Graph Database Management System

Che Liu
A53209595

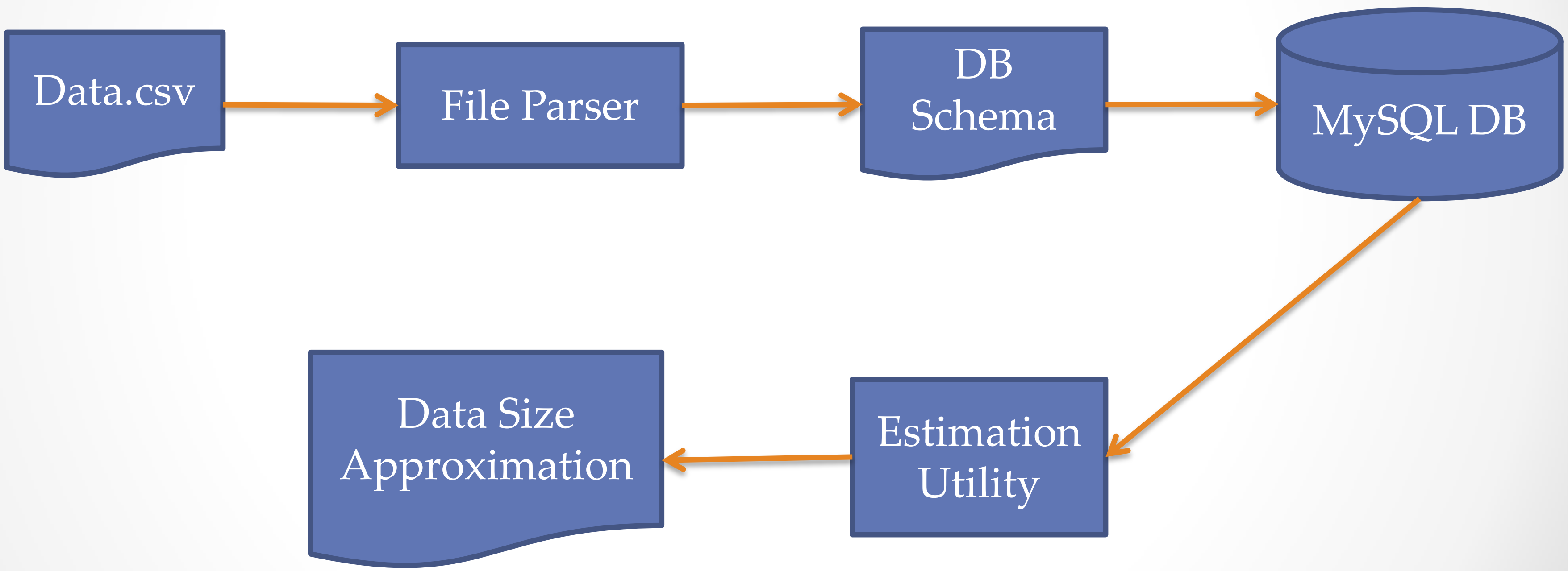
Overview

Overview

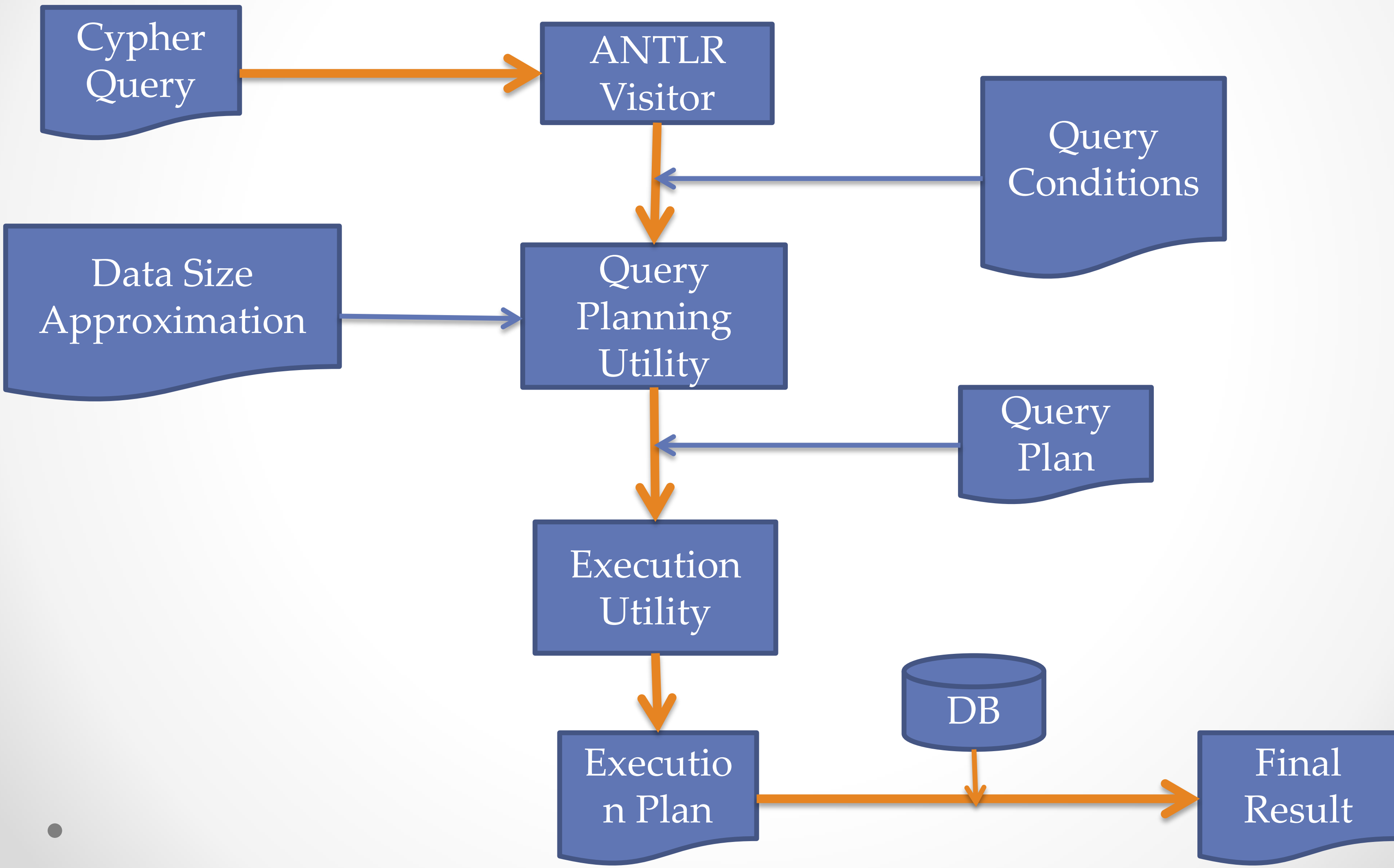
- **File Parsing**
  - Generate MySQL Schema
  - Import data into database
  - Data estimation

- **Query Execution**
  - Query parsing
  - Generate query plan, execution plan
  - Produce results

File Parsing



Query Execution



System Components

- File import utility
- MySQL Database
- Parsing query string
- Data size approximation
- Query planning and size estimation
- Query execution

File import utility

Parsing

- Buffered Reader to read file
- Convert to JSON Object

SQL

- Generate SQL Schema
- Generate insert statements

Import DB

- JDBC
- Batch insert to database

File Import Utility

File import utility

- **First scan:**
  - Create metadata:
  - Collect schema of different type of nodes
  - Convert type to SQL types.
  - Set up data tables and indices.

- **Second scan:**
  - Collect nodes and insert them to database
    - Assume ID in metadata is unique among nodes.
  - Assign each node with a unique GID (Global ID).
  - Insert node and its properties to DB.

