Data Bases for Big Data
Analysis of Apache Spark and Application to Graph Processing

January 2026

University of Primorska, FAMNIT; Dmytro Tupkalenko (89252101); GitHub repository

1 Introduction

This report presents an overview of Apache Spark's architecture and its application to computational graph processing, specifically the identification of highly irregular graph patterns. The work is structured in two main parts: a theoretical examination of Apache Spark and its graph processing components, followed by a practical application demonstrating distributed graph analysis.

The first part introduces Apache Spark, outlining its core architectural principles, the Resilient Distributed Dataset (RDD) abstraction, and some components including Spark SQL, GraphX, and GraphFrames; and provides all of the necessary references for further reading.

The second part focuses on a concrete implementation that uses Spark's distributed computing capabilities to solve a graph-theoretical problem: identifying highly irregular graphs and detecting their occurrences as subgraphs within larger network structures. Highly irregular graphs represent a special class of graphs where all vertices have distinct degree sequences among their neighbors, making them valuable for real-world applications. The implementation is a complete pipeline consisting of two main processing phases: classification of candidate graphs from an exhaustive search space to identify all highly irregular graphs for a given degree, and subgraph isomorphism detection to locate these patterns within target networks.

2 Apache Spark

2.1 Purpose and Overview

Apache Spark is a unified analytics engine for large-scale data processing [1], designed to address the limitations of earlier big data processing frameworks, in particular Apache Hadoop MapReduce [2]. Developed initially at UC Berkeley's AMPLab in 2009 and later donated to the Apache Software Foundation in 2013, Spark has become one of the most widely adopted frameworks for distributed data processing. The primary motivation behind Spark's development was to overcome MapReduce's inefficiency in iterative algorithms and interactive data mining tasks. MapReduce requires writing intermediate results to disk between each computation stage, leading to substantial I/O overhead. In contrast, Spark keeps data in memory across operations, achieving performance improvements of up to 20x for certain workloads [3]. Spark is written on Scala, and the code is compiled to JVM.

2.2 Architecture of Apache Spark

Core Components

As the foundation of the entire platform, Spark Core provides essential services including task scheduling and distribution, memory management, fault recovery mechanisms, and interaction with various storage systems. Central to this infrastructure is the Resilient Distributed Dataset (RDD), which serves as the fundamental programming abstraction in Spark [3]. An RDD is an immutable, partitioned collection of records that can be operated in parallel across a cluster. These datasets are characterized by several key properties: immutability ensures that once created, an RDD cannot be modified but only transformed into a new dataset; partitioning allows data to be divided and distributed across nodes; and lazy evaluation ensures that transformations are not executed until an action requires a specific result. Furthermore, RDDs achieve fault tolerance through lineage, maintaining a record of their derivation graph to recompute lost data partitions without the need for replication. The execution of operations is coordinated by the Driver Program, which runs the `main()` function and creates the SparkContext, while Executors - worker processes on cluster nodes - run the assigned tasks and store data. The resource allocation for these processes is handled by a Cluster Manager.

High-Level APIs and Libraries

Spark SQL [4]   Spark SQL provides a DataFrame API designed in the similar to Pandas or R DataFrames, offering integration with RDDs and support for multiple data sources such as Parquet, JSON, Hive, and JDBC. Spark SQL is a successor of Shark [5] - the initial framework designed to provide SQL capabilities within the Spark ecosystem. A central feature of this library is the Catalyst optimizer, which enhances performance through columnar storage support and automated code generation for efficient execution. The Catalyst optimizer functions by performing several distinct optimization passes, beginning with an analysis phase to resolve references and type checking, followed by logical optimization techniques like predicate pushdown, constant folding, and projection pruning. Then, it handles physical planning to select the most efficient operators and concludes with code generation to produce Java bytecode.

GraphX [6]   GraphX serves as Spark's specialized API for graph processing and graph-parallel computation, extending RDDs through a property graph abstraction where both vertices and edges have user-defined properties. Structurally, GraphX represents graphs using a dual-RDD approach: a Vertex RDD containing unique vertex IDs and their associated properties, and an Edge RDD containing source and destination identifiers along with edge-specific data. The library provides built-in tools for graph construction and transformation operators - such as `mapVertices` and `mapEdges` - alongside structural operators like `subgraph`. Furthermore, it includes join operators for merging graphs with external RDDs, the integration with Pregel API for iterative computation, and a library of built-in algorithms including PageRank, Connected Components, and Triangle Counting.

