EDS take home exam

2IMD10 Engineering Data-Intensive Systems

J. L. J. M. Serlier
1645757

I. Question 1

There is no doubt that world's data ingestion increases exponentially and, therefore, there is an urgent need for scalable data processing platforms. However, there is a big question on how to scale? Should we scale-up by adding more cores to a single machine or scale-out by adding more machines? Both approaches have their advantages and shortcomings. Which approach should be taken in the context of graph query processing? Present your argument by considering latest research in query processing, emerging hardware trends, and common workloads in different application domains.

When considering question of scale-up (vertical scaling) vs. scale-out (horizontal scaling) in traditional relational database environment, it seems that industry leaders in cloud storage solutions (Azure, AWS, Google Cloud Services) push their customers toward an approach where scaling out is the prominent answer. Arguments follow that scaling up is seen as trying to solve problems by 'buying a bigger box', but outward scaling is flexible and fits modern architecture paradigms better. However, both approaches do see their fit, depending on the client's needs. For example, Yap (a solution architect at AWS) argues that horizontal scaling works well for distributing read-heavy oriented databases as seen in figure 1, whereas up-scaling will be more beneficial when on-demand resources are needed on a single instance [1]. Modern techniques in cloud computing use a combination of scale-out and scale-up techniques in combination with flexible auto-scaling and load balancing depending on on the need for resource availability and performance [2].

Considering the same question for graph query processing specifically, the academic discussion on scale-out versus scale-up has, interestingly enough, been led by two parties from the University of Waterloo with conflicting views. In the paper *Scale Up or Scale Out for Graph Processing?* [3], Lin steadfastly argues scale up is the sensible viable option for graph processing, unless distributed solutions are unequivocally necessary to the organisation. His argument stems from his experience as an Engineer at Twitter where a 'simple' scale-up solution worked as the increase size of the graph dataset would not outpace Moore's Law (in this case loosely used as a term for computer capabilities). When met with the seemingly valid argument that industry

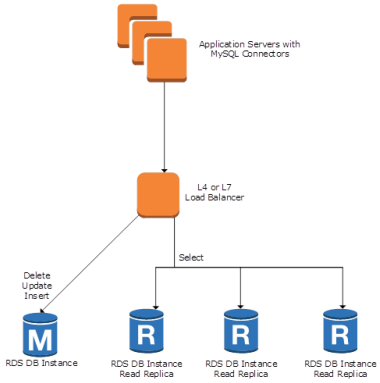

Fig. 1: Modern solution to load balancing EC2 instances using a scale-out method to increase availability [1]

hasn't considered or implemented scale-up as enthusiastically as they do so for scale-out, he argues that this is due to the echo-chambering effect of buzzwords in the likes of "scale-out is the only way" and "no single point of failure".

A few months later a response followed written by Salihoglu and Özsu [4], colleagues of Lin from the University of Watlerloo's Data Systems Group. Lin's colleagues agree with him that for many organizations running analytical calculations on a single machine (scaled-up) is more practical than running a distributed system. After this statement of agreement, their sentiments quickly diverge. Salihoglu and Özsu follow up with arguments considering common three cases where scale-out is a more practical and rational choice for scaling graph analytics. The first two fall within the same argument: graphs can naturally exceed the size which Lin considered in his anecdote and paper. This is either due to the nature of their domain ('Trillion-edge-size graphs' such as those in finance, e-commerce, ...) or due to the fact that extracted graphs are usually part of a much larger graph. The third case mentions that more complex computations necessitate scale-out, as even though the initial graph can fit in main memory of a single machine, the computations require multiple machines at work.

An illustrative example of the extent of the graph size issue is something which the authors call the 'blow-up' factor. Although the raw dataset might be relatively small, when the data is processed overhead is created in RAM due to

the internal data structures. Figure 2 shows the results from experimentation performed in the response paper.