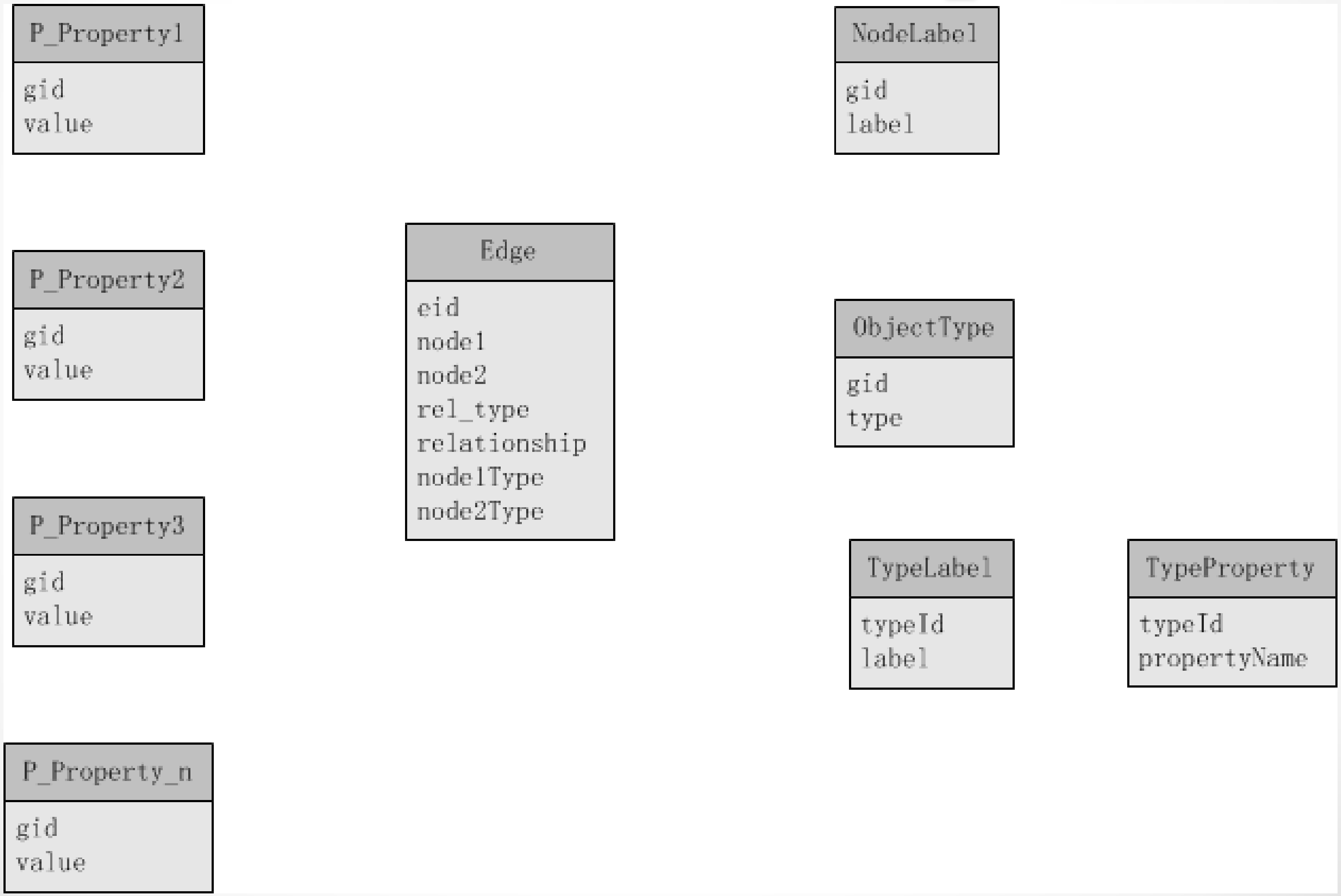
MySQL Database Design

MySQL Database

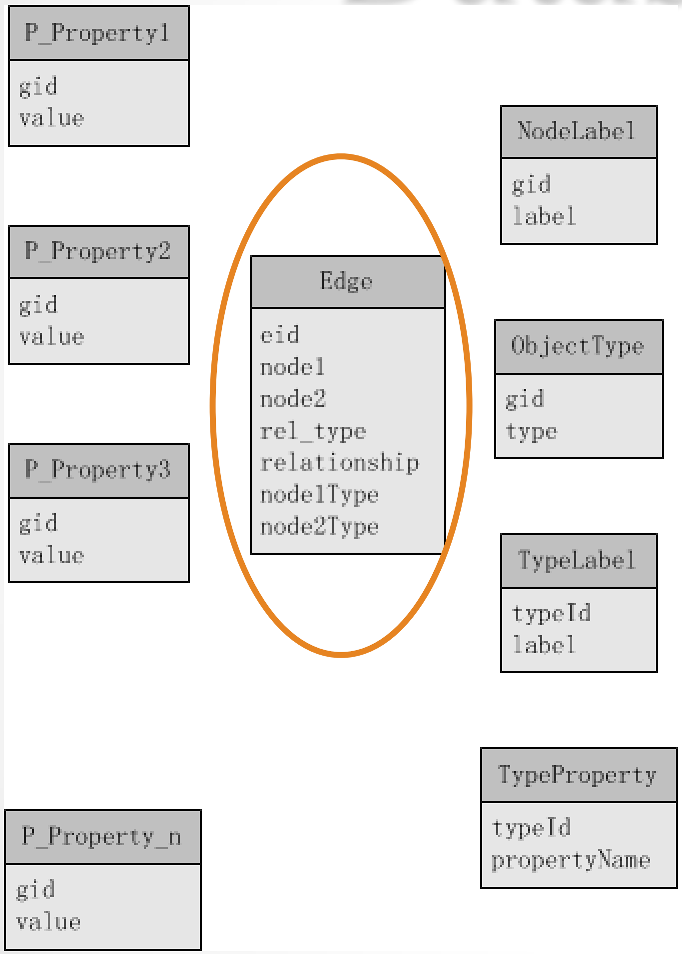
- **Column-based storage:**
  - Same properties of all the nodes are stored in the same table.
  - GID – value (key-value) storage.
  - Index on all primary keys
  - Index on frequent used values: TID, UID, Label, Name…

- **Small Edge table**
  - Unique index number
  - Node1 – relation_type – Node2
  - Additional information (nodeLabel) for fast lookup
  - If small enough, it can stay in memory.

Database Graph



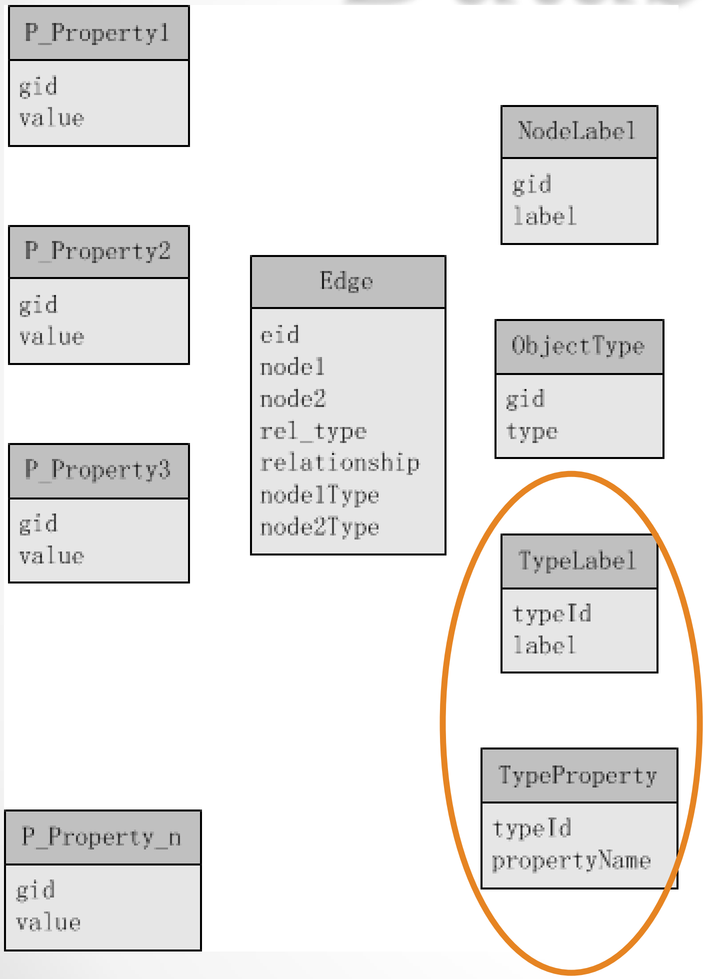
Database Graph



Edge Table

- Small enough to be kept in memory
- Node – Edge – Node
- Keep node type in table to accelerate lookup
- Everything have index.

