PGQL Query Planner

Zanetti Andrea
Politecnico di Milano
School of Industrial and Information Engineering
andrea4.zanetti@mail.polimi.it

Srinivasan Ravi Francesco
Politecnico di Milano
School of Industrial and Information Engineering
ravifrancesco.srinivasan@mail.polimi.it

I. COST MODEL

Our operator's cost calculation is unique for each operator and is divided into two parts: *searchCost* and *filterCost*.

We assumed that the query engine using our optimizer performs neighbor access on an in-memory stored CSR[1]. We assumed that each operation performed on the CSR (e.g. access to one neighbor of a chosen vertex) costs one *cpuOperationCost*. The number of operations performed on the CSR depends on the cardinality of the previous operator. Once all indexes and vertexes needed are retrieved, the operators apply the constraint filters on the data returned by the previous operations.

The filtering cost is based upon the cost of accessing sequential/indexed vertexes/edges and of accessing vertexes/edges labels/attributes (we assumed that accessing the label of a vertex/index costs the same as accessing one attribute of a vertex/index). The total cost of one operator is the sum of *searchCost* and *filterCost*.

A. SEARCH COST

Each operator performs several operations on the CSR. Hence, the *searchCost* is computed by multiplying the number of operations by *cpuOperationCost*:

1) Constant Vertex Match and Root Vertex Match: Based on the given skeleton, we assumed that these two operators will have the same cost if the constraints on the vertices are the same. As these two operators do not perform any operation on the CSR (they only perform vertices access and vertices filtering of all the vertices in the graph), there is no cost associated with these types of operations.

2) Neighbor Match: The neighbor match plan contains a *boolean* variable called outgoing which indicates which type of neighbors to search. Let SELECT: (a)->(b) be a query, and assume we are performing a neighbor match on vertex (a):

- If *outgoing* is *true*, the operator is searching for the outgoing neighbors of (a), meaning the vertices that "receive" an edge from the vertex (a). The total number of operations performed by the operator is the parent cardinality[2] multiplied by the average vertex degree.
- If *outgoing* is *false*, the operator is searching for all the neighbors of (b) matching (a). Therefore, for each (b), the operator binary searches all (a)s in its neighbors. This means that for each b, the total number of operations is $C \log n$, n being the average vertex degree, C being the cardinality of the parent plan. Therefore, the total number of operations is $C \log n \cdot L$, L being the total number of vertices in the graph.

3) Cartesian Product: Our cartesian operator plan does not include constraints: the two parents return two sets of the indexes of the vertices to the cartesian product operator; the operator then performs a cartesian product between the two sets and returns all the ordered pairs. We assumed that for each ordered pair returned the operator performs one operation.

4) Edge Match: We assumed that this operator takes ordered pairs of vertexes as input (parent must be a cartesian product). Let ((a),(b)) be an ordered pair of vertices.

For each (a), the operator binary searches (b)s in the neighbors of (a), then filtering based on the edges' constraints. For each (a), the total number of operations is $m \log n$, n being the average vertex degree, m being the cardinality of the parent (number of ordered pairs). As this operation is performed for every ordered pair, the total number of operations is $m^2 \log n$.

5) Common Neighbor Match: We implemented the cost calculation only for a common neighbor with *leftOutgoing* and *rightOutgoing* both true. This means that our cost calculation is valid for queries like SELECT: (a)->(b)<-(c). Moreover, our operator plan does not take into account filtering on the edges, which could be implemented.

Our operator takes ordered pairs of vertexes (parent must be a cartesian product). When *leftOutgoing* and *rightOutgoing* are both true it performs a neighbor scan of all both vertices of all pairs. The total number of neighbors' accesses (operations) is two times (one for each element of one pair) the number of pairs, then multiplied by the average degree of a vertex (average number of neighbors for each vertex).

It then computes the number of operations performed by the binary search (we assumed that the operator binary searches the neighbors of the *rightQueryVertex* in those of the *leftQueryVertex*). The number of operations performed by the binary search is $n \log n$, n being the average vertex degree3. This binary search is performed one time for each vertex, meaning the total number of operations is $n \log n \cdot C$, C being the parent cardinality (number of ordered pairs).

The total number of operations is obtained by adding the binary searches cost with the neighbor access costs.

6) Reachability operator: We didn't implement this operator's plan.

