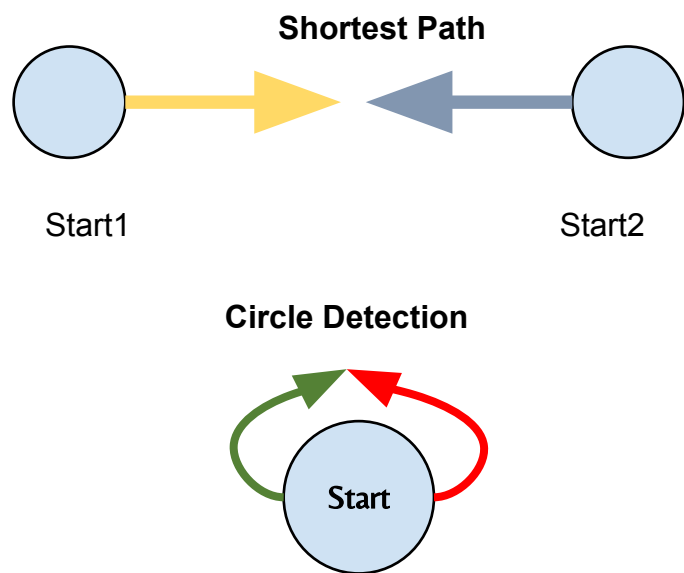
```
CREATE QUERY q1 (vertex<cities> v)  
FOR GRAPH g1 {  
  Start = {v};  
  ...  
}
```



A Better Traversal Plan

3. Start the traversal bidirectionally

When trying to find a path, it is much faster to do the traversal bidirectionally. For example: shortest path query and circle detection query.



Why? Because the number of edges traversed is reduced exponentially.

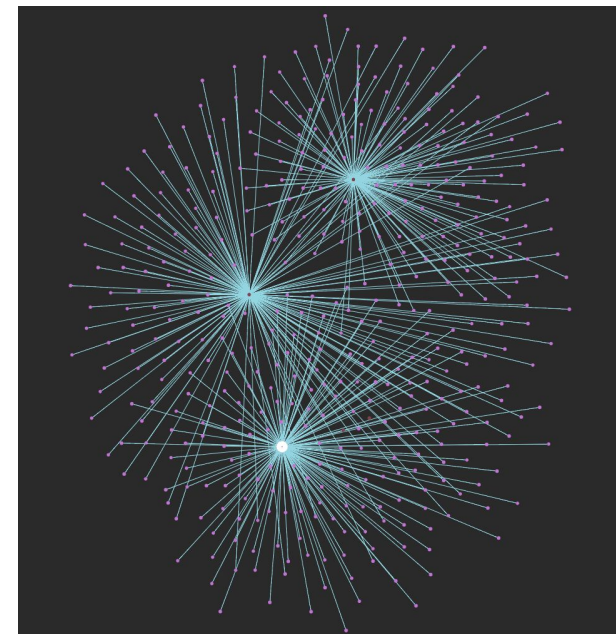
Suppose each vertex has an average of 10 edges, and shortest path from Source to Target turns out to be 4 hops. With a unidirectional search, we will traverse $10 \times 10 \times 10 \times 10 = 10,000$ edges. With bidirectional search, we will traverse only $10 \times 10 + 10 \times 10 = 200$ edges.

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 makes the query very slow.

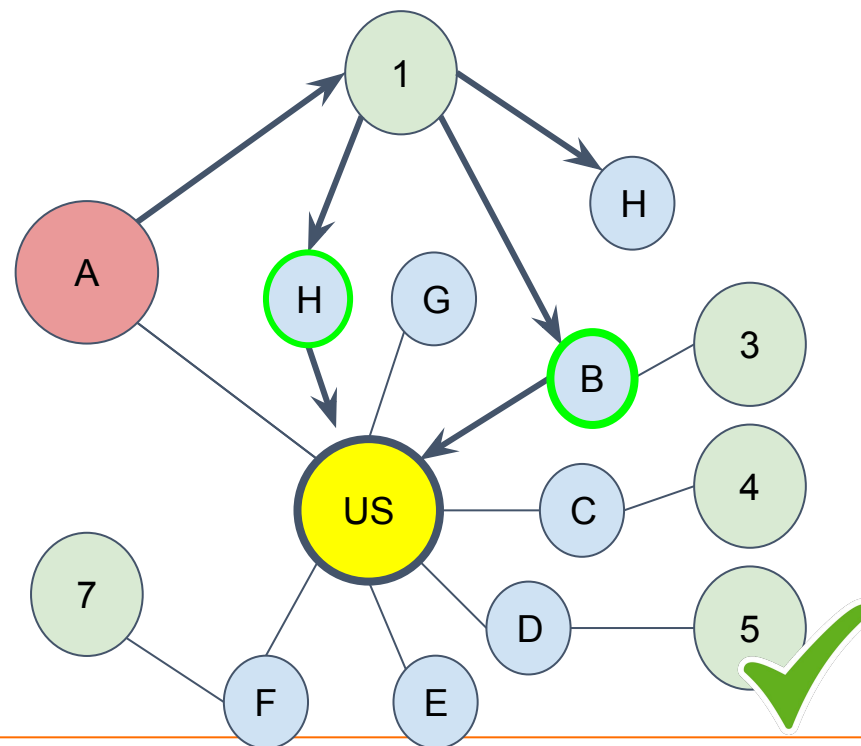
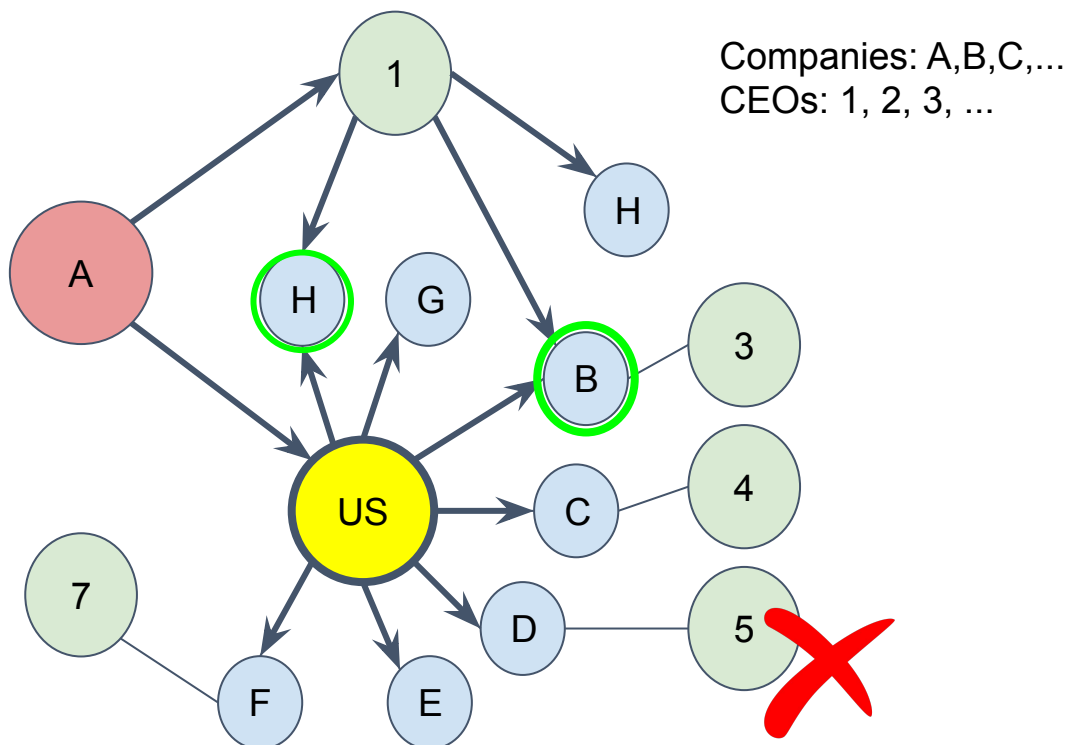
Design the traversal plan to avoid starting from the hub nodes.