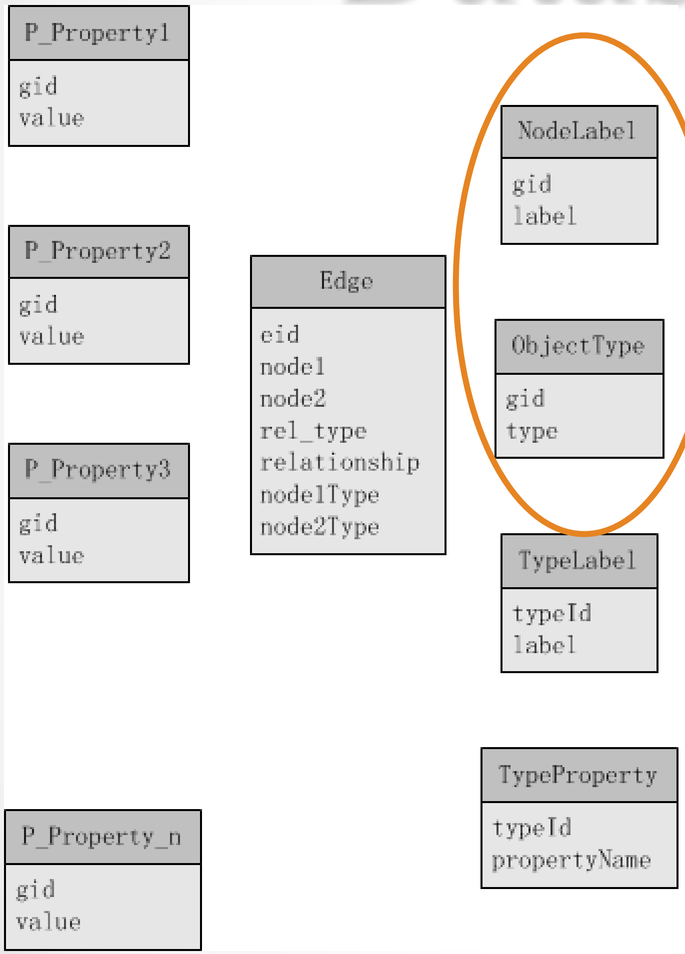
Database Graph



Database Meta

- Assumption:
  - Node with same label have the same schema.
  - One node can have many labels
- Assign each kind of node a type number
  - {Tweet} => 1
  - {Hashtag, LongHashtag} => 2
  - {Hashtag} => 3
- Keep schema in metadata

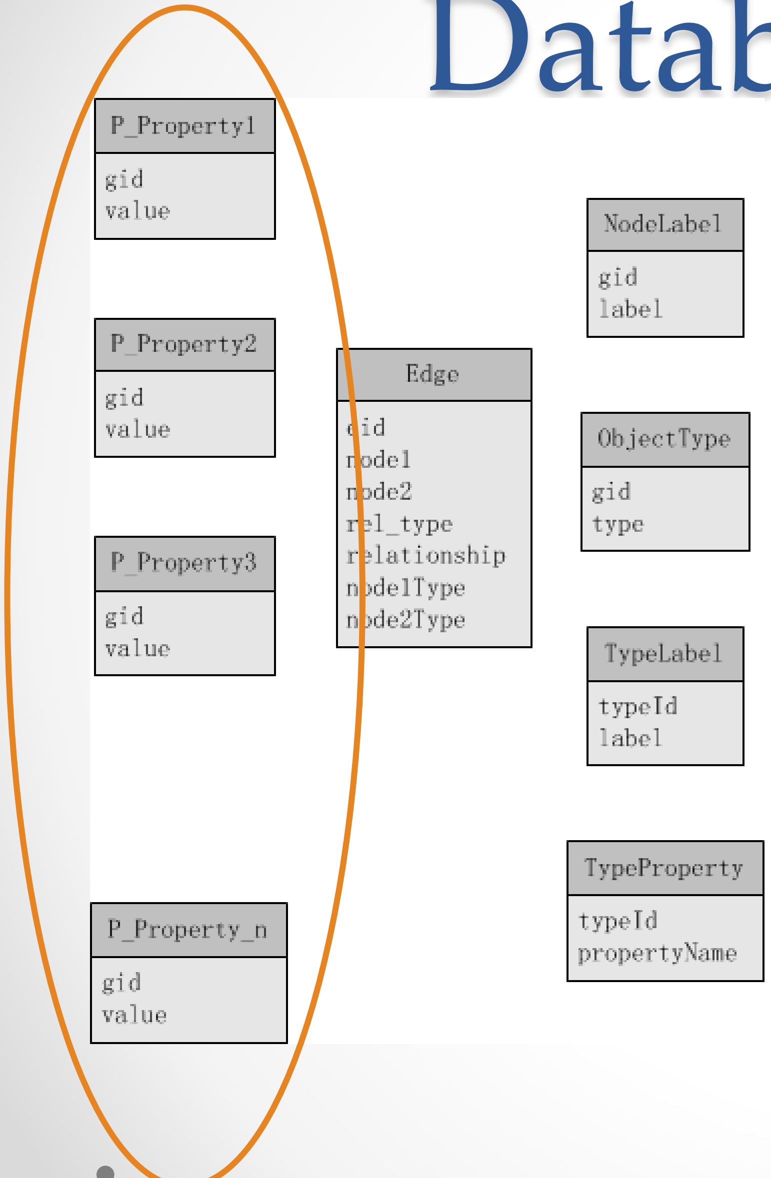
Database Graph



Node Meta

- Gid – value pairs.
- Information about labels and type of a node.