Dataset	No. of vertices	No. of edges	Raw data	PowerLyra (single machine)
Live Journal	4.8 M	68.9 M	1.08 Gbytes	6.30 Gbytes
USA Road	23.9 M	58.3 M	951.00 Mbytes	9.09 Gbytes
Twitter	41.6 M	1.4 B	26.00 Gbytes	128.00 Gbytes
UK0705	82.2 M	2.8 B	48.00 Gbytes	247.00 Gbytes
World Road	682.4 M	717.0 M	15.00 Gbytes	194.00 Gbytes

Fig. 2: Graph size 'blow-up' factor after loading in [4]

Conclusion. Although the authors from both sides disagree on many points, the arguments where they meet common ground shows the nuance which is essential in order to answer the scale-up versus scale-out question. The basic requirements in modern (cloud-) computing solutions are accessibility, reliability and scalability. If the graph size of your application is able to be loaded in a single machine, as well as allow for enough room for the complex computations to be performed, scale-up is the way to go in terms of scalability. Lin is correct in mentioning Moore's Law in this sense, as the rate of growth of the graph data should remain less than the rate of growth 'capacity' of the computers which process them. When running on a single machine, one should take into account that accessibility and reliability are still key. For many, many organizations this means that scale-up is a reasonable, viable and cost-efficient solution for approaching the scaling question in regards to graph query processing. Industry's focus towards scale-out can be essential to solve practical problems in terms of reliability and accessibility, as well as scalability, but at the cost of more complex architectures and increased financial costs. For a small portion of organization, mainly data-driven organizations where accessibility to data and query information is key, scaling up is an approach which should be taken seriously. As both academics and industry leaders keep innovating in this field, scale-out is becoming a more viable and cost-efficient solution. Finally, organizations for which the graph-size is simply too large, or the 'blow-up' factor and computations too complex to be (practically) dealt with on a single machine, will have to consider scaling out as their approach of choice.

II. Question 2

Consider the general problem of estimating the number of distinct tuples obtained after applying a projection operator $\Pi X(R(Y))$ *on an arbitrary table* $R(Y)$*, s.t.* $X \subseteq Y$*. How would you perform such an estimation? Give an estimation approach and its analysis.*

Cardinality estimation is key for cost-based query optimizing. By efficiently being able to estimate the resulting size of a query, one is able to selectively order predicates when comparing execution plans of a query. For this problem, we are specifically interested in finding a cardinality estimation for a projection. In the relatively old but still relevant paper *On Estimating the Cardinality of the Projection of a Database Relation* [5], R. Ahad attempts to tackle this problem by means of incorporating data semantics.

In the paper, they compare their work to that of Merrett and Otoo [6]. Merrett and Otoo estimate the cardinality of the output by means of modeling the distribution of the tuples in the relation in n dimensional space, where n is the amount of tuples. Their key assumptions are that tuples are assumed to be uniformly distributed in the sector space shown in figure 3, and that tuples sampled from the population are selected without replacement.

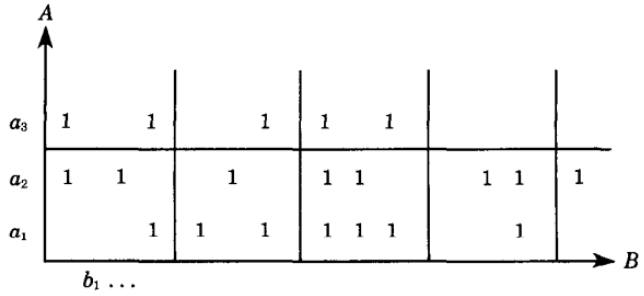

Fig. 3: Tuple distribution of R(A, B) [5]

Using this distribution model, the expected number of tuples of projection over B of the set of tuples R_b whose base is some sector b of B is:

$$\mathbf{E}\left(|R_B[B]|\right) = n\left(1 - \prod_{i=1}^{|R_b|} 1 - \frac{k}{nk-(i-1)}\right) \quad (1)$$

Where n is the number of values per sector and k is the number of A values.

R. Ahad continues improves upon this estimation by assuming certain a-priori knowledge of the relationships in the relational database is available. Altough our problem settings mentions we apply the projection over an arbitrary table, gaining a-priori knowledge could feasibly be done on arbitrary tables, for instance using sampling methods. An notable improvement is that the new formula does not require a scan over the relation. An assumption is made that there exist no duplicate tuples in the original relational model. Altough duplicity in the original relation is technically allowed, it will make the estimation less accurate. A flaw in the formulated equation was later removed in *A Note on Estimating the Cardinality of the Projection of a Database Relation* [7]. Therefore, I will describe the improved formula:

Let R'(A, B) be a relation with attributes A and B. We use $|Dom(A)| = m$ and $|Dom(B)| = n$, and assume (our a-priori-assumption) that for any R', the distinct values $a \in A$ where also $a \in B$ is p, and the distinct values $b \in B$ where

also $b \in A$ is q. Then let Q be the unary relation of k distinct A values. Finally, let R be the natural join of Q and R‘. Then we have:

$$\mathbf{E}_s\left(|R[B]|\right) = n\left(1 - \frac{\binom{m-p}{k}}{\binom{m}{k}}\right) \tag{2}$$