[1] Compressed Sparse Raw matrix
[2] Size of the parent operator's result
[3] Average number of vertices' neighbors in the graph

B. FILTER COST

Before computing the cost of filtering the result, the operators compute the cardinality of the non-filtered result. This is done differently by each operator and depends on the type of operator, on the cardinality of the parent plan and the average vertex degree of the graph. This operation should be clear to understand from our source code. Once the non-filtered result cardinality is computed, the operator will calculate the cost of the filtering:

- Constant Vertex Match and Root Vertex Match: as these two operators operate on the entire graph, the non-filtered result cardinality corresponds to the total number of vertices in the graph. The operator accesses sequentially all the vertexes in the graph, the cost of this being *sequentialVertexCost* per read vertex.
- Other operators: as the operations on the CSR return to the operator a set of indexes, the operator accesses the non-filtered result elements using indexes. The cost per access is *indexVertexCost* (*indexEdgeCost* for the Edge Match operator).

For each element of the non-filtered result, every operator must then perform n accesses to the labels/attributes, n being the number of constraints, and filter them according to the constraint (the cost of this operation is *cpuOperationCost*).

All the above-mentioned operations must be performed m number of times, m being the non-filtered result cardinality.

II. GRAPH STATISTICS

A. INDEPENDENCY ASSUMPTION

Our cost model is strictly dependent on the operator's cardinality computation. The operator result's cardinality computation is done by multiplying the non-filtered operator's result cardinality by the operator's selectivity (S). With this term we intend the size of the operator result's cardinality (C_O) over the non-filtered operator's result cardinality (C_{NFO}):

$$S = \frac{C_O}{C_{NFO}}$$

In other words, selectivity measures how many items are selected from the non-filtered result, depending strictly on the operator's constraints.

In order to measure constraints' selectivity, we assumed that predicates on labels/attributes are independent. Let a graph database contain vertex representing cars, with attributes like "manufacturer" and "type". Assume for example that a manufacturer "X" produces only "SUV". If the manufacturer "X" selectivity is 1/10, and SUVs model selectivity is 1/100. Now consider a query 'SELECT (a) WHERE a.manufactorer = "X" AND a.type = 'SUV''. With the independence assumption, the total selectivity is:

$$S = \frac{1}{100} \cdot \frac{1}{10}$$

While the true selectivity is, in fact, much higher: 1/100. We explain a solution to this problem in paragraph F.

B. UNIFORMITY ASSUMPTION

We did not assume uniformity of label and attribute values: our model computes statistics regarding the data before and during the execution of the planner. As data size grows, estimating graph's property becomes a time-consuming process, often impacting the planner's efficiency. Complete data analysis for graph properties retrieval is very precise but very inefficient, particularly for:

- real-time databases: as data continues to change, it must be continually analyzed. This process can become a not insignificant overhead impacting the planner performances
- distributed databases: when data is distributed across multiple servers, running a complete analysis is a complicated process, implying moving big chunks of data across the network which usually is time and resources consuming

The solution we proposed is based on Yahoo! DataSketches [4] library. As we explain in the following paragraphs, this solution provides efficiency for both of the above-mentioned types of databases.

C. BUILD OF THE SKETCHES

First, our system evaluates the type of data by reading the first raw of the indicated CSVs (one for edges and one for vertices). By reading the first raw, it decides whether a column contains numerical values or not (assuming that non-numerical values are strings). Then it builds sketches reading one column at the time according to the following:

1) *Numerical values:* for this type of values it builds a *quantile sketch*

2) *String values:* for this type of values it builds a *theta sketch* and a *frequent item sketch*

The dimensions of the sketches can be set from the settings package, please refer to DataSketches documentation to understand the best settings that suit your needs. During the reading of the CSV tables, it also saves the length of the tables. At the end of this process, sketches are saved as *.bin* data, as well as a *.json* file which contains the sketches path, the number of vertices and the average vertex degree. This process has to be done only one time per graph.

D. STATISTICS COMPUTATION

During startup, the planner loads the data contained in the sketches and in the *.json* in memory for fast access. It is then ready to compute statistics on the planner's demand. Selectivity is computed differently for numerical and string values:

1) *Numerical values:* the planner builds histograms and searches for the requested values in the histogram for retrieving selectivity for this type of data (comparators can also be used as allowed by histograms). Selectivity corresponds to the probability associated with a given value in the histogram.

[4] Documentation can be found on Yahoo! DataSketches website

2) String values: the planner searches for the required value in the most frequent item sketch, which contains the counts of a chosen number of items (can be set through settings package). Values not belonging to the most frequent items are assumed to be uniformly distributed in the remaining space. Constraint selectivity is then computed by dividing the value count by the total number of vertex/edges.