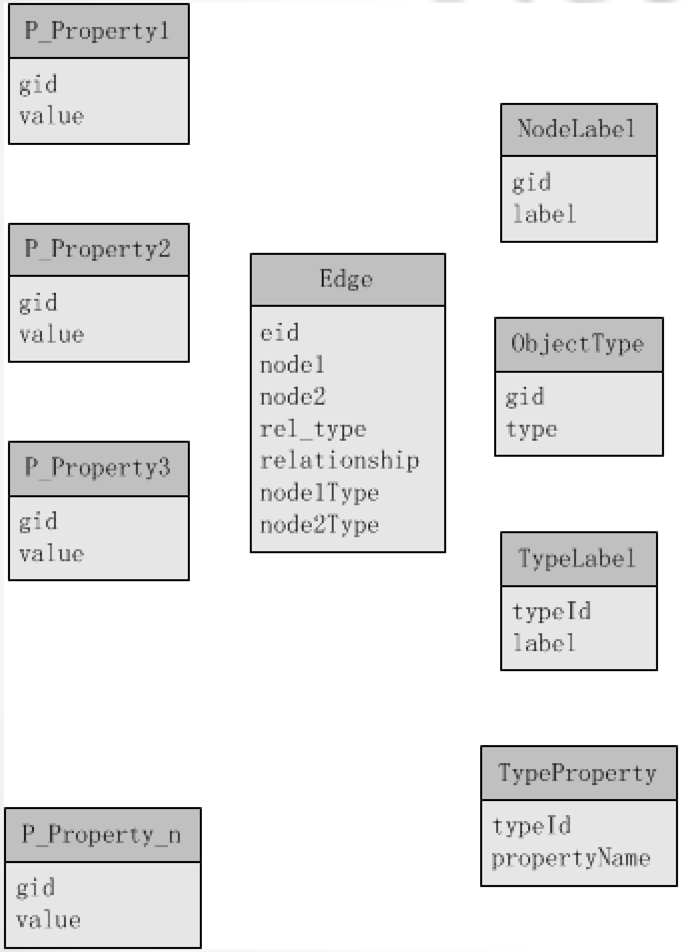
Database Graph



Properties table

- Gid – value pairs.
- Contents in the node

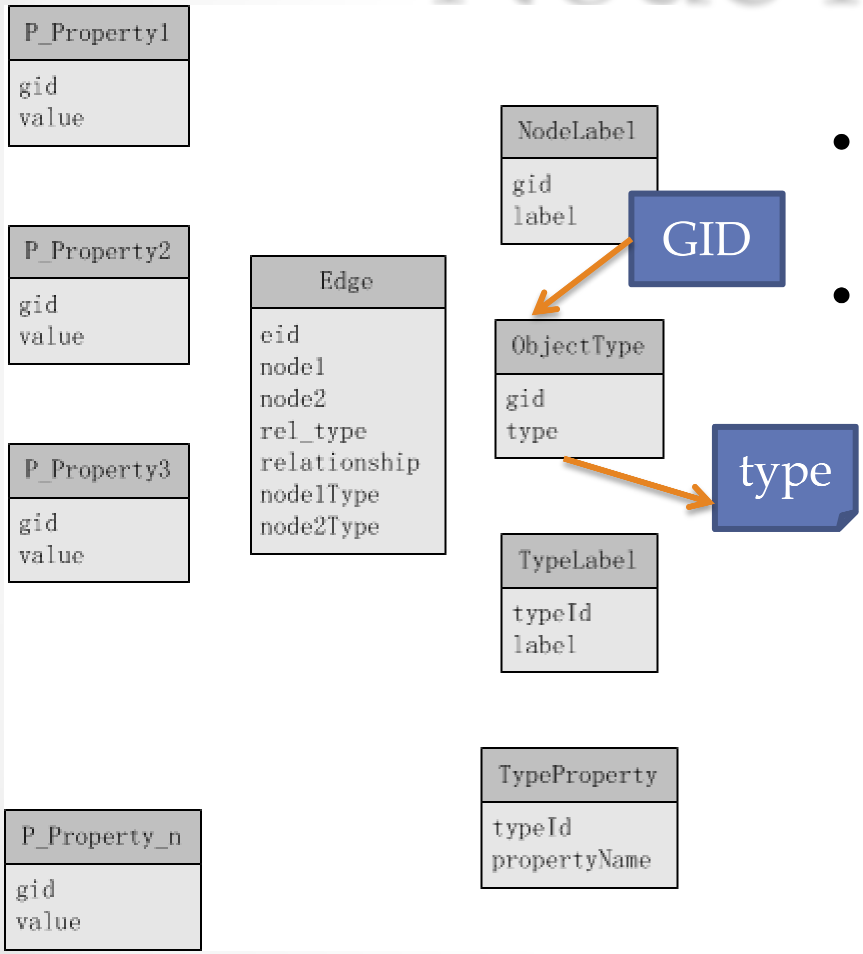
Node lookup



- GID

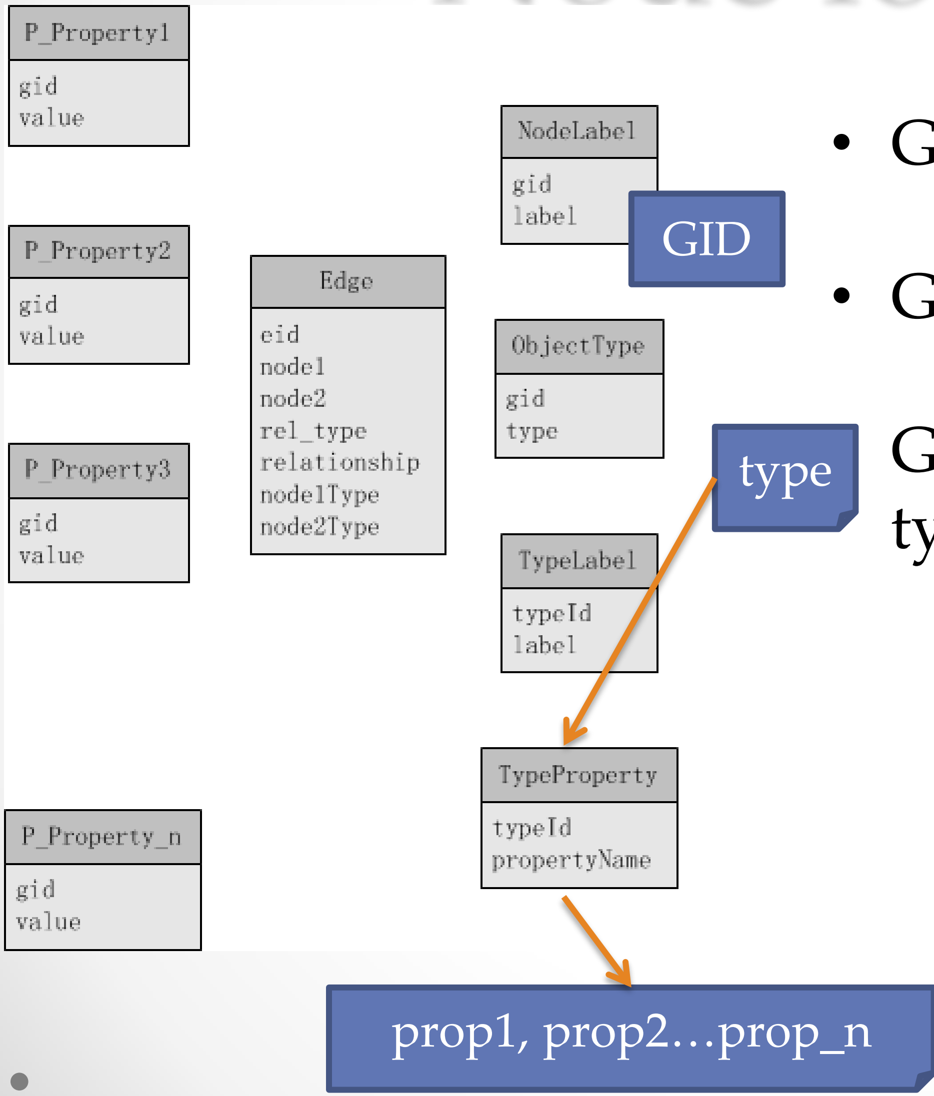
GID

Node lookup



- GID
- Get Node Type

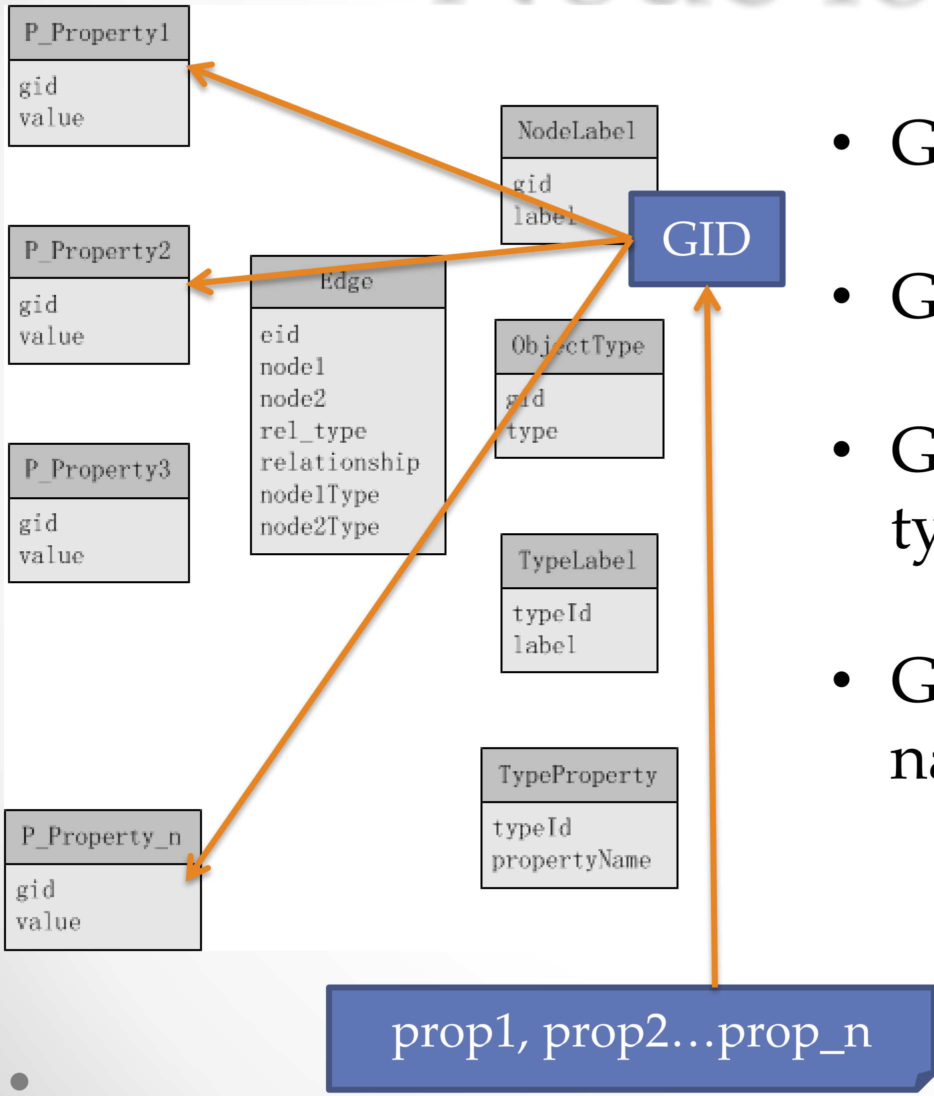
Node lookup



- GID
- Get Node Type

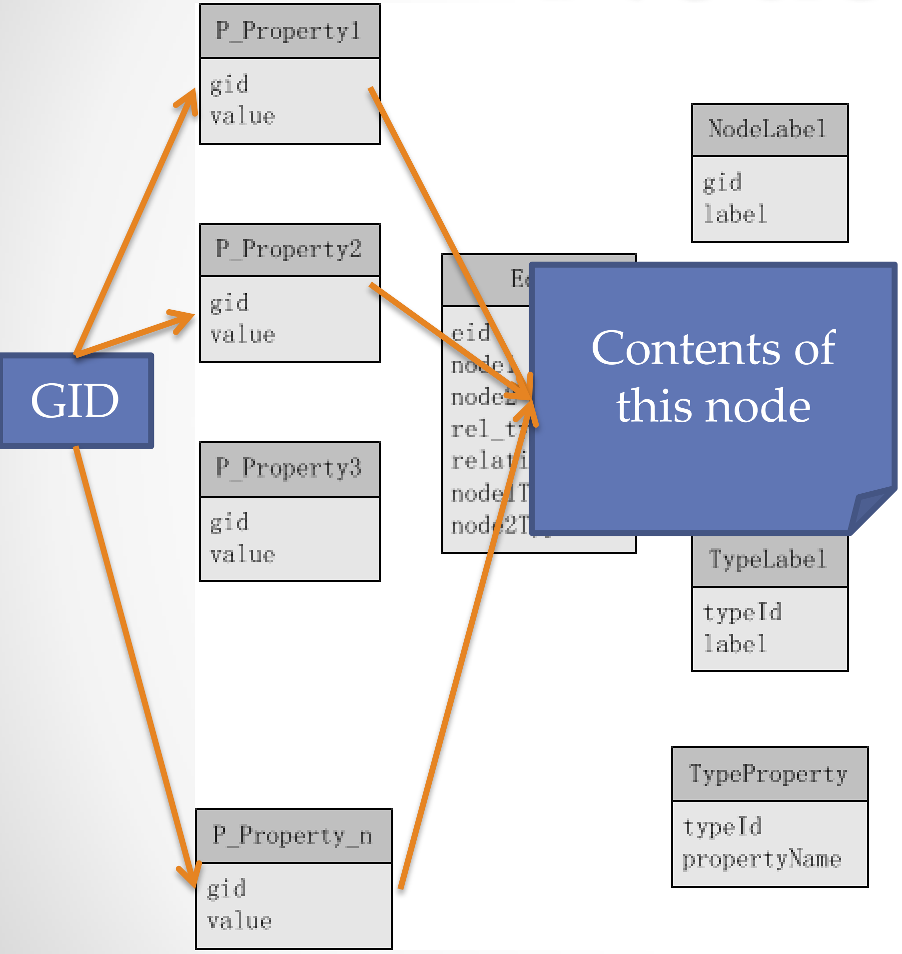
Get property names of this type of node

Node lookup



- GID
- Get Node Type
- Get property names of this type of node
- Get contents by property name and GID

Node lookup



- GID
- Get Node Type
- Get property names of this type of node
- Get contents by property name and GID
- Combine these results

Data Size Estimation

Data Size Estimation

- Run when importing finish.
- Provide rough insight of current dataset.
- Once created, never changes.
- Useful for size and cost estimation.

Data Size Estimation

Name	Description
NumberOfNodes	Total number of nodes
NumberOfRelations	Total number of relations
MaxNodeIn	Maximum in-degree of node
MaxNodeOut	Maximum out-degree of node
LabelRelation	Number of relations with each label(rel_type)
LabelNodes	Number of nodes with same label set
PropCntOfNode	Number of nodes with different value on each property
NodeLabelIncoming/ NodeLabelIOuting	Number of incoming/outgoing edges from nodes with same label
NodeRelationInEdge/ NodeRelationOutEdge	Number of incoming/outgoing edges of the same type from nodes with same label

Query Parsing

Query Parsing

- Visitor Method to walk the parse tree
- **Constants** are evaluated immediately
- **Variables** and related **expressions** are collected together.
- **Return** expressions are parsed and collected into somewhere else.
- Patterns are extracted after visiting **match** clause

Query Parsing Example

Query:

```
Match (a) – [r1:hashtagUsedIn] –> (b:tweet:longtweet) ,
        (c:user {id = 1234}) – [r2:mentioned*1..3] – (b),
MATCH (e) <- [r3] - (f)
Where a.name = d.name AND b = f
return a.id, r1, b, f
```

Query Parsing Example

Query:

```
Match (a) – [r1:hashtagUsedIn] –> (b:tweet:longtweet) ,
        (c:user {id = 1234}) – [r2:mentioned*1..3] – (b),
MATCH (e) <- [r3] - (f)
Where a.name = d.name AND b = f AND e.id = 1000
return a.id, r1, b, f
```

Query Parsing Example

- Collect conditions on each variable:

- a: -
- r1 :hashtagUsedIn
- b :tweet:longtweet
- c :user id = 1234
- r2 :mentioned*1..3
- e .id = 1000
- r3: -
- d: -

```
MATCH
    (a) – [r1:hashtagUsedIn] –>
   (b:tweet:longtweet) ,
   (c:user {id = 1234}) –
   [r2:mentioned*1..3] – (b),
MATCH
        (e) <- [r3] - (d)
WHERE
        a.name = d.name AND
        b = d AND
        e.id = 1000
return a.id, r1, b, d
```

Query Parsing Example

- Collect conditions between variables:

- a.name = d.name
- b = d
- r1 != r2

```
MATCH
    (a) – [r1:hashtagUsedIn] –>
   (b:tweet:longtweet) ,
   (c:user {id = 1234}) –
   [r2:mentioned*1..3] – (b),
MATCH
        (e) <- [r3] - (d)
WHERE
       a.name = d.name AND
       b = d AND
       e.id = 1000
return a.id, r1, b, d
```

Query Parsing Example

- Collect query patterns:

- a-r1->b
- c-r2->b
- e<-r3-d

- Long patterns are also supported.

```
MATCH
      (a) – [r1:hashtagUsedIn] –>
     (b:tweet:longtweet) ,
     (c:user {id = 1234}) –
     [r2:mentioned*1..3] – (b),
MATCH
           (e) <- [r3] - (d)
WHERE
          a.name = d.name AND
          b = d AND
          e.id = 1000
return a.id, r1, b, d
```

Query Parsing Example

- Collect return variables:

- a.id
- b
- d
- r1

-

```
MATCH
    (a) – [r1:hashtagUsedIn] –>
   (b:tweet:longtweet) ,
   (c:user {id = 1234}) –
   [r2:mentioned*1..3] – (b),
MATCH
        (e) <- [r3] - (d)
WHERE
        a.name = d.name AND
        b = d AND
        e.id = 1000
return a.id, r1, b, d
```

Query Parsing

- Parse Results:
  - A list of matching patterns (node – edge - node)
  - A map containing all the conditions on each individual variable
  - A list containing all equations between variables
  - A list containing what result to return

Query Planning

Query planning

- Parse Results
- Greedy operator ordering algorithm
- Size estimation on each plan
  - Data Size Estimation
  - Size estimation of previous plan
- Produce plan tree

Query planning

- **Parse Results:**
  - A list of matching patterns (**PathList**)
  - A map containing all the conditions on each individual variable (**VarCondition**)
  - A list containing all equations between variables (**Equality**)
  - A list containing what result to return **(ReturnList**)

Greedy Operator Ordering

Input: Query graph Q

Result: Logical plan P that covers all nodes from Q

```
𝒫 ← []                                          ▷ PlanTable
foreach n ∈ Q do                                 ▷ every node in the query graph
    T ← constructLeafPlan(n)                     ▷ take selections into account
    𝒫.insert(T)
do
    Cand ← []                                    ▷ candidate solutions
    foreach P1 ∈ 𝒫 do
        foreach P2 ∈ 𝒫 do
            if CanJoin(P1, P2) then
                T ← constructJoin(P1, P2)
                Cand.insert(T)
    foreach P1 ∈ 𝒫 do
        T ← constructExpand(P1)
        Cand.insert(T)
    if Cand.size ≥ 1 then
        T_best ← pickBest(Cand)                  ▷ pick the plan with the smallest cost
        foreach T ∈ 𝒫 do
            if covers(T_best, T) then
                𝒫.erase(T)                       ▷ delete plans covered by T_best
        T_best ← applySelections(T_best)
        𝒫.insert(T_best)
while Cand.size ≥ 1
return 𝒫[0]
```