A Better Traversal Plan

4. Avoid hub nodes

Example: Given a company **A**, find all companies that are in the same country and share a CEO.




A Better Traversal Plan


4. Avoid hub nodes

Example: Given a company **A**, find all companies that are in the same country and share a CEO.

```
Start = {inputCompany};  
  
Country = SELECT t FROM Start-(DomiciledIn)->t;  
  
CEO = SELECT t FROM Start-(HasCEO)->t;  
  
Comp1 = SELECT t FROM  
Country-(reverse_DomiciledIn)->t;  
  
Comp2 = SELECT t FROM  
CEO-(reverse_HasCEO)->t;  
  
Result = Comp1 INTERSECT Comp2;  
  
PRINT Result;
```



```
OrAccum @isTargetCountry;  
  
Start = {inputCompany};  
  
Country = SELECT t FROM Start-(DomiciledIn)->t  
          POST-ACCUM t.@isTargetCountry = true;  
  
CEO = SELECT t FROM Start-(HasCEO)->t;  
  
Comp = SELECT t FROM CEO-(reverse_HasCEO)->t;  
  
Result = SELECT s FROM Comp:s-(DomiciledIn)->t  
          WHERE t.@isTargetCountry == true;  
  
PRINT Result;
```



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

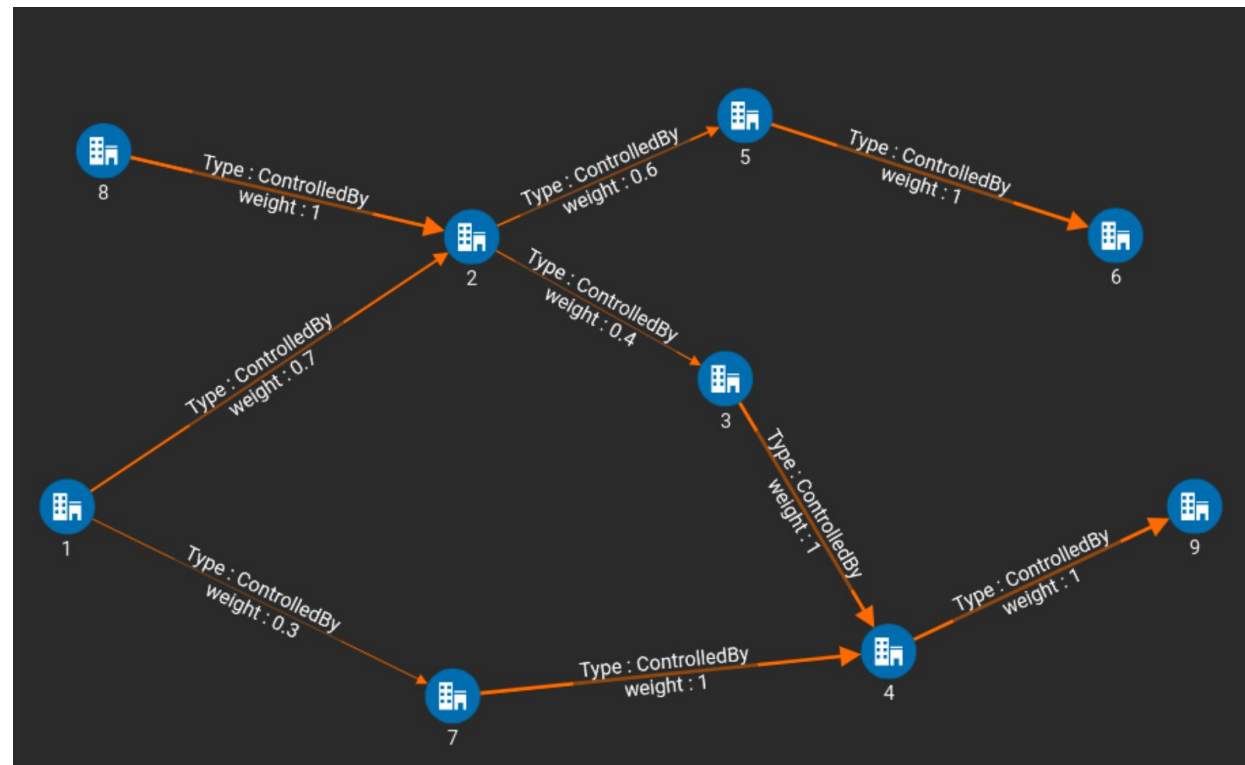
A Better Traversal Plan

5. Topology Sort

By using accumulators, control the visiting order of the traversal to reduce the amount of computation or guarantee the valid sequence of the task.

Example:

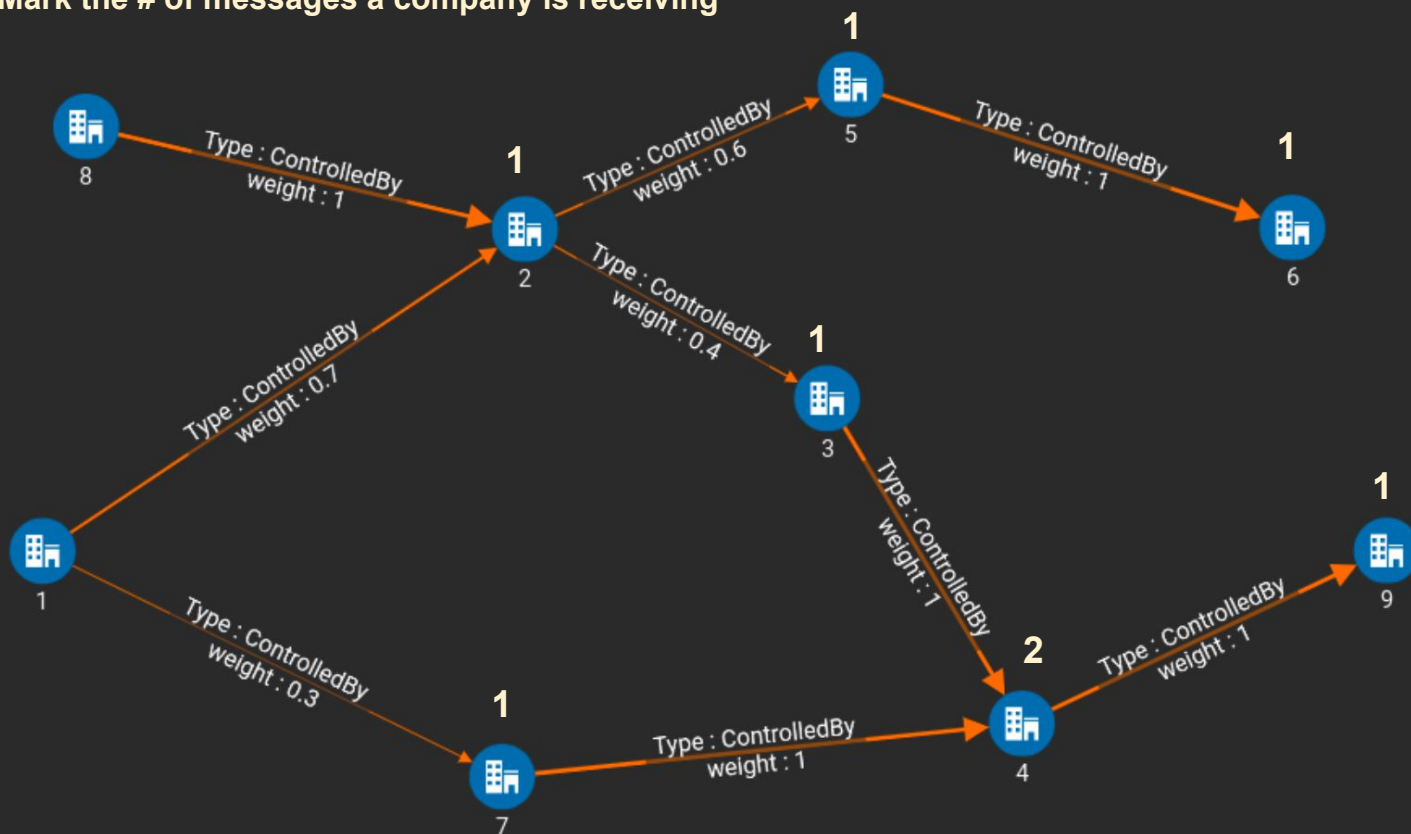
Given a company, find the ultimate parent companies that have the largest portion of ownership.



