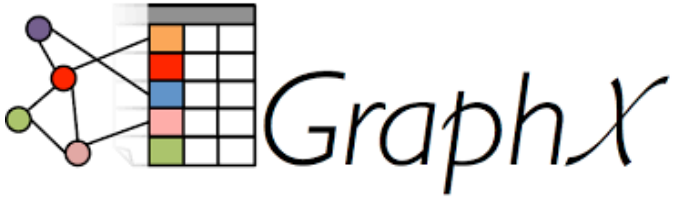
BIG DATA ANALYTICS (CS-431)

Dr. Sriparna Saha
Associate Professor

Website: https://www.iitp.ac.in/~sriparna/
Google Scholar: https://scholar.google.co.in/citations?user=Fj7jA_AAAAAJ&hl=en
Research Lab: SS_Lab
Core Research AREA: NLP, GenAI, LLMs, VLMs, Multimodality, Meta-Learning, Health Care, FinTech, Conversational Agents

TAs: Sarmistha Das, Nitish Kumar, Divyanshu Singh, Aditya Bhagat, Harsh Raj



Apache Spark GraphX: The Theory of Graph-Parallel Computation

A foundational look at how Spark processes graph-structured data at scale.

The Need for Graphs

The Limitation of Tables: Traditional databases store data in tables (rows and columns). This model is highly efficient for storing and retrieving structured records but struggles to represent complex, interconnected relationships.

The Problem with Joins: Answering relationship-based questions like "Find all friends of my friends" in SQL requires multiple, computationally expensive `JOIN` operations. As the depth of the relationship increases (e.g., "friends of friends of friends"), the queries become incredibly complex and slow.

Asking the Right Questions: Graphs are purpose-built to answer questions about connections, such as:

- "What is the shortest path between these two points?"
- "Who is the most influential person in this network?"
- "How are these two entities indirectly connected?"

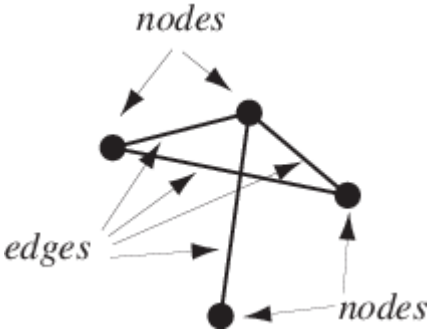
A Natural Fit: For data where the relationships are as important as the entities themselves (like social networks, supply chains, or financial systems), the graph model is the most natural and efficient solution.

What is a Graph?

Core Concept: A graph is a fundamental data structure designed specifically to model the connections between entities. It provides an intuitive way to visualize and analyze interconnected data.

Primary Elements: Every graph is composed of two simple elements:

- **Vertices (or Nodes):** These are the individual data points or entities. Think of them as the "things" in your dataset. Examples include users in a social network, airports on a map, or bank accounts in a financial system.
- **Edges (or Links):** These are the lines that connect the vertices, representing a specific relationship or interaction between them. Examples include a "friendship" edge between two users, a "flight" edge between two airports, or a "transaction" edge between two accounts.

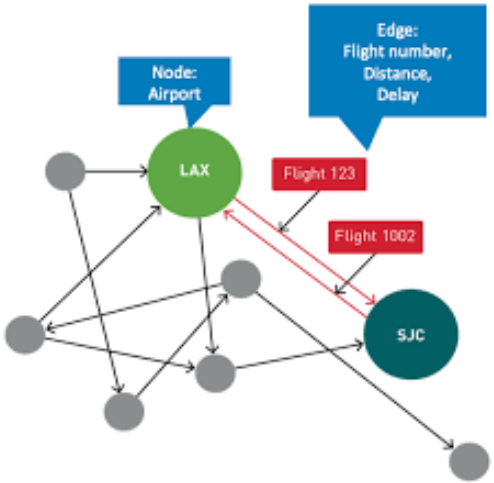


Why We Use Graph Analytics

Beyond Simple Queries: Graph analytics is the practice of using graph-specific algorithms to uncover insights that are not obvious from looking at individual records. It's about understanding the "big picture" structure of the network.