Query planning

- **Basic rules**
  - Push down selections (**ConstructLeafPlan**)
  - Choose the minimum cost plan at each step(**pickbest**)
  - Reduce candidate size by removing plans covered by the best plan. (**covers**)

Greedy Operator Ordering

- **Plan**
  - Name of this plan
  - Variables that this plan apply to
  - Estimated cost of this plan

- **Size estimation of a plan**
  - Data size estimation
  - Plan table that this plan is applied to
  - Type of plan

Greedy Operator Ordering

- **PlanTable**
  - Set of plans that are used
    - Represented by Query Plan Tree
  - Variables that covered by this plan table
  - Current estimated size
  - Cost to execute these plans
    - Sum of cost of all plans in this plan tree

Greedy Operator Ordering

- **ConstructLeaf**
  - For each node, apply selections immediately
  - Serve as leaf node in the plan tree.
  - Selection related with variables are stored in **VarCondition**

- **Produced plans:**
  - ScanByIdPlan, ScanByPropertyPlan, ScanByLabelPlan, AllNodeScanPlan
  - FilterConstraintsPlan (not a leaf plan)

Greedy Operator Ordering

- **ConstructJoin**
  - If two plan tables have common variables, or if there are equalities between two tables, produce NodeHashJoinPlan
    - [a, b, c] & [a, d, e]
    - [a, b] & [c, d] ON b = c
  - Otherwise, use Cartesian Product

- **Produced plans:**
  - CartesianProductPlan, NodeHashJoinPlan

Greedy Operator Ordering

- **ConstructExpand**
  - If some node in the PlanTable have an edge linked to it, produce ExpandAllPlan
    - Table: {a, r, c}, Edge: (a)-[r2]->(b)
  - If both end of the edge lies in the node set of a PlanTable, produce ExpandIntoPlan
    - Table: {a, b, r}, Edge: (a)-[r2]-(b)
  - If this expansion is variable length expand, construct RangedExpansion

- **Produced plans:**
  - ExpandAllPlan, ExpandIntoPlan
  - RangeExpandAllPlan, RangeExpandIntoPlan

Greedy Operator Ordering

- **AddAdditionalFilter**
  - Expand operation add variables into PlanTable
  - Check if there are additional constraints on these variables.
    - Table: {a, r, b}, ExpandAll: (a)->[r2]->(c)
    - Suppose additional condition:
      - r != r2, c.id = 2
- **Produced plans:**
  - FilterConstraintsPlan, FilterRelationEqualityPlan

Greedy Operator Ordering

- **ProduceFinalResult**
  - Look into return variables and conditions (**ReturnList**)
- **Produced plans:**
  - ProduceResultPlan

Greedy Operator Ordering

- **Types of generated plans:**
  - ScanByIdPlan, ScanByPropertyPlan, ScanByLabelPlan, AllNodeScanPlan
  - CartesianProductPlan, NodeHashJoinPlan
  - ExpandAllPlan, ExpandIntoPlan
  - RangeExpandAllPlan, RangeExpandIntoPlan
  - FilterConstraintsPlan, FilterRelationEqualityPlan
  - ProduceResultPlan

Query Plans

- **ScanByIDPlan(NodeVar)**
  - Id is unique
  - **Estimated size** = 1
  - **Estimated cost** = 1
  - **Variable** = Node

Query Plans

- **ScanByLabelsPlan(NodeVar, Labels[])**
  - **Variable** = Node
  - **Estimated size** = Minimum number of nodes in the dataset with one of the labels provided
    - Data size estimation provides:

      LabelNodes -> Number of nodes with same label set
  - **Estimated cost** = estimated size
    - Don't need to scan whole dataset since labels are indexed.

Query Plans

- **ScanByPropertyPlan**
- **(NodeVar, Property)**

  - **Variable** = Node

  - **Estimated size** = numberOfNodes / number of distinct values of this property.

  - **Estimated cost** =
    - Estimated size, if this property is indexed
    - Number of nodes, if not indexed.

Query Plans

- **AllNodeScanPlan(NodeVar)**
  - **Variable** = Node
  - **Estimated size** = number of nodes in dataset
  - **Estimated cost** = estimated size

Query Plans

- **FilterByConstraintPlan(Table, Condition)**
  - Takes a table, filter its result by the condition provided, and return the filtered result
  - **Variable** = Node
  - **Params** = condition
  - **Estimated size** =
    - min(number of records in previous table, maximum number of records in the dataset)
  - **Estimated cost** = number of nodes in previous table

Query Plans

- **FilterByRelationEquality**

  (Table, Equality)

  - Takes a table, filter its result by the relation equation provided, and return the filtered result
  - **Variable** = Variables in the equation
  - **Estimated size** =
    - Table size (Not easy to estimate)
  - **Estimated cost** =
    - number of records in previous table

Query Plans

- **ExpandAllPlan(Table, conditions, PatternEdge)**
  - Expand the result set by the edge provided
  - **Variable** = Start variable of the edge
    - Start variable is the node in this edge that also occurs in the provided table.
  - **Size estimation** is same as
    - Pattern (a:X:Y)-[:T1|:T2]->(b:W:Z)
      - a and b each has two labels
      - Upper bound on the cardinality: C
        - T1 = min(X-[:T1]->(), Y-[:T1]->()) + min(X-[:T2]->(), Y-[:T2]->())
        - T1′= min(()-[:T1]->W, ()-[:T1]->Z) + min(()-[:T2]->W, ()-[:T2]->Z)
        - C = min (T1, T1′)

Query Plans

- **ExpandAllPlan(Table, conditions, PatternEdge)**
  - If there are no conditions provided:
    - **Estimated size** =
    - tableSize * maxDegreeOfNode (conservative estimation)
    - tableSize * NumberOfEdges / NumberOfNodes
    - (rough estimation)

Query Plans

- **ExpandIntoPlan(Table, conditions, PatternEdge)**
  - Similar to ExpandAllPlan
  - Only difference is:
    - **Cost estimation** = min(size estimation, number of records in previous table)
    - If previous table is small, only a scan through previous table is needed.

Query Plans

- **RangedExpandAllPlan(Table, conditions, PatternEdge, Range)**
  - Range: (min, max)
  - Similar to ExpandAllPlan
  - We have to perform ExpandAll operation for MAX times, so:
  - **Cost Estimation** =
    - (cost of similar ExpandAllPlan) ^ MAX

Query Plans

- **RangedExpandIntoPlan(Table, conditions, PatternEdge, Range)**
  - Range: (min, max)
  - Similar to ExpandIntoPlan
  - We have to perform ExpandInto operation for MAX times, so:
  - **Cost Estimation** =
    - (cost of similar ExpandIntoPlan) ^ MAX

Query Plans

- **CartesianProductPlan(Table1, Table2)**
  - Perform CartesianProduct on two tables
  - No other condition needed.
  - **Variable**: All the variables in both tables
  - **Size estimation** =
    - Size of table1 * size of table2
  - **Cost Estimation** = size estimation