A Better Traversal Plan

5. Topology Sort

Mark the # of messages a company is receiving



```
CREATE QUERY getOwnershipPert (vertex<Company> inputComp) FOR GRAPH MyGraph
{
    SumAccum<int> @msgCnt1, @msgCnt2;
    OrAccum<bool> @visited;
    SumAccum<float> @score = 0;
    SetAccum<vertex> @@results;

    Start = {inputComp};

    WHILE Start.size() > 0 limit 8 DO
        Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
            ACCUM t.@msgCnt1 += 1
            POST-ACCUM s.@visited = true
            HAVING t.@visited == false; // don't start again for the
second visit;
        end;

        Start = {inputComp};

        Start = SELECT s FROM Start:s ACCUM s.@score = 1; // initialize @score

        WHILE Start.size() > 0 LIMIT 8 DO
            Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
                ACCUM t.@msgCnt2 += 1, t.@score += s.@score * e.weight
                POST-ACCUM
                CASE WHEN t.outdegree("ControlledBy") == 0 THEN
                    @@results += t
                END
                HAVING t.@msgCnt2 == t.@msgCnt1; // make sure got all the
scores
            END;

            Start = @@results;

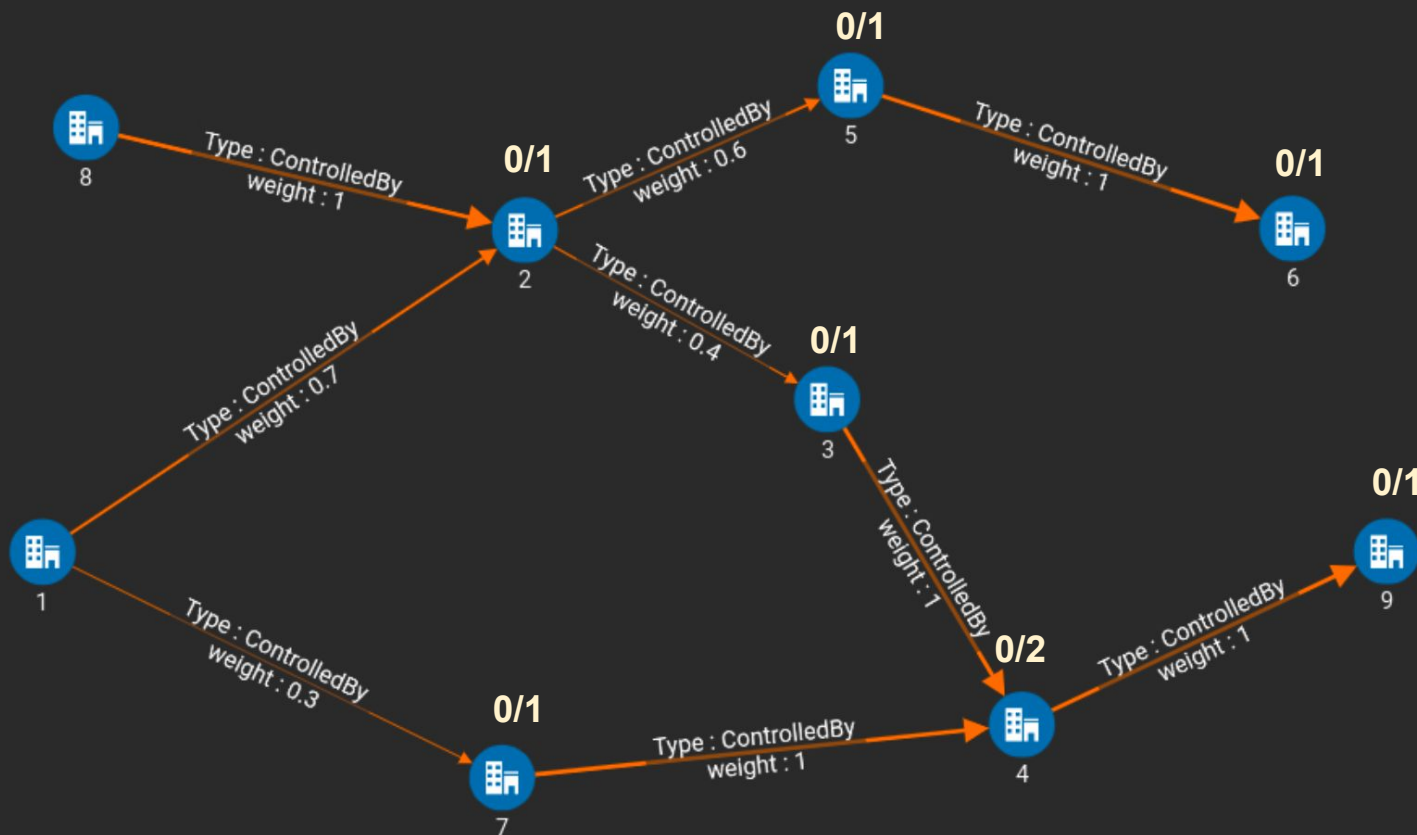
            Start = SELECT s FROM Start:s ORDER BY s.@score desc LIMIT 5;

            PRINT Start;
        }
}
```