As mentioned, for our problem, we consider applying a projection on an arbitrary table. Therefore, we cannot conclusively say we have any a-priori knowledge regarding the relational data. In many cases however, we are able to use sampling and statistical techniques to gain a-priori insights which are needed for the assumptions of p and q in the improved estimator. A practical approach to this can be found in in *Random Sampling for Histogram Construction: How much is enough?* [8]. By accurately using these histograms, a query plan optimizer can make better choices for the execution plan. Statistics from these histograms can also be used as the a-priori knowledge required for the improved formula 2. Various other methods are also possible to efficiently used samlping and statistical methods in order to gain knowledge about the relations an arbitrary table, for instance those described in the book Database Management Systrems [9]. Lastly, I wanted to mention that a paper by Naughton and Seshadri which show a more algorithmic approach to solving our problem [10], which is interesting for practical scenario’s but a bit out of scope for our question.

Conclusion. Using either of the two formula’s, an effective cardinality estimation of a projection on an arbitrary relation can be performed. This allows us to compare execution plans and thereby improve query evalutation times. If we have no a-priori Merrett and Otoo’s formula can be applied. Alternatively, using a-priori beforehand knowledge or sampling methods, the improved version of the algorithm (equation 2) can be used which is slightly more accurate under certain conditions. Although these formula’s are quite old, they are not dated as they are still used in practice today. Therefore, they are a fitting solution to our problem.

III. Question 3

*Consider building a database for (a better version of) G**gle Maps. Consider a spatial user query in which you are required to return top-K points-of-interest (POIs) nearest to the current (given) location. Describe how would you design such a database and how would you efficiently execute such queries? I am looking for time/memory efficient solutions.*

I will answer this question by proposing multiple solutions and comparing their advantages and disadvantages in terms of time and space complexity, as well as general feasibility.

First, let’s consider some assumptions; on a small-scale, a spatial datapoint is just an ’x’ and ’y’ value on which you can index your results. As you consider a larger area, the curvature of the earth is not negligible. Two datapoints then represent a curved line in 3-d space. There are a multitude of ways to deal with this, but the easiest is to just stick with existing google’s datalayer conventions [11] and assume we use longitude and latitude. Therefore, I make the assumption that longitude and latitude data is processed with any of the existing frameworks before the spatial query is evaluated in the database.

Taking this into account, my first intuition was to make sure that a user’s query was ’tackled’ in a divide-and-conquer approach, as this will make the runtime of the query evaluation logarithmic (assuming the data is stored in a tree-like format). I discussed this approach with a colleague studying Geomatics at TU Delft who recommended the unpublished book *Computational modelling of terrains* [12]. In chapter 10, the construction of a kd-tree is discussed as an efficient option to tackle spatial nearest neighbour queries. Interestingly, this is able to deal with 3 dimensional queries as well as the 2 dimensional queries we are interested in. By traversing the tree depth-first and using the tree properties, large parts of the tree can be eliminated. For a **single** nearest neighbour query, the closest point to the user is simply stored in a temp variable C_{temp}. Expanding this to an **N-nearest neighbour query**, we can simply maintain the N closest points instead. During the traversal, the order is that of the N ”most promising” nodes, at each iteration updating the C_{temp_i} nodes using the Euclidean distances where needed. Other sub-trees should also be considered and are visited recursively, and trees are eliminated as soon as the distance of the furthest point in C_{temp} is further away than the ’bounding box’, which are indicated as the semi-squares in the figure 4. This figure shows the several states a kd-tree (with k=2) goes through when finding a single nearest neighbour.