Query Plans

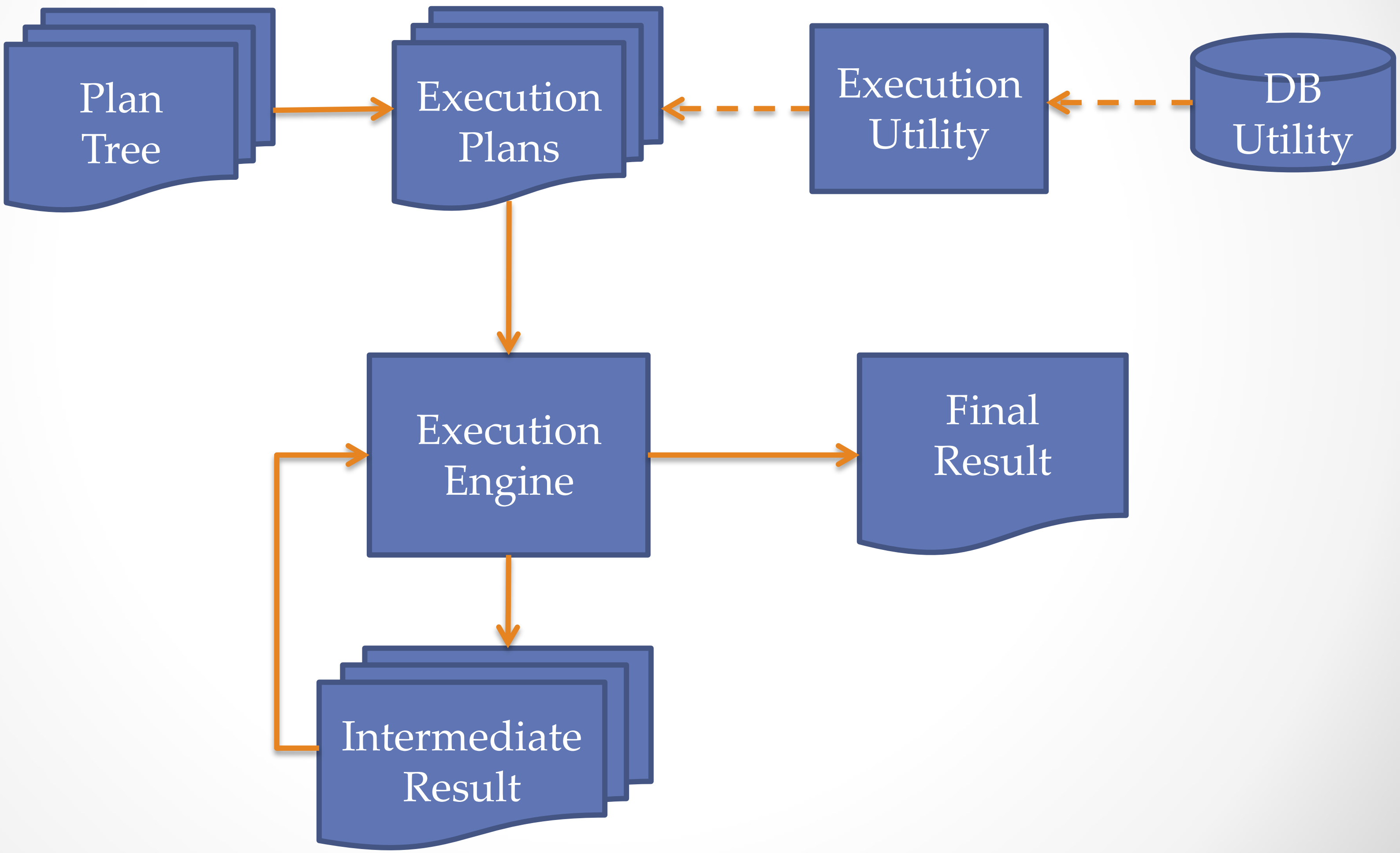
- **NodeHashJoinPlan(Table1**, **Table2**, **VarEqualities)**
  - Perform Hash Join on two tables, hash is created by the common variables or the equation provided.
  - **Variable**: All the variables in both tables
  - **Size estimation** =
    - Minimum size of two tables
  - **Cost Estimation** =
    - Sum of size of two tables

Query Execution

Query execution

- **Best possible Plan Tree**
- **Convert query plan into execution plan**
  - Done in the execution plan itself.
  - Each query plan have a counterpart execution plan
- **Execution Plan takes intermediate result and produces intermediate result (ResultTable)**
- **Execution Engine takes care of execution plans**
- **Produce final result and the SQL statements used**

Query execution



Query Plan Tree

- Query Plans is a **plan tree**.
- **Post-order traversal** to turn the tree to a list
- Easy to implement on a single-threaded process

Execution Engine

- **Stack-based Post order traversal**
  - Get the next execution plan, and operands needed. Different execution plans takes different number of operands(tables)
  - Fetch operand from stack
  - Call execution plan,
  - Put result back to stack

Execution Utility

- Contains methods to convert query into SQL statement
- Eg:
  - GetNodeBy: Get GID of a node by (its label, property and value…)
  - GetEdgeBy: Get Edge ID by the start/end/relation type/…

- Multiple conditions can be combined together to form a query.
  - GetNodeBy can be assigned both node label and node property as query condition at the same time

- Decouple data representation from data model.

Database Utility

- Execute the SQL query, and parse result into different types of Java Objects.
  - getAsList, getAsMap, getAsTable, getAsString…

- Record the SQL statements used.

Execution Plans

- **AllNodeScanExec(Plan)**
  - Get GIDs of all nodes and put them into result table. Returns a new result table.
  - Number of nodes is >2M, takes about 10s to finish, and consumes about 16M space
  - ResultTable is **huge**, not good for future plans

Execution Plans

- **ScanByIdExec(Plan)**
  - Get GIDs of nodes with some ID. Returns a new result table.
  - Only returns 1 result. Lookup is fast since ID is indexed in backend database.
    - **SQL: SELECT gid FROM P_id where value = SOME_ID**

Execution Plans

- **ScanByLabelExec(Plan)**
  - Get GIDs of all nodes with the label occurred in that plan. Returns a new result table.
  - Average number of nodes with same label is not small
  - Reduced intermediate result size.
  - Labels of all nodes are stored in **NodeLabel** table in MySQL.

Execution Plans

- **ScanByPropertyExec(Plan)**
  - Get GIDs of all nodes who has some specific value on a property. Returns a new result table.
  - Column-based storage, fast.
    - **SELECT gid FROM P_prop WHERE value = VALUE**
  - Slow on property tables that are not indexed

Execution Plans

- **FilterByConditionExec(ResultTable, Plan)**
  - Scan every item in result table, check if it has the value on the property to be checked. Drop those who does not have that value.
  - Supports inequtions.

Execution Plans

- **FilterByConditionExec(ResultTable, Plan)**
  - Two execution methods:
    - A. If result table is small, or dataset is large: Check every value in the previous result.
      (Can be concurrently executed)
    - B. If dataset is small but result table is large: Get all valid values from DB, and drop records with invalid values.
      (Reduce slow DB lookups.)

Execution Plans

- **FilterRelationEqualityExec(ResultTable, Plan)**
  - Scan every item in result table, check if the two variables satisfies the equation in the plan. Keep those who satisfy.
  - One scan through the table.
  - No DB lookups. (Same EdgeID, same edge.)

Execution Plans

- **ExpandAllExec(ResultTable, Plan)**
  - Get all edges of that expanded variable in previous Result Table.
  - Put then into a set.
    - Reduce DB lookups.
  - Get all edges that starts from those values(nodes). Add those edges and nodes into result table. Drop those who owns no such edges.
  - If there are conditions on the end of that edge, keep only edges who satisfy these conditions.

Execution Plans

- **ExpandAllExec(ResultTable, Plan)**
  - Expand operation in ResultTable:
    - (a)-[r1]->(b1), (a)-[r2]->(b2)… (a)-[r2]->(bn)
  - For each row, if value is a:
    - Concatenate all [r_i] ->(b_i) to that row.
    - Create n new rows in result.
  - Need two scan through the whole table.
    - One to get the values to be expanded
    - One to add new values.

Execution Plans

- **ExpandAllExec(ResultTable, Plan)**
- Prev:

A	R1	B
n11	r1	n21
n11	r2	n22
n12	r3	n23
n13	r4	n24

- Insert

A	R2	C
n11	r5	n31
n11	r6	n32
n13	r7	n33

Execution Plans

- **ExpandAllExec(ResultTable, Plan)**

- After:

A	R1	B	R2	C
n11	r1	n21	r5	n31
n11	r2	n22	r5	n31
n11	r1	n21	r6	n32
n11	r2	n22	r6	n32
~~n12~~	~~r3~~	~~n23~~		
n13	r4	n24	r7	n33

Execution Plans

- **ExpandIntoExec(ResultTable, Plan)**
  - If both ends of an edge are already in the result table, but we want the edge, we need ExpandInto.
  - Different execution methods considering the number of records in previous result.
  - Suppose expandInto operation is applied to
    (a) – [r] ->(b), where **(a)** and **(b)** are already found, and we need to expand **[r]**.

Execution Plans

- **ExpandIntoExec(ResultTable, Plan)**
  - A. If table is really small:
    - Check every a-b pair, get all edges from a to b.
    - Similar to ExpandAll.
  - **Small** : the execution time of performing #Small checks does not exceed the time to get all values in the edge table.