A Better Traversal Plan

5. Topology Sort

Continue to traverse only when enough # of message is received



```
CREATE QUERY getOwnershipPert (vertex<Company> inputComp) FOR GRAPH MyGraph
{
    SumAccum<int> @msgCnt1, @msgCnt2;
    OrAccum<bool> @visited;
    SumAccum<float> @score = 0;
    SetAccum<vertex> @@results;

    Start = {inputComp};

    WHILE Start.size() > 0 limit 8 DO
        Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
            ACCUM t.@msgCnt1 += 1
            POST-ACCUM s.@visited = true
            HAVING t.@visited == false; // don't start again for the
second visit;
        end;

        Start = {inputComp};

        Start = SELECT s FROM Start:s ACCUM s.@score = 1; // initialize @score

        WHILE Start.size() > 0 LIMIT 8 DO
            Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
                ACCUM t.@msgCnt2 += 1, t.@score += s.@score * e.weight
                POST-ACCUM
                CASE WHEN t.outdegree("ControlledBy") == 0 THEN
                    @@results += t
                END
                HAVING t.@msgCnt2 == t.@msgCnt1; // make sure got all the
scores
            END;

            Start = @@results;

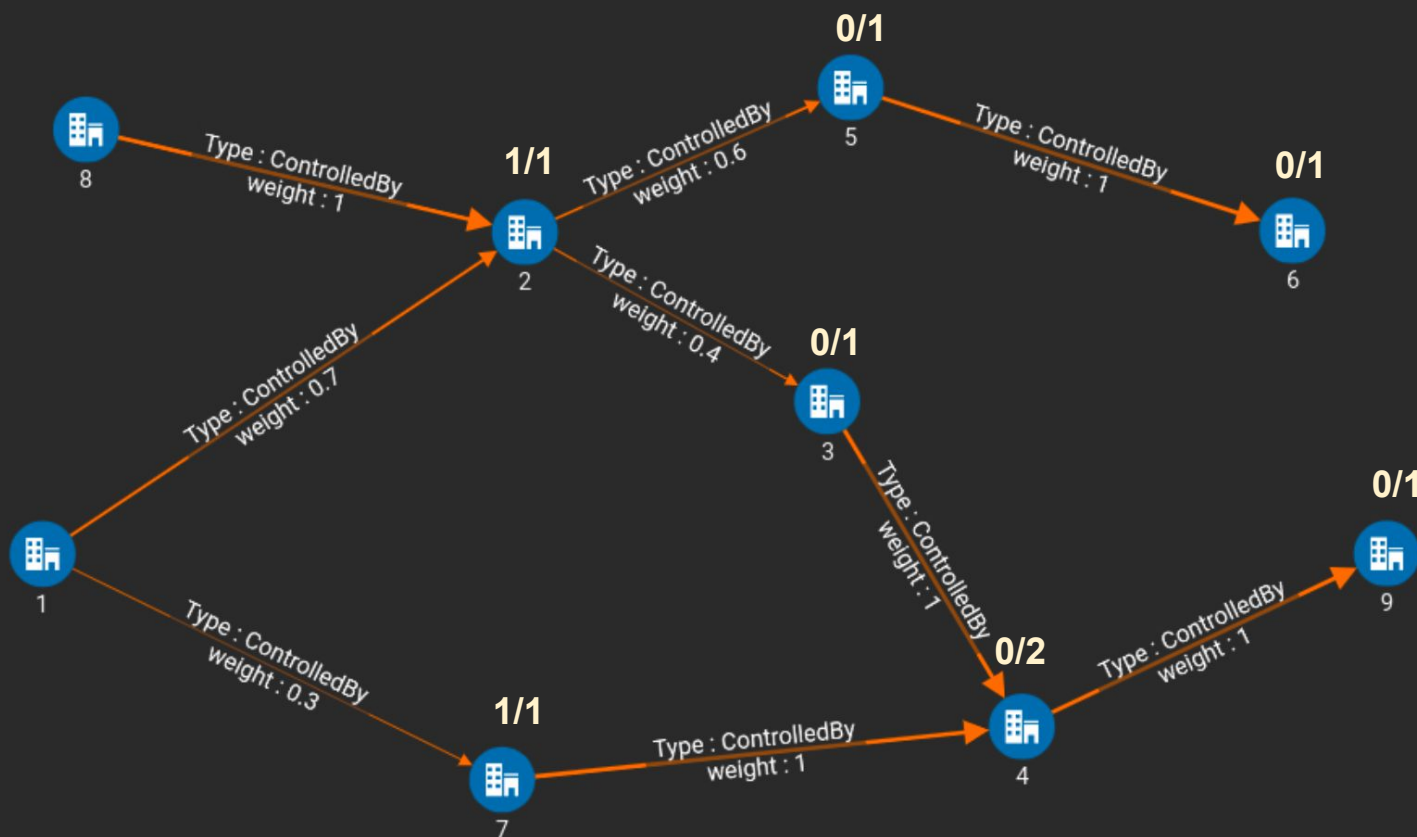
            Start = SELECT s FROM Start:s ORDER BY s.@score desc LIMIT 5;

            PRINT Start;
        }
}
```

A Better Traversal Plan

5. Topology Sort

Continue to traverse only when enough # of message is received



```
CREATE QUERY getOwnershipPert (vertex<Company> inputComp) FOR GRAPH MyGraph
{
    SumAccum<int> @msgCnt1, @msgCnt2;
    OrAccum<bool> @visited;
    SumAccum<float> @score = 0;
    SetAccum<vertex> @@results;

    Start = {inputComp};

    WHILE Start.size() > 0 limit 8 DO
        Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
            ACCUM t.@msgCnt1 += 1
            POST-ACCUM s.@visited = true
            HAVING t.@visited == false; // don't start again for the
second visit;
        end;

        Start = {inputComp};

        Start = SELECT s FROM Start:s ACCUM s.@score = 1; // initialize @score

        WHILE Start.size() > 0 LIMIT 8 DO
            Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
                ACCUM t.@msgCnt2 += 1, t.@score += s.@score * e.weight
                POST-ACCUM
                CASE WHEN t.outdegree("ControlledBy") == 0 THEN
                    @@results += t
                END
                HAVING t.@msgCnt2 == t.@msgCnt1; // make sure got all the
scores
            END;

            Start = @@results;

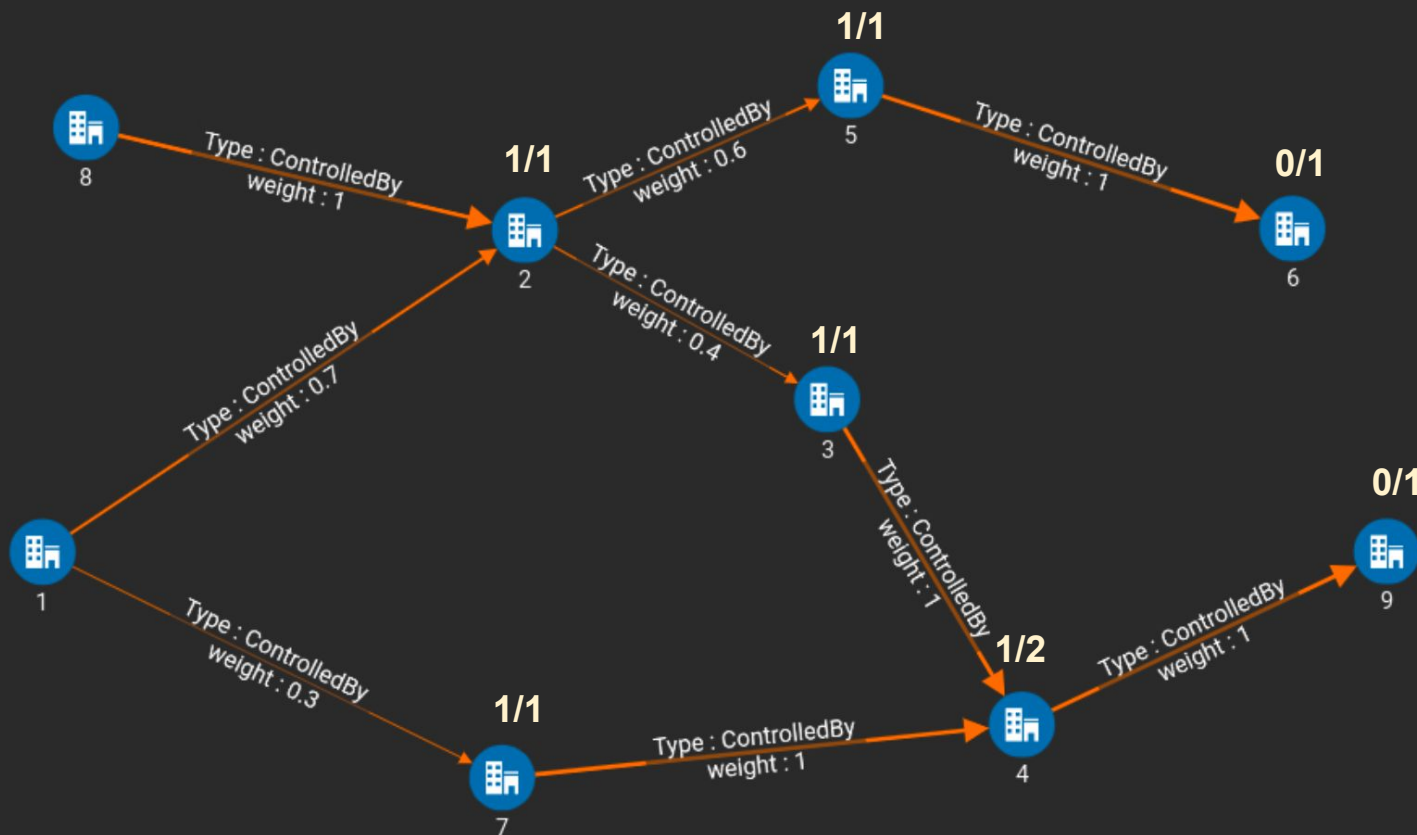
            Start = SELECT s FROM Start:s ORDER BY s.@score desc LIMIT 5;

            PRINT Start;
        }
}
```