GraphFrames [7]   GraphFrames builds on top of GraphX and SparkSQL by introducing a DataFrame-based API for graph processing, which allows for declarative graph queries using SQL-like syntax. This allows graph operations to benefit from query optimization via the Catalyst engine and ensures compatibility with standard Spark SQL operations and pattern matching for motif finding. Similarly as the RDD-based approach of GraphX, GraphFrames represents graphs using two DataFrames: a Vertices DataFrame, and an Edges DataFrame. This structure supports analytical algorithms, such as Label Propagation and Strongly Connected Components Identification.

2.3   Architectural Features

Execution Model

Spark's execution is described by a directed acyclic graph (DAG) model which specifies the flow of data processing. This process begins when the driver program creates a logical DAG of operations based on the user's code. The DAG scheduler then intervenes to break this graph into distinct stages, specifically at shuffle boundaries where data must be redistributed across the cluster. Each stage is comprised of individual tasks that can be executed in parallel, which the task scheduler subsequently distributes to the available executors (by default Spark uses Pull-based approach, but push based can be customly enabled). Finally, the executors perform the actual computation of these tasks and return the results back to the driver or write them to storage.
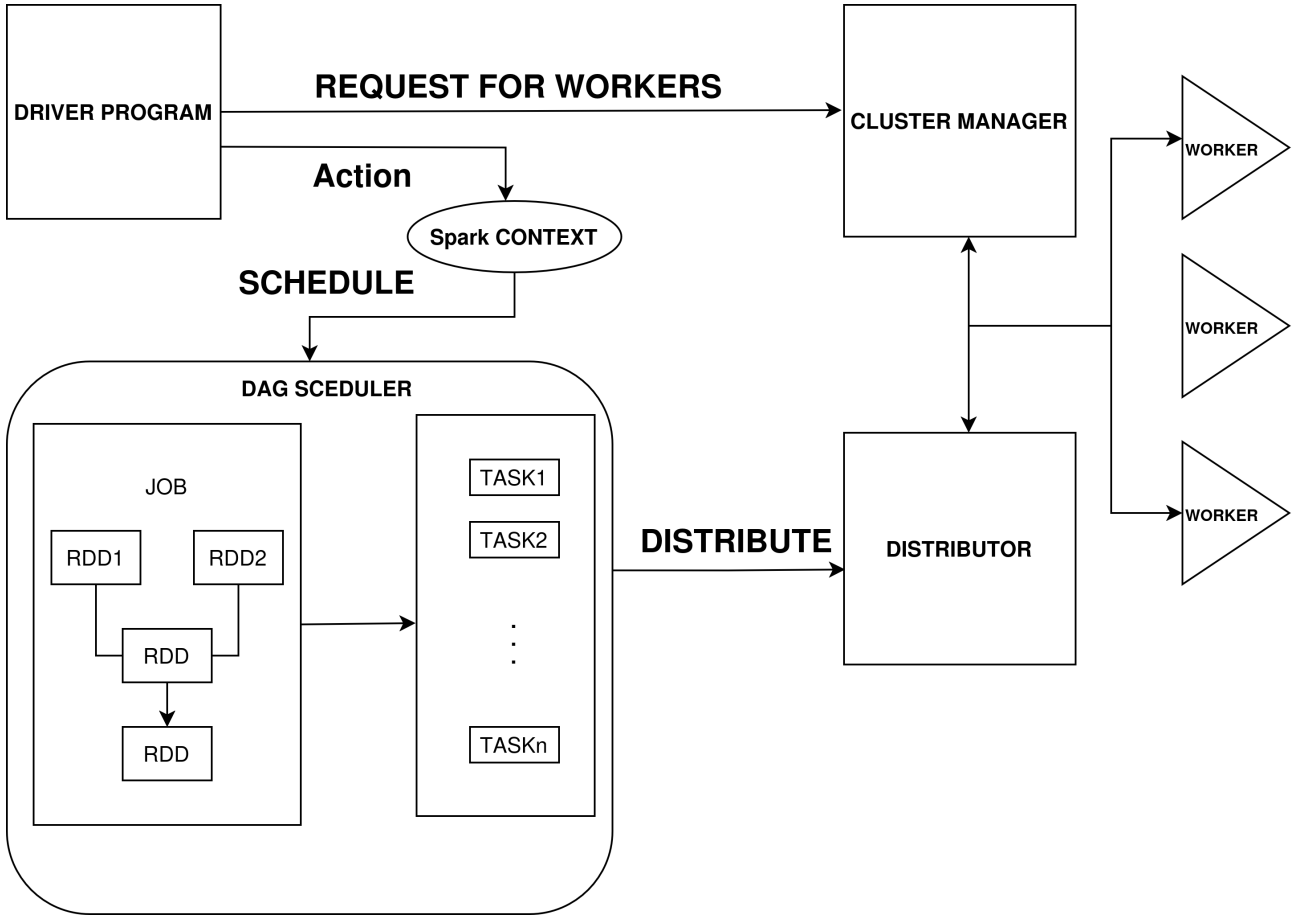
Memory Management