    (time to execute 3000 single query = time to query the whole table with 2M records)

Execution Plans

- **ExpandIntoExec(ResultTable, Plan)**
  - B. If table is larger, but not as large as the Edge table:
    - Get possible (a)-(b) pairs.
    - Put them into a set.
    - Find all edges by these pairs.
      - Edge table supports edge lookup by combining multiple conditions.
    - Rest of work is similar to ExpandAll
  - Much faster if many duplicated (a)-(b) pairs

Execution Plans

- **ExpandIntoExec(ResultTable, Plan)**
  - C. If table is larger than the total number of edges in dataset:
    - Get all edges from DB.
    - Construct (a)-(b) pairs of every record in the result table.
    - Get all edges starts from a and ends at b.
      - In-memory lookup. Can be faster by creating index on result table.
  - Cost of scanning result table is more than cost of getting all edges.

Execution Plans

- **CartesianProductExec(Table1, Table2, Plan)**
  - Cartesian product on two tables. Two tables have no common variables.
  - Nested for-loops.
  - Result is stored in table1, drop table2.
    - Saves memory space

Execution Plans

- **NodeHashJoinExec(Table1, Table2, Plan)**
  - Join on two tables with same variable or with equation between two tables.
  - Hash is calcualted by multiplication of hashcode() of all the join variables(common variables)
  - Hashed variables are GID of a node (as a string), or the string value of a Property Lookup.

Execution Plans

- **NodeHashJoinExec(Table1, Table2, Plan)**
  - Steps:
    - Calculate hash value of all rows in two tables.
    - Use a map to store the hash values of table1:
      - Key : Hash
      - Value: indices of records with this hash to the table1
    - Java Hashmap provides O(1) lookup.

Execution Plans

- **NodeHashJoinExec(Table1, Table2, Plan)**
  - Steps:
    - For each row in table2, get the hash value of that row
    - Lookup the hash in table1.
    - Perform Cartesian Product on rows with same hash. (If there are multiple rows with same hash value). Insert result into new table
    - Return new table.

Execution Plans

- **NodeHashJoinExec(Table1, Table2, Plan)**
  - Result Table implementation:

```
new_table = []
foreach row : table2
        hash = hash_func(row)
        if hash in table1.hashmap:
                indices = table1.hashmap.get(hash)
                table1_rows = table1.get_rows(indices)
                new_rows =
                        cartesian_product(row,
table1_rows)
                new_table.add(new_rows)
```

Execution Plans

- **RangeExpandAllExec(Table1, Plan, PathList)**
  - Looks for all possible variable length path expansions on the start node.
  - Get all conditions on edge variable and end node variable.
  - Get the starting nodes, and BFS on each of them searching for path, tracking passed nodes and edges.

Execution Plans

- **RangeExpandAllExec(Table1, Table2, Plan, PathList)**

```
queue <- []
result <- []
for i <- (0 to range.end):
    for each path in queue:
           if path length is within range:
                  if path ends at a valid node:
                         result <- path
    // BFS
    new_queue = []
    for path in queue:
           edges <- getNeighbors(path.end_node)
           for edge in edges:
                  new_path <- path + edge + edge.end
                  new_queue <- new_path
    queue = new_queue
```

Execution Plans

- **RangeExpandAllExec(Table1, Plan, PathList)**
  - Result is stored in pathlist, and kept globally.
  - Path is represeted by index to itself in the pathlist.
  - Cons: Saves space. Faster.
  - Pros: Path list is append only. No modifications.

Execution Plans

- **RangeExpandIntoExec(Table1, Plan, Pathlist)**
  - Similar to RangeExpandAll.
  - Difference is that end node should be in same as the end node in the previous result.

Execution Plans

- **ProduceResultExec(Table1, Plan, Pathlist)**
  - Produce the final result of the query.
  - A seperated List<Object> to store final results.
  - Expand intermediate result by its IDs.
    - Node: ExpandNode() to get all contents of a node by its GID.
    - Edge: ExpandEdge() to get edge Eid, start node GID and end node GID.
    - PropertyLookup: Just get the string
    - Path: Fetch the path by index to Pathlist.

Working example

Example

- **Query String:**

```
match
  (a)-[r]->(b),
  (c)-[r2]-(b),
  (a)--(c)
where
  b.id = "8"
return
  a.id, r, b.name
```

Example

- **Query String:**

```
match
  (a)-[r]->(b),
  (c)-[r2]-(b),
  (a)--(c)
where
  b.id = "8"
return
  a.id, r, b.name
```

Query Plan:

```
ScanById|b|
ExpandAll|b|-[r]-(a)
ExpandAll|b|-[r2]-(c)
FilterRelationEquality||r2 != r
ExpandInto|a|-[anonRelation0]-(c)
FilterRelationEquality||anonRelation0
 != r AND anonRelation0 != r2
ProduceResult|a,b|a:a.id b:b.name
```

Example

- **Query Result:**

…(many rows)

```
========================
0       {node2Label=4, eid=2, node2=2, rel_type=HashTagUsedBy, node1Label=3, relationship=null, node1=1}        ThinkBIGSundayWithMarsha
0       {node2Label=4, eid=2, node2=2, rel_type=HashTagUsedBy, node1Label=3, relationship=null, node1=1}        ThinkBIGSundayWithMarsha
2       {node2Label=4, eid=276, node2=2, rel_type=HashTagUsedIn, node1Label=2, relationship=null, node1=261}    ThinkBIGSundayWithMarsha
2       {node2Label=4, eid=276, node2=2, rel_type=HashTagUsedIn, node1Label=2, relationship=null, node1=261}    ThinkBIGSundayWithMarsha
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
4       {node2Label=4, eid=29270, node2=2, rel_type=hashTagComenation, node1Label=4, relationship=null, node1=3}        ThinkBIGSundayWithM
```

Example

- **SQL Statements**

…(many rows)

```
SQL:
========================
SELECT gid FROM P_id WHERE value = "8";
SELECT node1, eid FROM Edge WHERE node2 = "2" ;
SELECT node1, eid FROM Edge WHERE node2 = "2" ;
SELECT node2, eid FROM Edge WHERE node1 = "2" ;
SELECT node2, eid FROM Edge WHERE node1 = "1" ;
SELECT node2, eid FROM Edge WHERE node1 = "3" ;
SELECT node2, eid FROM Edge WHERE node1 = "261" ;
SELECT node1, eid FROM Edge WHERE node1 = "261" AND node2 = "1" ;
SELECT node1, eid FROM Edge WHERE node1 = "261" AND node2 = "3" ;
SELECT node1, eid FROM Edge WHERE node1 = "261" AND node2 = "261" ;
SELECT value FROM P_id WHERE gid = "1";
SELECT * FROM edge WHERE eid = "2";
SELECT value FROM P_name WHERE gid = "2";
SELECT value FROM P_id WHERE gid = "1";
SELECT * FROM edge WHERE eid = "2";
SELECT value FROM P_name WHERE gid = "2";
SELECT value FROM P_id WHERE gid = "261";
SELECT * FROM edge WHERE eid = "276";
SELECT value FROM P_name WHERE gid = "2";
```

Example

- **Query String:**

```
match (a)-[r]->(b), (c)-[r2]-(b),(a)--(c) where
b.id = "8"
return a.id, r, b.name
```

Future work

Future work

- **DB Lookup is expensive**
  - Some data can reside in memory.
  - Perform batch lookup, and use call-backs.
  - Add cache on Database Utility

- **Concurrency improves a lot.**
  - Concurrent execution on the execution plans.
    - Plans of same depth in the planning tree can be executed concurrently.
  - Concurrent execution on neighbor lookup in variable length expansion

Future work

- **Nodes are biased**
- (Kurant M, Markopoulou A, Thiran P. On the bias of BFS (breadth first search)[C]//Teletraffic Congress (ITC), 2010 22nd International. IEEE, 2010: 1-8.)
  - Most of nodes have low degree. But some have really high degree.
  - BFS is not efficient.
  - Apply parallel BFS algorithm.

- **Partitioning on database**
  - Connected graph
  - On biased nodes

Future work

- **Dataset is huge**
  - Lazy evaluation on expensive SQL queries
  - Column-based intermediate result table
  - Partitioning also helps.

- **Execution Plan optimization**
  - Reordering
  - Combining related plans
  - Intermediate result sharing

Thanks