A Better Traversal Plan

5. Topology Sort

Continue to traverse only when enough # of message is received



```
CREATE QUERY getOwnershipPert (vertex<Company> inputComp) FOR GRAPH MyGraph
{
    SumAccum<int> @msgCnt1, @msgCnt2;
    OrAccum<bool> @visited;
    SumAccum<float> @score = 0;
    SetAccum<vertex> @@results;

    Start = {inputComp};

    WHILE Start.size() > 0 limit 8 DO
        Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
            ACCUM t.@msgCnt1 += 1
            POST-ACCUM s.@visited = true
            HAVING t.@visited == false; // don't start again for the
second visit;
        end;

        Start = {inputComp};

        Start = SELECT s FROM Start:s ACCUM s.@score = 1; // initialize @score

        WHILE Start.size() > 0 LIMIT 8 DO
            Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
                ACCUM t.@msgCnt2 += 1, t.@score += s.@score * e.weight
                POST-ACCUM
                CASE WHEN t.outdegree("ControlledBy") == 0 THEN
                    @@results += t
                END
                HAVING t.@msgCnt2 == t.@msgCnt1; // make sure got all the
scores
            END;

            Start = @@results;

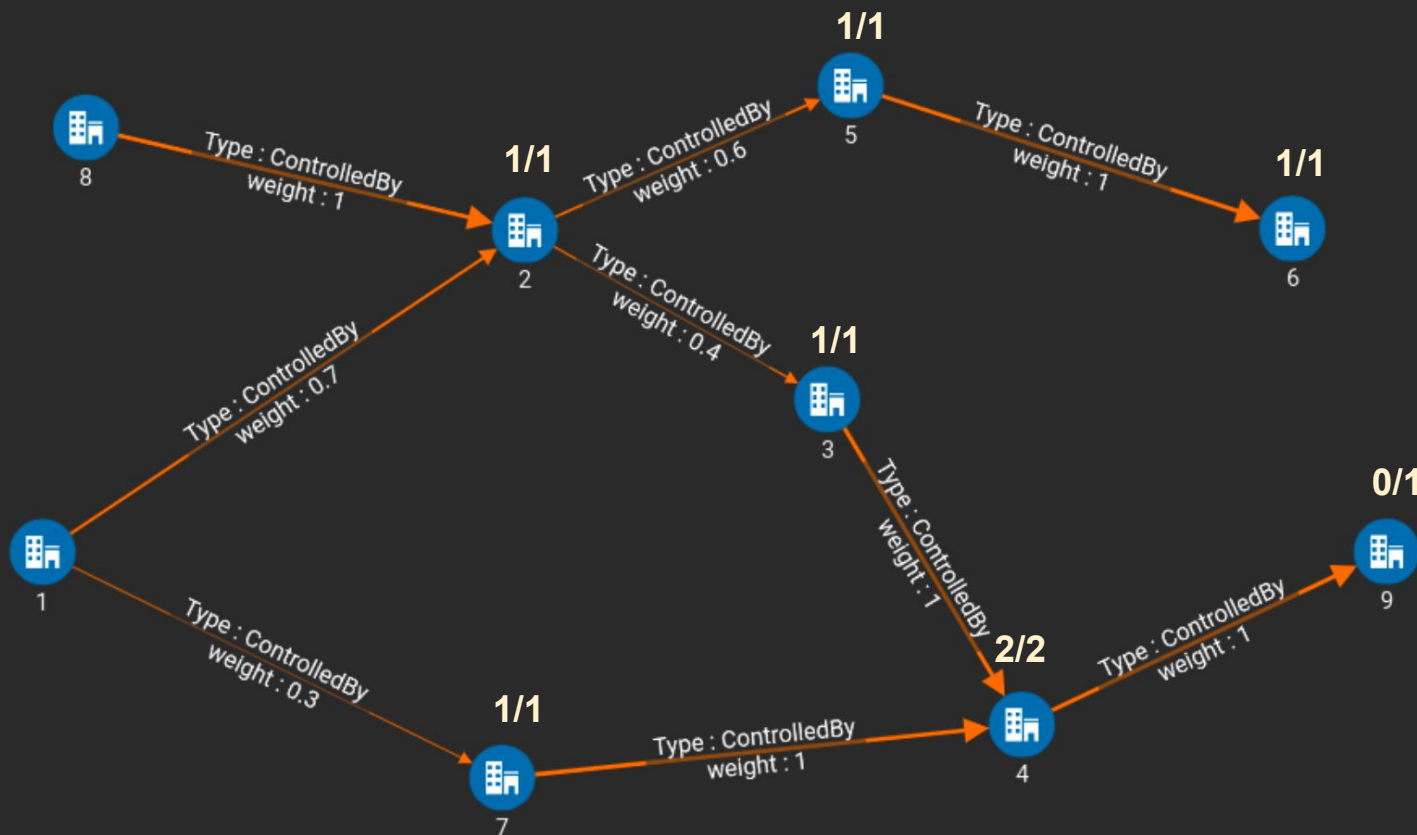
            Start = SELECT s FROM Start:s ORDER BY s.@score desc LIMIT 5;

            PRINT Start;
        }
}
```