We also allow the user to select the estimate error bound by allowing the set of an error bound parameter in the setting package (please refer to DataSketches documentation).

E. DATASKETCHES ADVANTAGES

The usage of DataSketches library entails a number of advantages compared to other methods for selectivity computation:

1) DataSketches is fast: sketches sizes are customizable by the user, and are usually kilobytes in size, orders-of-magnitude smaller than required by the exact solutions, allowing an impressively fast computation of selectivities, which is also independent of the size of the data.

2) DataSketches is precise: we tested the precision of sketches created for the IMBD database provided. Our tests were performed on a laptop (Asus UX430UN) running Ubuntu. First, we performed 10000000 selectivity estimations on the selectivity of the filter "person" in the "type" column. On average, the computation of the selectivity of this filter took <u>25 nanoseconds</u>, returning a result with a precision of 10^4 compared to the selectivity computed by parsing the entire CSV. We then tested the selectivity of numerical values. On average, the computation of the selectivity of this filter took <u>800 nanoseconds</u>. This difference in time is due to the fact that our system creates a histogram every time the quantileSketch is accessed (histograms can be also saved for future usage, drastically decreasing the time of the computation; please refer to DataSketches documentation). Due to the type of data contained in the database, we only tested the selectivity of "role" column, computing 10000000 times the selectivity of the value "1". The result was returned with a precision of 10^4 compared to the selectivity computed by parsing the entire CSV.

3) DataSketches works on data streams: sketches are streaming algorithms, in that they only need to see each incoming item only once. This is a great advantage in real-time databases where data is continuously moving in and out of the database.

4) Sketches are mergeable: the input data can be partitioned into many fragments, also across different machines. Once all sketches are created, they can be merged in milliseconds without accuracy loss. This allows for the usage in real-time and distributed databases: data doesn't need to be scanned every time data is updated, which would take a significant amount of time for big databases.

F. POSSIBLE IMPROVEMENTS

Improvements can be made in order to make estimates more precise. Contrarily to what we assumed, in most cases, predicates on labels/attributes are not independent. This assumption can lead to underestimating the true cardinality, resulting in massive blowup during execution [1] [2]. This problem can be solved by building a *Bayesian network*, a *moral graph*, and a *junction tree* on the graph. Attribute selectivity can be then calculated using propagation algorithms on the *junction tree* [3].

III. HEURISTICS

We decided to implement a selectivity estimation heuristic [4] , which is based on the parameters calculated from the cost model that we created. The implementation is based on the following pseudo-code:

Algorithm 1

```
FUNCTION subPlan(plan.lastVert, verticesLeft)
    FOR (V in verticesLeft)
        FOR op IN operators
                selMap.put(op(V), op(V).sel)
    orderMap(selMap)
    FOR key IN orderMap with val within 10% diff:
        remove key.v from verticesLeft
        plan.lastVert.addChild(op(v))
        subPlan(plan.lastVert, verticesLeft)
```

Algorithm 2

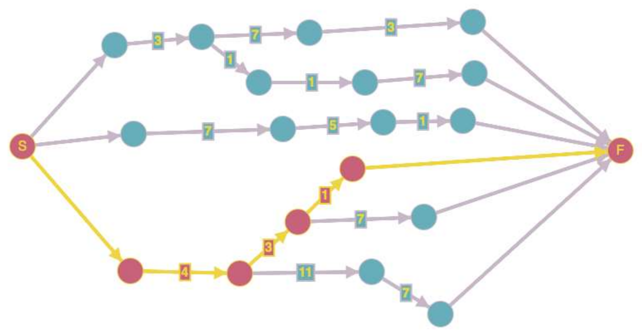
```
FUNCTION generatePlan(GraphQuery query)
    constantVertices <- number of c.v. in query graph
    rootVertices     <- number of r.v. in query graph
    IF constantVertices > 1
        plan         <- findOptimalConstVertexOrder()
    ELSE
        plan         <- findMaxRootVertexSel()
    subplan(plan.lastVert, verticesLeft)
    Run Dijkstra on plan to find the shortest path
```

A. ALGORITHM

First, if there are 1 or more constantVertices, generatePlan (Algorithm 2) looks for the optimal order in which to apply the ConstantVertexOperators, in order to get the maximum selectivity. Else it finds the RootVertex applied to the vertex which gives maximum selectivity. Then there's a call to subPlan (Algorithm 1), passing the last Vertex added to the Plan and the Vertices left to analyze. Abbreviations and Acronyms

In subPlan (Algorithm 1) we combine every Vertex left with all the 6 operators insert the couple operator-vertex associated with their selectivity in a HashMap. Now if we find that the operations with maximum selectivity have a similar selectivity (within 10% difference) we branch by creating a parallel execution plan in the graph by recursively calling subPlan (Algorithm 1) for each similar operator to apply. At the end of the recursive calls, generatePlan (Algorithm 2) runs Dijkstra on the graph to find the optimal execution path from the graph created (lower cost). An example of the final graph is shown in Figure 1.