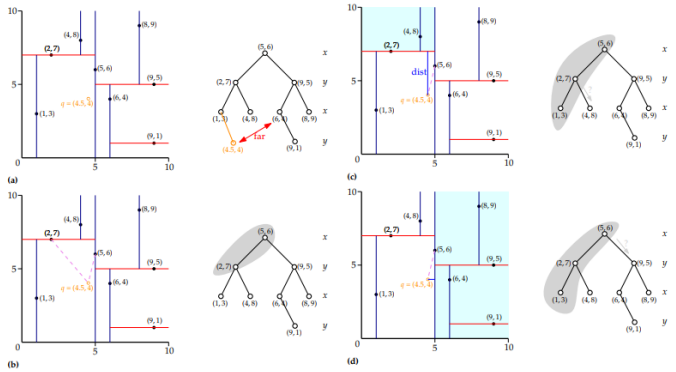

Fig. 4: States of finding a single nearest neighbour in in a kd-tree ($\mathbb{R}^2$), order: a - d [12]

The construction of the kd-tree can be done efficiently in real-world scenario’s by creating a balanced tree (desirable due to quicker searches) using the estimated or sampled median of a subset (1% is mentioned in the book) of the graph to split the nodes in the graph. The tree is built by inserting a node one at the time by traversing the tree based on the splitting dimension, in this case the spatial index.The time complexity of this algorithm is $O(logn)$ for insertion

and lookup, and the space complexity is $O(n)$.

The highly influential paper *Nearest neighbor queries* by Roussopoulos et al. proposes the use of R-trees in order to find the (k) nearest neighbours to an object and introduces a branch-and-bound traversal algorithm for solving our problem. An R-tree groups nearby objects within a minimum bounding rectangle, which are used to define the tree hierarchy structure. This is visualized in figure 5. In comparison to the kd-tree previously discussed, an R-tree is disk-oriented, meaning they map their data to a disk representation which makes them more practical in real-life scenario's. Huang et al discusses how to optimize storing such trees for dynamically indexed spatial databases [13]. The compact R-tree can achieve high levels of storage utilization while still maintaining all relevant properties. The R-tree and its variants allow for efficient query lookup and insertion (average $O(log_m n)$ and worst case $O(n)$), and a space complexity of $O(logn)$ [14]. As R-Trees are balanced, they are more suitable for cases where we would need to keep adding new points of interest. In general, the literature seemed to indicate that R-tree's and its variants are most suitable for real-world applications for a K-nearest-neighbour problem.

Conclusion. We have discussed two promising data-structures which allow to efficiently execute spatial queries where a user requests the top-k points of interest near to the user. These can both be used as the foundational design of our databases. To fully answer the question we should consider how these data-structures can be applied in order to efficiently execute the spatial user queries. As R-trees have been shown to be more practical, I will consider this data-structure. Practically designing a database for the use of R-trees and querying is straightforward. Various cloud providers offer services where R-trees can be used as the foundation, such as IBM's Informix service, offering a fully working R-tree implementation. This service only has 20 bytes overhead for each page (e.g. a bounding-box or a range of POI's) of the R-tree (leaf, branch or root) [15]. Such a service could be the foundation of our database. Other optimization methods could also be considered, for instance caching certain pages that contain data-points from a certain area closer to the user. For instance, users from the EU might have pages containing POI's near them cached in a datacenter close to the user (e.g. in Frankfurt). This concludes the efficient design of a top-K points of interest database.

IV. Question 4

Consider a simple shortest-path reachability query $Q(s,t)$. Evaluation of Q on a directed graph G ($[[Q]]_G$) returns the length of a shortest path p if t is reachable from s by following p in G, and returns "-1" otherwise. How would you perform evaluation $[[Q]]_G$ efficiently given $Q(s,t)$ and G, both in time and in space? I am looking for solutions faster than an online search algorithm like Dijkstra's and more memory efficient than offline computation of the whole (shortest-path) transitive closure of G. Consider the latest research on this topic, justify your solution and document all of your assumptions

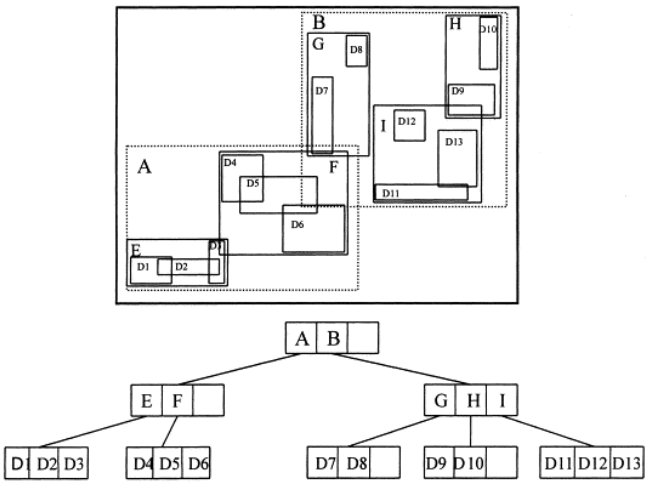

Fig. 5: Hierachy of spatial objects in an R-tree showcasing the bounding rectangles [13]

As I have personal experience in dealing with a similar problem, I will start off by discussing my experience in dealing with Dijkstra's optimizations and I how tackled it. Afterward, I will dive into more theoretical (academic) as well as modern practical approaches. The assumptions I make for answering this question is that I will not consider purely 'scaling' optimizations such as parallelizing, since I feel like this is not in the spirit of the question. Lastly, I will use the term 'user' as an entity which requests queries, but this can be interpreted as any entity which requires the query result.