A Better Traversal Plan

5. Topology Sort

Continue to traverse only when enough # of message is received



```
CREATE QUERY getOwnershipPert (vertex<Company> inputComp) FOR GRAPH MyGraph
{
    SumAccum<int> @msgCnt1, @msgCnt2;
    OrAccum<bool> @visited;
    SumAccum<float> @score = 0;
    SetAccum<vertex> @@results;

    Start = {inputComp};

    WHILE Start.size() > 0 limit 8 DO
        Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
            ACCUM t.@msgCnt1 += 1
            POST-ACCUM s.@visited = true
            HAVING t.@visited == false; // don't start again for the
second visit;
        end;

        Start = {inputComp};

        Start = SELECT s FROM Start:s ACCUM s.@score = 1; // initialize @score

        WHILE Start.size() > 0 LIMIT 8 DO
            Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
                ACCUM t.@msgCnt2 += 1, t.@score += s.@score * e.weight
                POST-ACCUM
                CASE WHEN t.outdegree("ControlledBy") == 0 THEN
                    @@results += t
                END
                HAVING t.@msgCnt2 == t.@msgCnt1; // make sure got all the
scores
            END;

            Start = @@results;

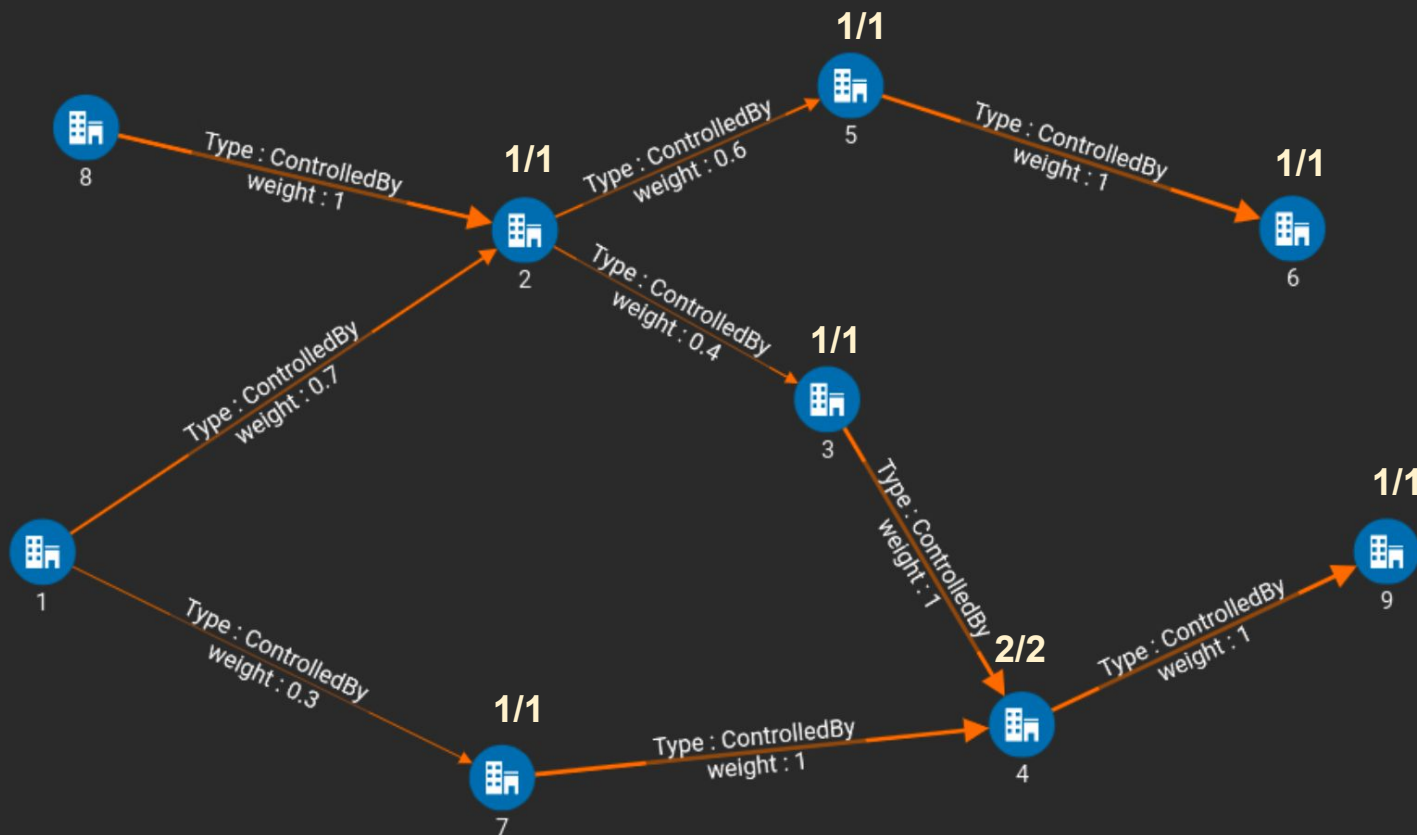
            Start = SELECT s FROM Start:s ORDER BY s.@score desc LIMIT 5;

            PRINT Start;
        }
}
```


A Better Traversal Plan

5. Topology Sort

Continue to traverse only when enough # of message is received



```
CREATE QUERY getOwnershipPert (vertex<Company> inputComp) FOR GRAPH MyGraph
{
    SumAccum<int> @msgCnt1, @msgCnt2;
    OrAccum<bool> @visited;
    SumAccum<float> @score = 0;
    SetAccum<vertex> @@results;

    Start = {inputComp};

    WHILE Start.size() > 0 limit 8 DO
        Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
            ACCUM t.@msgCnt1 += 1
            POST-ACCUM s.@visited = true
            HAVING t.@visited == false; // don't start again for the
second visit;
        end;

        Start = {inputComp};

        Start = SELECT s FROM Start:s ACCUM s.@score = 1; // initialize @score

        WHILE Start.size() > 0 LIMIT 8 DO
            Start = SELECT t FROM Start:s-(ControlledBy:e)-:t
                ACCUM t.@msgCnt2 += 1, t.@score += s.@score * e.weight
                POST-ACCUM
                CASE WHEN t.outdegree("ControlledBy") == 0 THEN
                    @@results += t
                END
                HAVING t.@msgCnt2 == t.@msgCnt1; // make sure got all the
scores
            END;

            Start = @@results;

            Start = SELECT s FROM Start:s ORDER BY s.@score desc LIMIT 5;

            PRINT Start;
        }
}
```

Parallelization

1. Run queries in parallel **(7/10)**
2. Write to a file in parallel **(7/10)**
3. Single Server mode VERSUS Distributed mode **(8/10)**



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.

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.

Parallelization

2. Write to a file in parallel

```
CREATE QUERY sequentialExample (FILE fileObj) FOR GRAPH exampleGraph
{
    ListAccum<STRING> @@result;
    Start = {TYPEA.*};
    Start = SELECT s FROM Start:s
        ACCUM @@result +=s.strAttr;
    FOREACH str in @@result DO
        PRINT fileObj.println(str);
    END;
}
```



- This approach first gathers all the data in an accumulator → overhead for the storage and large number of string updates.
- Then sequential iteration over the output set → slow.

Parallelization

2. Write to a file in parallel

```
CREATE QUERY parallelPrintSub (FILE fileObj) FOR GRAPH exampleGraph {  
    Start = {TYPEA.*};  
    Start = SELECT s FROM Start:s  
        ACCUM f.println(s.strAttr);  
}
```



- This approach writes each output item direct → no intermediate storage or string updates.
- Parallel printing in ACCUM / POST-ACCUM clause or in subquery is more efficient.