Figure 1



B. TIME COMPLEXITY

The heuristic is characterized by the following Time Complexity:

$$T(N) = O(N \cdot \log(N))$$

where N represents the number of Vertices in a given Query.

One of the important aspects of such a heuristic is computation time and quality flexibility based on current needs at execution time. In fact, by adjusting the similar selectivity parameter (now set in the heuristic at 10%) we can decide by how many branches we want to create, to have a longer planning time but a more optimal result or shorter planning time and a result of minor quality.

IV. Heuristics – A quantum approach

Lately, Quantum Computers based on Quantum Annealing (such as D-Wave's) are becoming more and more relevant in the High-Performance Computing scenario. By exploiting properties of Quantum mechanics such as quantum tunneling these computers can easily solve optimization problems by finding the global minimum of an energy function. The only effort to be made is to map a general problem to an energy minimization problem. First, we need to find a penalty function in the QUBO formulation (Quadratic Unconstrained Binary Optimization) which minimum value corresponds to the configuration we want to find. A lot of High-Performance Computing problems regarding graphs can be solved using this solution:

Data clustering, structural balance and community detection in signed graphs [5], link prediction [6] [7], network evolution [8] and node classification [9].

For this project it comes really useful the use of Quantum Annealing to find the shortest path in a graph using Dijkstra [10], in case the number of Vertices in our previously shown graph becomes too big to handle. This would let us find the shortest path in microseconds instead of milliseconds or even seconds.

Unfortunately, the number of qubits in a D-Wave QPU is still very limited, ~2038 qubits. This means that we are barely on the edge of being able to have a real advantage in applications over classical approaches. But in the near future (approx. Q3 2020) D-Wave will release its new 5000 qubits QPU which will bring a real advantage in using Quantum Computers.

V. References

[1] W. Cai, M. Balazinska, and D. Suciu, "Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities," 2019, doi: 10.1145/3299869.

[2] V. Leis TUM, A. Gubichev TUM, A. Mirchev TUM, P. Boncz CWI pboncz, cwinl Alfons Kemper TUM, and T. Neumann TUM, "How Good Are Query Optimizers, Really?," 2150.

[3] K. Tzoumas, A. Deshpande, and C. S. Jensen, "Lightweight Graphical Models for Selectivity Estimation Without Independence Assumptions," 2150.

[4] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds, "SPARQL basic graph pattern optimization using selectivity estimation," in *Proceeding of the 17th International Conference on World Wide Web 2008, WWW'08*, 2008, pp. 595–604, doi: 10.1145/1367497.1367578.

[5] E. Zahedinejad, D. Crawford, C. Adolphs, and J. S. Oberoi, "Multi-Community Detection in Signed Graphs Using Quantum Hardware," Jan. 2019.

[6] D. Liben-Nowell and J. Kleinberg, "The link prediction problem for social networks," in *Proceedings of the twelfth international conference on Information and knowledge management - CIKM '03*, 2003, p. 556, doi: 10.1145/956863.956972.

[7] K. Y. Chiang, N. Natarajan, A. Tewari, and I. S. Dhillon, "Exploiting longer cycles for link prediction in signed networks," in *International Conference on Information and Knowledge Management, Proceedings*, 2011, pp. 1157–1162, doi: 10.1145/2063576.2063742.

[8] C. Aggarwal and K. Subbian, "Evolutionary network analysis: A survey," *ACM Computing Surveys*, vol. 47, no. 1. Association for Computing Machinery, 2014, doi: 10.1145/2601412.

[9] J. Tang, C. Aggarwal, and H. Liu, "Node classification in signed social networks," in *16th SIAM International Conference on Data Mining 2016, SDM 2016*, 2016, pp. 54–62, doi: 10.1137/1.9781611974348.7.

[10] C. Bauckhage, E. Brito, K. Cvejoski, C. Ojeda, J. Schücker, and R. Sifa, "Towards Shortest Paths Via Adiabatic Quantum Computing," 2018, doi: 10.1145/nnnnnnn.nnnnnnn.