Memory within a Spark executor is divided into several regions. Storage memory is for the cached RDDs and broadcast variables. Execution memory handles the requirements of shuffles, joins, sorts, and aggregations. Additionally, user memory is set aside for custom user data structures and functions, and a small portion of reserved memory is maintained for internal Spark metadata. In Spark 1.6, there was and advancement in this point where the unified memory management was introduced - which allows for dynamic borrowing between the storage and execution regions.

Shuffle Operations

Shuffle operations is a component of Spark's architecture, that performs the redistribution of data across partitions, which is a requirement for operations such as `groupByKey`, and joins. Spark's implementation of the shuffle process is multifaceted, it uses a hash-based shuffle for standard operations and a sort-based shuffle for more memory-efficient processing. Additionally, Spark supports an external shuffle service that allows for better resource management by using compression techniques to minimize the impact of network I/O during the data transfer.

2.4 Diagram

The diagram summarizing general execution model of Pure Spark at a high-level:

3 Application: Highly Irregular Graph Identification

3.1 General Context

Highly Irregular Graphs   A graph $G = (V, E)$ is defined as highly irregular if for every vertex $v \in V$, all neighbors of v have distinct degrees [8]. Formally:

$$\forall v \in V : |\{d(u) : u \in N(v)\}| = |N(v)|$$

where $N(v)$ denotes the neighborhood of vertex v and $d(u)$ represents the degree of vertex u. This property creates a strong structural constraint that is both mathematically interesting and computationally challenging to verify at scale. Highly irregular graphs have applications in Network topology design, Error-correcting codes, Chemical graph theory, Social network analysis.

The graph can be classified for this property in a linear in the number of edges time.

Induced Subgraph Isomorphism   The induced subgraph isomorphism asks whether a pattern graph P appears as a subgraph within a larger target graph T. Formally, we look for an injective function $f : V(P) \to V(T)$ such that:
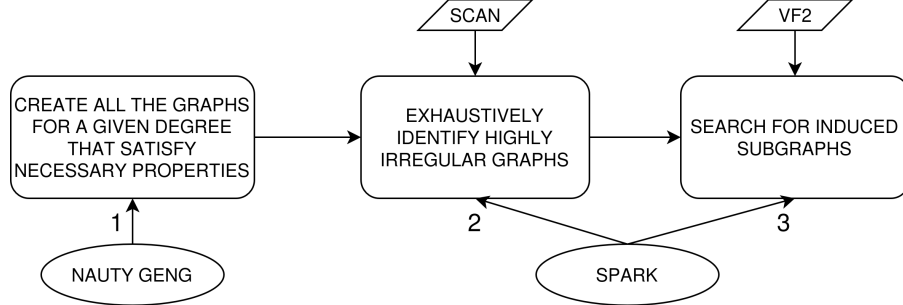
$$\forall (u, v) \in E(P) : (f(u), f(v)) \in E(T)$$

This problem is NP-complete.

3.2 Problem + Proposed Approach

The problem that we solve with our application can be described as follows: *Given a degree of Highly Irregular graphs of interest, and a Target graph - Identify all occurrences of all Highly Irregular graphs in the Target graph as an induced subgraph.*

The approach that we went with can be summarized with this diagram:

That is we have 3 phases: first we generate all of the graphs for a given degree that satisfy properties that are necessary for the graph to be highly irregular (connected, maximal degree, number of edges, ...) [8], second we perform a scan on the generated graphs with the custom function that checks if the graph is highly irregular, third we run VF2 algorithm (algorithm for induced subgraph identification) [9]. With this processing pipeline we achieve that the end user needs to provide just a target graph (large network) and a degree of interest - and we will be able to produce all the matches for all highly irregular graphs of a given degree within the given target graph.