Parallelization

2. Write to a file in parallel

```
CREATE QUERY subQuery (VERTEX v, FILE fileObj) FOR GRAPH exampleGraph
{
    Start = {v};
    .....
    Start = SELECT s FROM Start:s
        ACCUM f.println(s.strAttr);
}
```

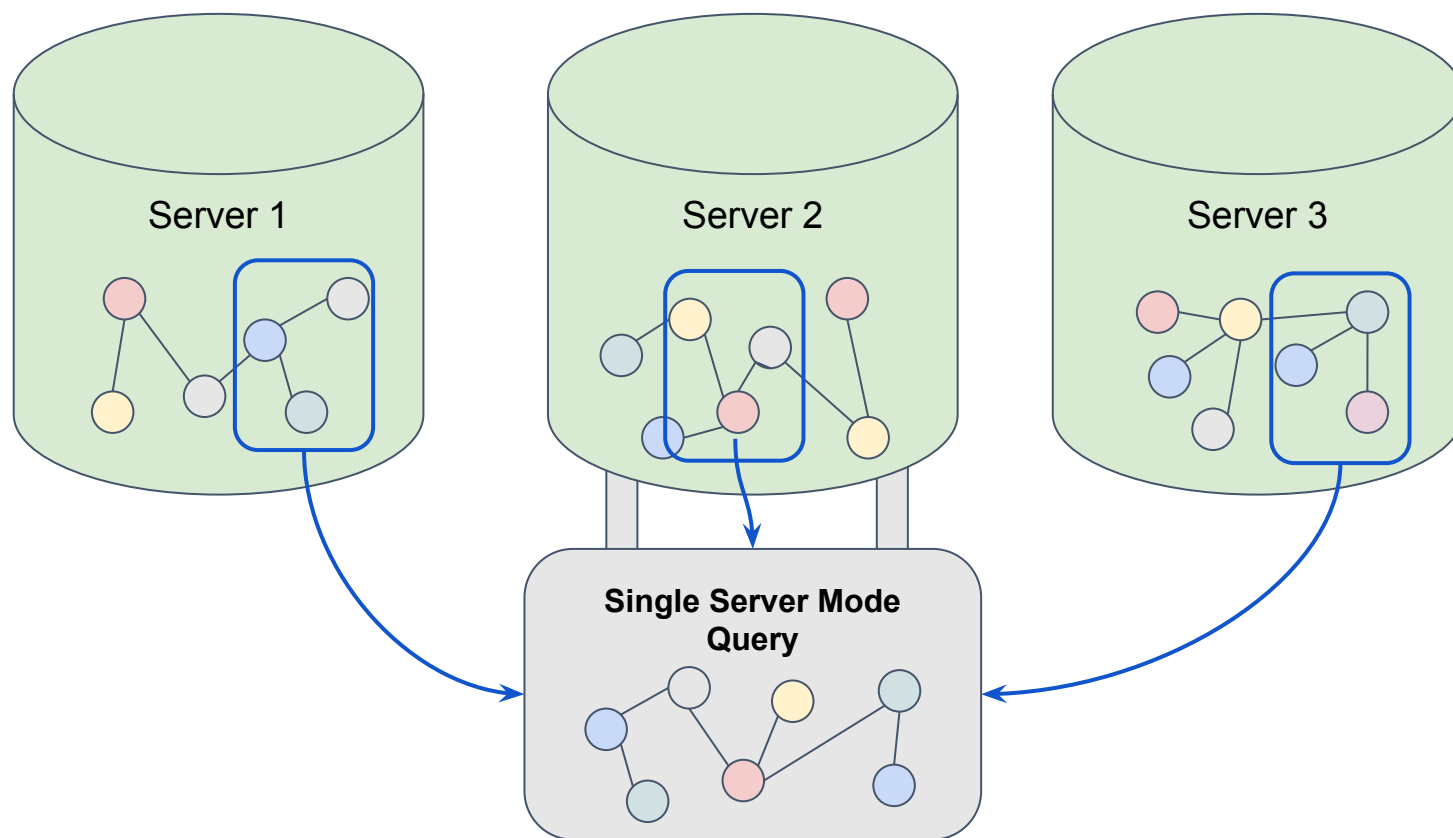
```
CREATE QUERY mainQuery (FILE fileObj) FOR GRAPH exampleGraph {
    Start = {TYPEA.*};
    Start = SELECT s FROM Start:s
        ACCUM subQuery(s, fileObj);
}
```



- This is a query-calling-query example of passing file object as input parameter