The efficiency of the implementation of Dijkstra's algorithm and optimizing it is a billion dollar industry question. Interestingly enough, the algorithm itself is quite optimal at a worst-case performance of $\Theta(|E| + |V| \log |V|)$ in terms of its vertices V and edges E. Even still, there are ways that for real-world applications this can be practically sped up. The problem at hand simply asks whether or not two nodes of a query $Q(s,t)$ are connected, and if so, what the length of the shortest path is. The simplest way to investigate whether or not two vertices are connected is to construct the transitive closure of the graph. The transitive closure of a graph is a graph itself which contains edges from each node to every node which it is able to reach via existing edges in the original graph, making it useful for this scenario. However, this is quite computationally heavy. The Floyd-Warshall algorithm for computing the distance between all pairs of vertices has similar benefits and issues, as it allows for pre-computing the shortests paths efficiently but at a high upfront computational cost. Since we will not be computing the full transitive closure or all shortest paths

of the graph, I will instead answer the question by looking at methods to improve shortest path querying.

For my Bachelor's final project, I had to tackle a relevant problem to this question. Considering I was working together with researchers from TU-Delft, I believe my approach is justifiable up-to-date. I was investigating the effect of timing of disruption events on the London underground network. This means I had to compute a large amount of shortest paths between all the vertices in the network for various scenario's. The assumptions I made was that a disruption event would effectively 'remove' and edge from the network, and passengers would follow the next-available shortest path which allowed them to reach their final destination. This approach allowed me to estimate the increased strain on particular parts in the network as disruption events occur throughout the day. In case you are interested, you can find more about the project on my website. As I was investigating the impact over all the edges in the network, I would have to calculate an enormous amount of shortest paths if I did think about optimizations. Since I was running the computations on an AWS EC2 instance, I was eager to reduce the computational and financial costs of this. Apart from the relatively uninteresting approach of multi-processing, I found out that caching the shortest paths of the undisrupted (full) network worked quite well. After a disruption event, I simply had to check whether or not a shortest path between two vertices contains the disrupted edge and recompute that shortest edge if that was indeed the case. Disrupting edges with a low betweenness centrality (a centrality measure based on how many shortest paths contain that edge) therefore required very little additional computation.

Practical approach for the shortest-path reachability problem: For our problem, we are not looking to pre-compute all shortest paths of the network, but a similar approach of caching could be very beneficial. I suggest first trying to gauge which queries will come up often, which is of course heavily dependent on the use case of the graph database at hand. For instance, we could generate plausible queries and cache N amount of shortest paths. These would be indexed on the vertexes, sorting and keeping track of the amount of times they are retrieved, starting at 0. We can then use this as a baseline 'cache' storage. As actual queries start being requested, we can lookup in the cache to see if the index $Q(s,t)$ exists in the cache. If not, we compute the shortest path and add it (or "-1" if there exists no shortest path) to the cache, as long as there is still space in the cache left for a new path. It is important to keep in mind what happens if user behavior changes, meaning they will start requesting different queries. In case the frequent queries change and we have a 'dropout' threshold in the cache based on the amount of retrievals, newer queries that might be relevant to users will not be cached. Therefore, to counteract this, the bottom Z cached paths should be dropped periodically. The choice of the number of saved shortest paths N and the dropout Z are values which need to be experimented with, but coming up with a reasonable number based on the graph size and the query frequencies is straightforward.

The increased efficiency of the approach above is mainly dependent on the frequency distribution of the queries which are requested. If the distribution is heavily skewed in favor of certain queries, this approach becomes more efficient. Although the cache C is sorted by request frequency, the worst runtime complexity of checking if a path is cached for a vertex pair could reasonably be assumed to be $O(N)$ where N is the cache size, and will be quicker for queries which are more frequent. However, this will be practically negligible compared to computing the shortest path again in a practical scenario. This means that for $Q(s,t)$ where the pair $(s,t) \in C$ the runtime will be $O(N)$, and for $(s,t) \notin C$ it will be $O(|E| + |V| \log |V| + N)$. Assuming certain queries are more popular than others, this is a significant improvement in terms of computational complexity compared to online Dijkstra's, as well as to computing the transitive closure.