Tool for graph generation called Nauty Geng was used to generate the graphs for a given degree (1st phase), Spark was used for the 2nd and 3rd phases. We will now describe the 2nd and 3rd phases in more details. The first phase is omitted, because it is not related to Spark - what we need to take from it, is that after this phase we have all graphs needed for exhaustive search in parquet format.

Phase 2: Classification

In general terms, this process applies a boolean classification flag to every row in the dataset. This represents a pure "inter-parallelization" workload: the table partitions are distributed across nodes, processed in parallel, and then collected. This phase relies primarily on the DataFrames API from Spark SQL to manage the distribution and execution of the classification logic.

Algorithm 1 Pseudocode Representation:

1: **Input:** Parquet files containing graphs of order n that satisfy necessary properties
2: **Output:** Classified highly irregular graphs
3:
4: Load graph data as DataFrame with schema (graph6, order, edges)
5: Apply UDF to filter highly irregular graphs
6: **for** each partition **do**
7:     **for** each graph in partition **do**
8:         Parse edge string into adjacency structure
9:         Compute degree sequence for all vertices
10:         **for** each vertex v **do**
11:             Get degrees of neighbors: $D_v = \{d(u) : u \in N(v)\}$
12:             **if** $|D_v| \neq |N(v)|$ **then**
13:                 **reject** graph (not highly irregular)
14:             **end if**
15:         **end for**
16:         **accept** graph (highly irregular)
17:     **end for**
18: **end for**
19: Transform accepted graphs into vertex/edge tables
20: Write results to Parquet format

Phase 3: Identification

Conversely, this phase requires a more complex "intra-parallelization" workflow. We initially operate on the target graph using the built-in GraphFrames Connected Components algorithm, which parallelizes computation across the graph's structure. Then, we use Spark SQL to filter components smaller than the pattern graphs. Finally, we use RDDs to parallelize the isomorphism checks, processing independent component-pattern pairs concurrently (we decided to go with Cartesian Product, because the number of patterns is very small (e.g. for n=10, the total number of connected graphs is 739310 Ref., whereas the number of highly irregular graphs is 13 Ref.).

Algorithm 2 Pseudocode Representation

1: **Input:** Pattern graphs P, Target graph T
2: **Output:** Isomorphism mappings
3:
4: Load pattern graphs into JGraphT structures
5: Load target graph as GraphFrame
6: Extract connected components using GraphFrames algorithm
7: Filter components with $|V| \geq |V(P)|$
8:
9: Create component RDD: $C = \{(id, V_c, E_c)\}$
10: Create pattern RDD: $P = \{id, JGraph)\}$
11: Compute Cartesian product: $C \times P$
12:
13: **for** each $(component, pattern)$ pair in parallel **do**
14:     Construct JGraphT graph from component
15:     Initialize VF2 isomorphism inspector
16:     Find isomorphic mappings
17:     **if** mapping found **then**
18:         **yield** $(component\_id, pattern\_id, mapping)$
19:     **end if**
20: **end for**
21: Collect and save results to Parquet