Parallelization

3. Single Server mode vs. Distributed mode

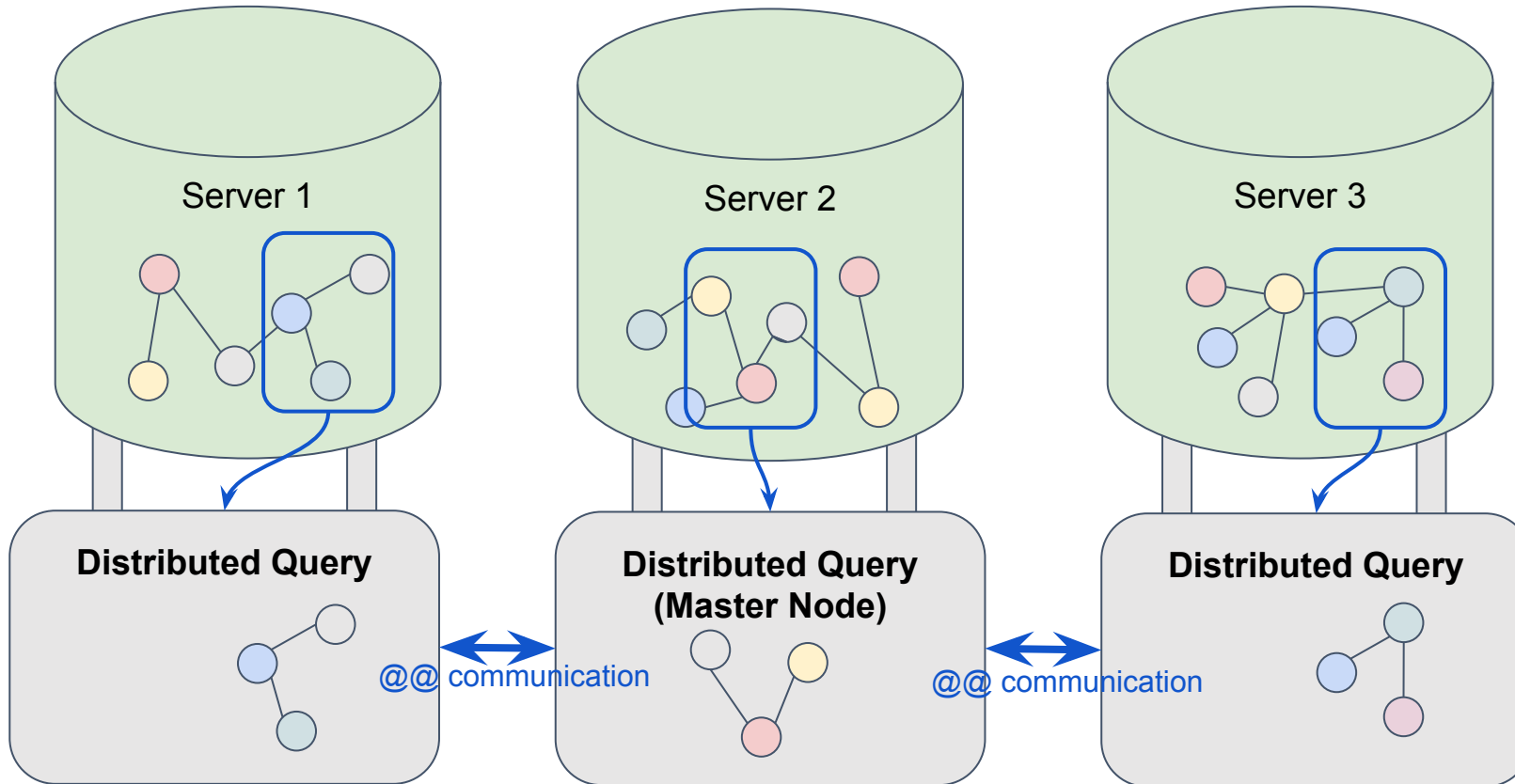


Single Server Mode

- 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.**

Parallelization

3. Single Server mode vs. Distributed mode

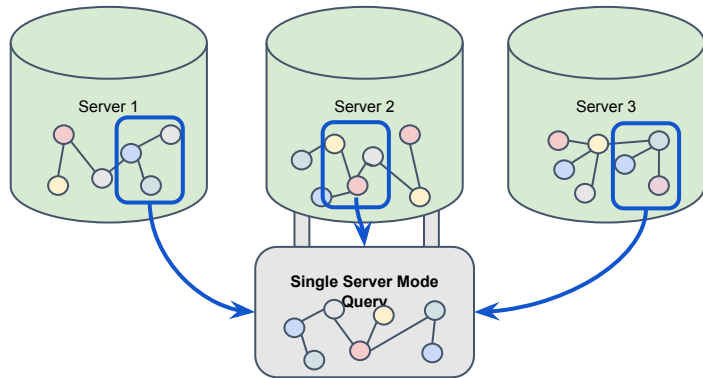


Distributed Mode

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

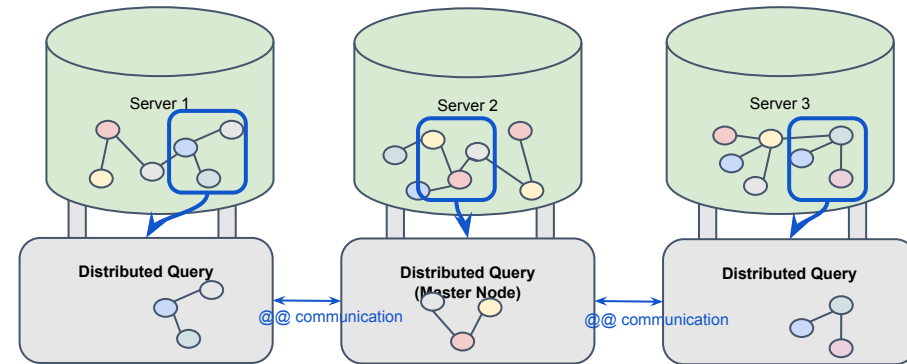
Parallelization

3. Single Server mode VERSUS Distributed mode



Single Server Mode is better when

1. Starting from a single or small number of vertices.
2. Modest number of vertex and edges are traversed.
3. Heavy usage of global accumulators.



Distributed Mode is better when

1. Starting from all or a large number of vertices.
2. Very large number vertex/edges are traversed.
3. Little usage of global accumulators.

Saving Memory

1. Split the query load into batches **(4/10)**
2. Query Calling Query **(5/10)**
3. Clear up used containers **(3/10)**



Saving Memory

1. 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.

Saving Memory

1. Split the query load into batches

```
// split the calculation into k batches
CREATE QUERY split (int k) FOR GRAPH exampleGraph {
  ... // define local and global accumulators
  AllV = {ANY};

  FOREACH i IN range [0, k] DO
    Start = SELECT s FROM AllV:s
      WHERE getvid(s) % k == i
    ... // do the calculation, populate global accumulator
    ... // clear up the local structure
    ;
  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.

Saving Memory

2. Query Calling Query

When is this needed?

1. Starting from a large amount of vertices
2. Result for each starting vertex needs to be collected from a subgraph in more than 1 hops.

- Breaking a query into subqueries usually results in simpler data structures.
- Only a limited amount of subqueries run in parallel
- Memory is recycled after a subquery is done.

Memory Saving

2. Query Calling Query

E.g. In a enterprise graph, find companies having more than n bankrupt subsidiaries within k hops.

```
CREATE QUERY bankruptSub (int n, int k) FOR GRAPH exampleGraph {  
  
  SetAccum<vertex> @bankruptSet, @@result;  
  Start = {Enterprise.*};  
  Start = SELECT s FROM Start:s WHERE s.bankrupt == true ACCUM s.@bankruptSet += s;  
  WHILE Start.size() > 0 LIMIT k DO  
    Start = SELECT t FROM Start:s-(ControlledBy)->t  
      ACCUM t.@bankruptSet += s.@bankruptSet,  
      t.@bankruptSet += t  
    POST-ACCUM  
      CASE WHEN t.@bankruptSet.size() > n THEN  
        @@result += t  
      END;  
  END;  
  PRINT @@result;  
}
```



Memory Saving

2. Query Calling Query

```
CREATE QUERY subQuery (vertex v, int n, int k) FOR GRAPH exampleGraph RETURNS (BOOL){
  SumAccum @@cnt;
  Start = {v};
  WHILE Start.size() > 0 AND @@cnt <= n LIMIT k DO
    Start = SELECT t FROM Start:s-(Controls)->t
      POST-ACCUM
        CASE WHEN t.bankrupt == true THEN
          @@cnt += 1
        END;
  END;
  CASE WHEN @@cnt >= n THEN RETURN true ELSE RETURN false END;
}

CREATE QUERY bankruptSub (int n, int k) FOR GRAPH exampleGraph {
  SetAccum<vertex> @@result;
  Start = {Enterprise.*};
  Start = SELECT s FROM Start:s CASE WHEN subQuery(s,n,k) THEN @@result += s END;
}
```



When there are a huge amount of subqueries called, calling subquery **might be slower**. It depends on the # of hops traversed. However with limited memory it is a better choice.

Memory Saving

3. Clear Up Used Container

When is this needed?

1. The query uses vertex-attached container type accumulators.
2. The query traverses multiple hops and doesn't revisit vertices.

- Us the `clear()` function to empty container type accumulators releases memory.

Preprocessing

1. Build derived edges **(8/10)**
2. Fetch the attributes needed far away **(8/10)**



Preprocessing

1. Build derived edges

Relationships, which do not exist in the input data, can be discovered by running graph analytic algorithms and then adding them to become part of the graph. Update the graph schema to include new relationship types.

For example: Queries frequently search for the info of the ultimate parent company. Create edges from company to ultimate parent.

1. Create new edge type **has_ultimate_parent** in graph schema.
2. Run a preprocessing query to create the **has_ultimate_parent** edges.
3. Utilize the **has_ultimate_parent** edge to speed up future queries.

Preprocessing

2. Fetch the attributes needed far away

When some of the attributes needed for filtering or calculation are far away from the target vertex and are frequently used, preprocessing can be done to accelerate the query and avoid duplicated work load in retrieving those attributes.

For example: A query frequently traverses companies that have more than 100 subsidiaries.

1. Add the boolean attribute **has_100_plus_sub** to the company vertex type.
2. Run a preprocessing query to populate the **has_100_plus_sub** attributes.
3. Utilize the **has_100_plus_sub** attribute to speed up future queries.

Miscellaneous

1. Log Statement **(-/10)**
2. Specify Vertex Types **(2/10)**
3. Avoid using target attribute in edge-induced ACCUM **(6/10)**



Miscellaneous

1. Log Statement


Another method of outputting is the LOG statement. To find the output of the log, use the command **gadmin log gpe** and open the file ending in INFO. It is good practice to leave a distinct string, which would allow you to find the log statement quicker.


```
BOOLEAN debug = TRUE;  
INT x = 10;  
LOG(debug, 20);  
LOG(debug, "X MARKS THE SPOT", x);
```

Miscellaneous

2. Specify Vertex Set types

When defining a vertex set. Its type can be specified. Specifying the vertex types can improve the performance to some degree, since it saves some condition checking in the generated C++ code.

Start (TypeA | TypeB) = {@@set}; 

Start (ANY) = {@@set}; 

Miscellaneous

3. Avoid using target vertex attributes in edge-induced ACCUM or WHERE (Especially in DISTRIBUTED mode)

In an edge-induced ACCUM clause, the data chunk containing source vertex attributes and edge attributes are already at hand. Getting target attributes requires extra lookup.

If the query is in DISTRIBUTED mode, then the target vertices may be on another server. Then, getting target attributes is even more expensive.

Therefore, to achieve the best performance, one should try to avoid using target attributes.