Most of what I have written above is from personal experience and intuition and will serve as a practical approach to solving the problem. I will finish this section by performing a broef literature study about other methods used in industry and academics. In *TEDI: Efficient Shortest Path Query Answering on Graphs* [16] a method is considered where the shortest path query problem is sped up by using TEDI: Tree Decomposition based Indexing. Although this is an interesting and practical approach, one effectively has to pre-compute every shortest path in the decomposed bags of trees. In *Optimizing Dijkstra for real-world performance*, Aviram and Shavitt implement a Que implementation which speeds up the runtime of Dijkstra's in real-world and synthetic graphs substantially. Using their method, they bound the runtime to $O(E + U)$ where U is the distance of the vertex farthest from the starting vertex. When we have usable heuristics about the graph, we can use the A* algorithm instead of Dijkstra's algorithm, which is a classic goal-directed searching algorithm [17].

Lastly, The survey *Route Planning in Transportation Networks* [18] is a very in-depth practical survey from engineers from Industry and academics showing many interesting methods for solving shortest-path queries in a practical real life setting. Unfortunately, I only found this source very late, as it is exceptionally thorough and in-depth in it's many optimizations which could be useful for our problem. Examples are:

1) Precomputing distances within clusters in the graph which are separated by 'cut arcs'. This works on the principle that partial Dijksta's can be used to construct a full shortest paths for certain (sub)-graphs (image 6).
2) Separator-band techniques which can be used to preserve 'shortcuts' in a subgraph over which Dijkstra's can be

run

3) Hierarchical techniques which assume that the further away one is from the source and target nodes, the less vertices it has to consider in the (sub)graph. This is a very effiecent technique for certain real-life graphs, but it does not always return the optimally shortest path.

The paper goes on to showcase various methods to implementing these and other techniques, as well as ranking them on in a practical manner.

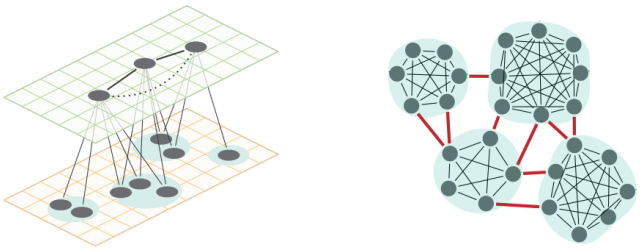

Fig. 6: Left: Multilevel overlay graph with two levels. The dots depict separator vertices in the lower and upper level. Right: Overlay graph constructed from arc separators. Each cell contains a full clique between its boundary vertices, and cut arcs are thicker. From: [18]

.

Conclusion. For this question, I have given you insights coming from my personal experience in dealing with the shortest path problem. The in-depth method I have detailed is based on an approach used for a previous research project, and should fit the constraints quite well of being more efficient than pure Dijkstra's and more memory efficient than the full Transitive Closure. In addition, I have detailed some of the literature I came across which showcase both academic and practical solutions to similar problems which could be used as effective means to solving the problem of the question.

V. Question 5

The problem of subgraph isomorphism is known to be NP-complete. NP-complete problems are considered to be intractable (of course, up until somebody shows P = NP). Yet, "efficient" practical solutions for "real-world" workloads exist for subgraph query matching, e.g., by employing query languages like SQL, SPARQL, and Cypher. What is the trick, then? Please carefully explain what is going on.

The graph isomorphism problem is one of the more prominent NP-complete problems in computer science. It gained traction in the 1950's for matching molecular graphs, and has become more and more relevant in computer science as we have become dependent on solving complex computations in large graph data-structures and databases. In this section, I will cover the basic theory behind graph isomorphisms, after which I will dive deeper into the practical side of things; how is the graph isomorphism solved in real-world scenarios?