3.3 Deployment Details + Version Outline for Reproducibility

The application was built primarily on Scala 2.12.19 and Apache Spark 3.5.1, Python 3.12 + Nauty Geng 2.8081 was used for Graph Generation, GraphFrames 0.8.3 was used for distributed graph analysis and JGraphT 1.5.2 for local $VF2$ isomorphism logic. The nodes in our case had different OS (Linux, and Windows + WSL), that is why the system was deployed via Docker Swarm. All the data was stored in Apache Parquet format. Performance hyperparameters were tuned to balance coordination overhead with throughput: the driver was allocated 2GB of memory, while executors were configured with 1500MB and a default parallelism of 2x(number of cores). To maintain stability during iterative GraphFrames operations, a checkpoint directory was set (this makes the lineage chains smaller - increased I/O overhead, but more stable. Input datasets were partitioned into Parquet batches each containing approximately one million graphs.
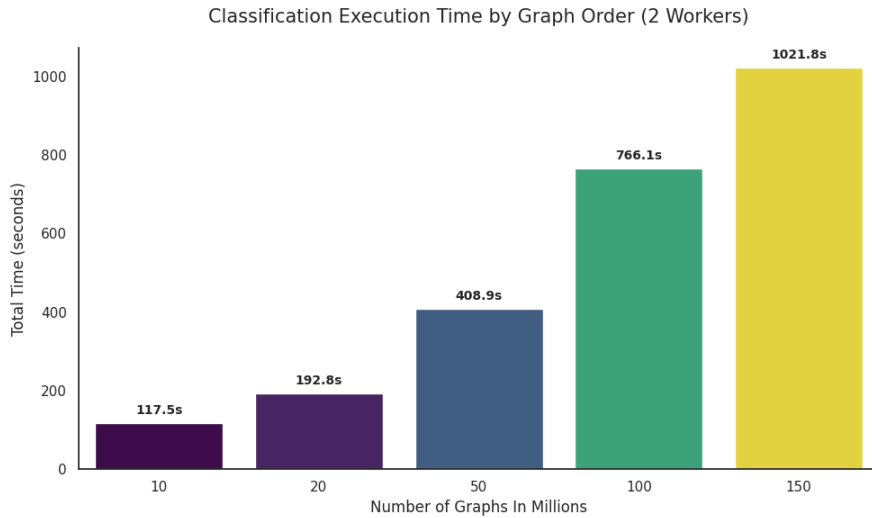
3.4 Experiments

The first experiment we conducted was micro-benchmarking on the amount of RAM allocated per executor for our setup. For this we measured time to process 20 million graphs with varying amount of memory: 500MB, 1500MB (Total amount of RAM available in our setup 24GB / Number of Executors (16) - (Total number of cores = 6 + 2) · 2), and 4GB. We got the following results:

Executor Memory Configuration	Total Execution Time (s)
500 MB	247.15
1500 MB (Optimal)	**186.17**
4 GB	309.28

Clearly, the 1500MB was the optimal amount, so we then used this configuration to conduct all of the remaining experiments.
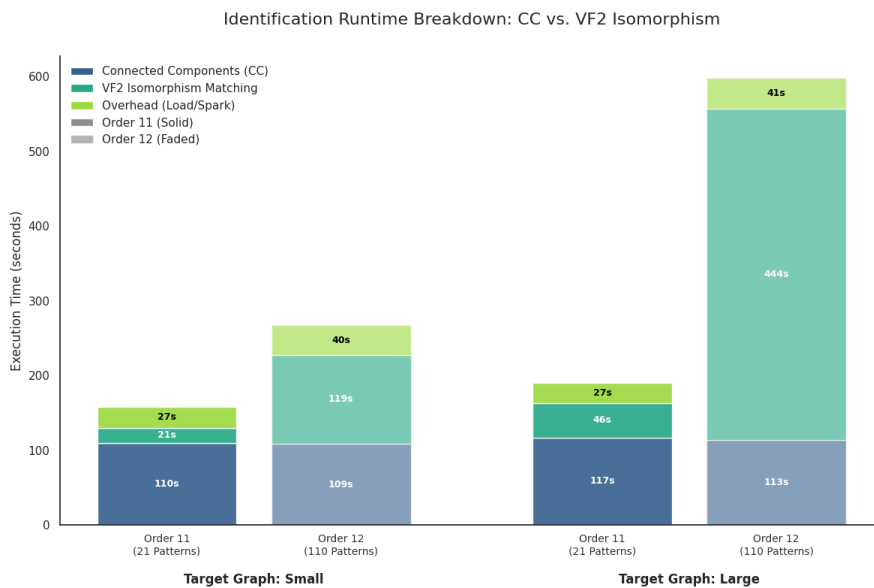
Classification Performance (Phase 2)

We evaluated classification performance across 10, 20, 50, 100, 150 millions graphs, the results can be seen on the following visualization:

Classification Execution Time by Graph Order (2 Workers)

The observation that we can draw from this plot is that the timing grows better than linearly (smaller than the ratio between the number of graphs), meaning that as the amount of data increases - the spark process overhead pays-off more, that is the overhead of using Spark decreases compared to the distributional processing speedups - the results are completely as one would expect, but what it means for us is that we configured Spark in a correct and optimal way.

Identification Performance (Phase 3)

Subgraph identification was tested on synthetic target graphs using Barabasi-Albert model [10] with varying number of connected components and number of vertices per component and with 2 orders of Highly Irregular graphs 11: (21 highly irregular graph) and 12: (110 highly irregular graph). Due to limited computational power, the syntetic target graphs we came up with are: **small** (1 connected component with approximately 100 vertices) and **large** (5 connected component with approximately 500 vertices). The sizes of the graphs are not big - nonehow comparable with the sizes of networks that are currently processed in the domain, but with this experiment we wanted to discover how processing of this phase scales. The results are presented on the following visualization:

Identification Runtime Breakdown: CC vs. VF2 Isomorphism

What we can directly interpret from the plot within each group (small, large) is that the overhead of spark + loading grows slower than linear - 5 times more patterns to load yields only 1.5 time increase. Then, the Connected components identification is also very efficient in the sense that 5 times more components to identify yielded less than 10% overhead. There is nothing we can directly draw from the VF2 performance because the algorithm has worst-case time complexity of $O(V! \cdot V)$. Overall, what we can takeaway from this experiment is that the parts that directly used Spark (VF2 was taken from external library) work optimally and as expected within our system.

4 Conclusion

In conclusion, this report has outlined the architectural principles of Apache Spark and showed its application to a combinatorial problem in graph theory. We successfully established a distributed pipeline that is able to identify highly irregular graphs and detect their occurrences as induced subgraphs within target networks. By using the Spark ecosystem we were able to transition a task from discrete mathematics into a parallelized workflow. The results of experiments confirm the possibility of this type of processing in a scalable manner.

Personal Comments   Regarding Spark in general, my experience with Spark was very positive, there are a lot of resources I could study the components from, the documentation was very helpful. The ability to work in Python was beneficial in the beginning, at first I wanted to implement everything in Python, but I then Switched to Scala because Python API (PySpark) did not have all the features I needed - and most of the things I just directly translated from Python to Scala, which saved a lot of time.

I never had any experience with distributed computations, so in general, almost everything was new to learn. I also mentioned that I used Docker - it proved to be a very good tool to use, because I tried to configure workers without it, but they had different OS, so it was quite hard and in the end I did not even manage to do it in a proper way. But once I switched to Docker - it just worked. I also tried to use as many Spark Components as I could, and mainly included everything I intended to use, except GraphX - but I did not need it anywhere. There were also some dependencies issues, that is why for example I did not use the latest version of Spark.

Regarding experiments, at first I wanted to test the pipeline on some real-world network, but in the end it was computationally impossible because of the VF2 - there is a combinatorial explosion and it was not possible even in terms of computational time, but in terms of space I needed for storing the matches - so I just included the micro-benchmark I used just to determine the optimal configuration and experiments on synthetic data.

References

[1]   Matei Zaharia et al. "Apache Spark: a unified engine for big data processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.

[2]   Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2010. URL: https://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf.

[3]   Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2012, pp. 15–28. URL: https://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf.

[4]   Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 1383–1394. URL: https://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf.

[5]   Reynold S Xin et al. "Shark: SQL and rich analytics at scale". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 2013, pp. 13–24.

[6]   Joseph E. Gonzalez et al. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014, pp. 599–613. URL: https://amplab.cs.berkeley.edu/wp-content/uploads/2014/02/graphx.pdf.

[7]   Ankur Dave et al. "GraphFrames: An Integrated API for Mixing Graph and Relational Queries". In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems (GRADES)*. ACM, 2016. DOI: 10.1145/2960414.2960416. URL: http://dx.doi.org/10.1145/2960414.2960416.

[8]   Yousef Alavi et al. "Highly Irregular Graphs". In: *Journal of Graph Theory* 11.2 (1987), pp. 235–249. DOI: 10.1002/jgt.3190110214.

[9]   Luigi P. Cordella et al. "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.10 (2004), pp. 1367–1372. DOI: 10.1109/TPAMI.2004.75.

[10]   Albert-László Barabási and Réka Albert. "Emergence of Scaling in Random Networks". In: *Science* 286.5439 (1999), pp. 509–512. DOI: 10.1126/science.286.5439.509. URL: https://www.science.org/doi/10.1126/science.286.5439.509.