Two graphs $G = (V, E)$ and $H = (W, F)$ are said to be isomorphic if there exists a mapping of any of the vertices in V to the vertices in W such that $\forall v, w \in V$, with $f : V \to W$ we have $\{v, w\} \in E \iff \{f(v), f(w)\} \in F$. This means adjacency is preserved for all vertices after the mapping. This is visualized in figure 7. Only recently in 2016, Babei showed that isomorphic testing can actually be performed in quasipolynomial time, with the time complexity being $N^{P(logn)}$ where n is the number of vertixes in the graph and P is some polynomial [19]. In a review article of the graph isomorphism problem, Grohe and Schweizer interpret this as "being almost efficiently solvable— theoretically" [20]. However, how is this problem tackled in real-world scenarios?

It is key to understand that the complexity of the graph isomorphism problem is heavily dependent on the type of graph one is dealing with. For instance, Hopcroft and Tarjan showed early on that a planar graph, meaning a graph that can be drawn on a surface without intersecting edges, can be solved in $O(nlogn)$ time. A similar feat was shown in the guest lecture of this course given by George Fletcher, where, for acyclic cases, we are able to solve the query problem in polynomial time. This means that a subset of the graph isomorphism problem are cases which are tractable.

Diving deeper into the tractability of graph isomorphims, Gottlob et al. discusses methods of identifying so called 'islands of tractability' for problems considering their graph topology [21]. As previously stated, acyclic structural problem are tracable, but ones which are nearly acylic can also be dealt with using structual decomposition methods, specifically the hypertree decomposition method. A (hyper-)tree decomposition maps a graph into a tree, where the tree width is the minimum width of any tree decomposing the graph. An example is shown in figure 8. Using this decomposition allows for solving certain computational problems on a graph much more efficiently. Among the points in made in their conclusion, the one which stands out is the comment underlining the ubiquity of real-life problems which show sparse tree-like structures with dense local cyclic structures. Here lies the key in answering the question at hand: one can project the local cyclic clusters into subproblems which are relatively easily to solve on their own. Research on this subject is still very much ongoing. For instance, in 2020, Ganian et al. introduces a new method of bounding treewidth of hypertrees allowing for fixed parameter tractability [22].

Conclusion. Although the graph isomorphism problem is NP-complete, depending on the shape of the underlying graph, the problem is practically tractable in many cases. Real-life graph datasets often have sparse tree-like structure with local cyclic clusters. Using various decomposition techniques, solving this problem for "real-world" workloads is feasible

and practical. For instance, in *H-DB: a hybrid quantitative-structural sql optimizer* [23] it is shown that structural decomposition can be used for optimizing SQL queries for (near-)acyclic queries. In a paper written by Mhedhbi and Salihoglu it is shown that tree decompositions are used in order to optimize queries for Graphflow DBMS, which supports a subset of the Cypher language [24]. In a more academic setting, Wang et al. showed that subgraph matching on large RDF graphs is possible using a star-based decomposition method [25]. Conclusively, for the graph isomorphism problem in relation to real-life query languages, it is often possible to transform the problem in one for which we can create a much tighter bound while still being able to achieve the desired results of the query.

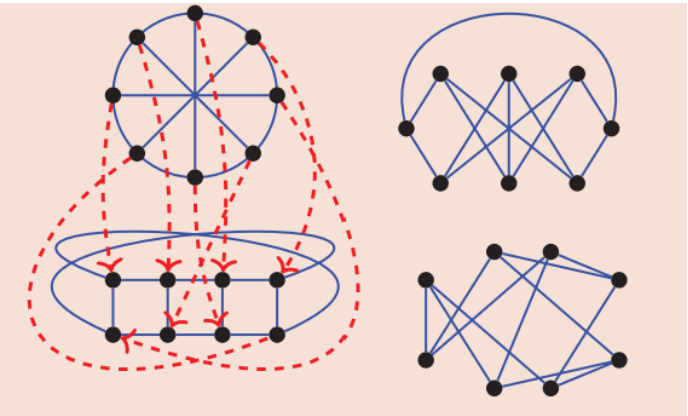

Fig. 7: Four isomorphic graphs shown by the ability to remap vertexes while maintaining adjacency [20]

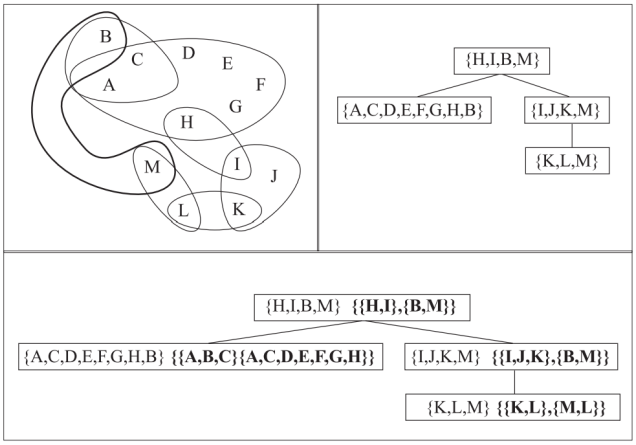

Fig. 8: A hypergraph H, a tree decomposition for its primal graph, and a width-2 hypertree decomposition for H [21]

References

[1] Yap. Scaling your amazon rds instance vertically and horizontally, 2016.

[2] Kai Hwang, Yue Shi, and Xiaoying Bai. Scale-out vs. scale-up techniques for cloud performance and productivity. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 763–768. IEEE, 2014.

[3] Jimmy Lin. Scale up or scale out for graph processing? *IEEE Internet Computing*, 22(3):72–78, 2018.

[4] S. Salihoglu and M. T. Özsu. Response to "scale up or scale out for graph processing". *IEEE Internet Computing*, 22(5):18–24, 2018.

[5] Rafiul Ahad, K. V. Bapa, and Dennis McLeod. On estimating the cardinality of the projection of a database relation. *ACM Trans. Database Syst.*, 14(1):28–40, March 1989.

[6] TH Merrett and Ekow Otoo. Distribution models of relations. In *Fifth International Conference on Very Large Data Bases, 1979.*, pages 418–425. IEEE, 1979.

[7] Ravi Mukkamala and Sushil Jajodia. A note on estimating the cardinality of the projection of a database relation. *ACM Trans. Database Syst.*, 16(3):564–566, September 1991.

[8] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. Random sampling for histogram construction. *ACM SIGMOD Record*, 27(2):436–447, June 1998.

[9] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw-Hill, 2000.

[10] Jeffrey F. Naughton and S. Seshadri. On estimating the size of projections. In Serge Abiteboul and Paris C. Kanellakis, editors, *ICDT '90*, pages 499–513, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[11] The google maps data layer. https://developers.google.com/maps/documentation/javascript/datalayer, 2020.

[12] Ravi Peters Hugo Ledoux, Ken Arroyo Ohori. Computational modelling of terrains. https://github.com/tudelft3d/terrainbook/releases, 2020.

[13] Po-Whei Huang, Phen-Lan Lin, and HY Lin. Optimizing storage utilization in r-tree dynamic index structure for spatial databases. *Journal of Systems and Software*, 55(3):291–299, 2001.

[14] Rajendra Prasad Mahapatra and Partha Sarathi Chakraborty. Comparative analysis of nearest neighbor query processing techniques. *Procedia Computer Science*, 57:1289–1298, 2015.

[15] Ibm informix r-tree index user's guide, Apr 2008.

[16] Fang Wei. TEDI. In *Advances in Data Mining and Database Management*, pages 214–238. IGI Global.

[17] Akshay Kumar Guruji, Himansh Agarwal, and D.K. Parsediya. Time-efficient a* algorithm for robot path planning. *Procedia Technology*, 23:144–149, 2016. 3rd International Conference on Innovations in Automation and Mechatronics Engineering 2016, ICIAME 2016 05-06 February, 2016.

[18] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. *CoRR*, abs/1504.05140, 2015.

[19] László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, page 684–697, New York, NY, USA, 2016. Association for Computing Machinery.

[20] Martin Grohe and Pascal Schweitzer. The graph isomorphism problem. *Commun. ACM*, 63(11):128–134, October 2020.

[21] Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. Treewidth and hypertree width. *Tractability: Practical Approaches to Hard Problems*, 1, 2014.

[22] Robert Ganian, André Schidler, Manuel Sorge, and Stefan Szeider. Threshold treewidth and hypertree width. In *IJCAI*, 2020.

[23] Lucantonio Ghionna, Gianluigi Greco, and Francesco Scarcello. H-db: A hybrid quantitative-structural sql optimizer. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, page 2573–2576, New York, NY, USA, 2011. Association for Computing Machinery.

[24] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins, 2019.

[25] Xin Wang, Lele Chai, Qiang Xu, Yajun Yang, Jianxin Li, Junhu Wang, and Yunpeng Chai. Efficient subgraph matching on large RDF graphs using MapReduce. *Data Science and Engineering*, 4(1):24–43, March 2019.