Key Analytical Capabilities: This approach enables us to answer high-value, complex questions:

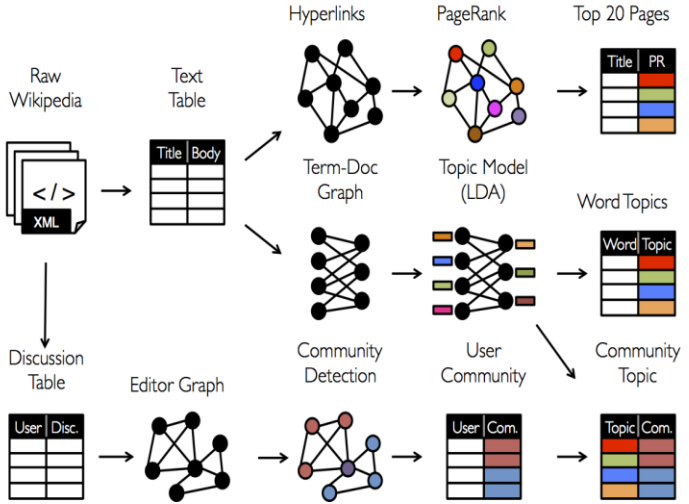
- **Paths and Flows:** Calculating the shortest or most optimal route in logistics, or identifying bottlenecks in a network's flow.
- **Influence and Centrality:** Finding key influencers in a social network, critical infrastructure hubs, or super-spreaders of information.
- **Community Structure:** Automatically detecting distinct clusters or communities within a larger network, which is invaluable for customer segmentation or audience targeting.
- **Anomalies and Fraud:** Identifying unusual patterns, such as complex rings of fraudulent accounts, that deviate from normal behavior.



Example of Graph Analytics with Airplane and Flights

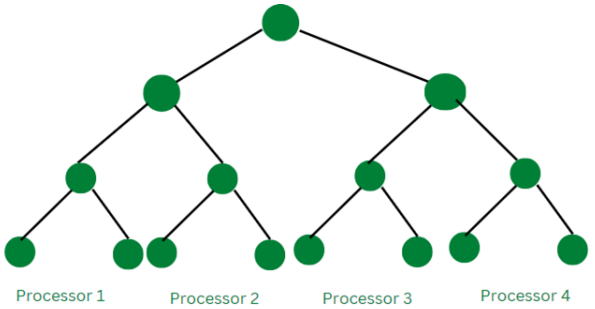
Introduction to Apache Spark GraphX

- **A Library, Not a Standalone System:** GraphX is a core component of the Apache Spark ecosystem. It's an API that extends Spark's powerful data processing capabilities to include graph computation.
- **The Power of Unification:** The primary advantage of GraphX is its **unified approach**. In the past, data engineers needed separate systems for data processing (ETL) and graph analysis. This created complex pipelines with costly data movement.
- **Seamless Integration:** With GraphX, you can perform ETL, SQL queries, machine learning, and advanced graph analysis all within a single Spark application and a single data pipeline. This dramatically simplifies development and improves performance by keeping everything in one ecosystem.



The Graph-Parallel Computation Model

- **Scaling Beyond a Single Machine:** GraphX is built for **graph-parallel computation**. Real-world graphs can be enormous, with billions of vertices and edges, far too large to fit into the memory of a single computer.
- **Partitioning and Distribution:** The graph-parallel model works by **partitioning** (splitting) the graph's vertices and edges across a cluster of multiple machines. The analysis is then performed in parallel on each partition.
- **Minimizing Communication:** An edge that connects two vertices living on different machines is called an "edge cut." Sending data across the network for these cuts is a major performance bottleneck. A key goal of GraphX's model is to partition the graph intelligently to minimize these cuts and keep communication local.



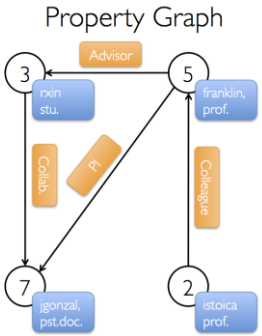
The Property Graph Model

Richer Data Representation: GraphX uses a **Property Graph**, which is a highly flexible model. It allows you to attach arbitrary key-value data, or "properties," to both the vertices and the edges.

Descriptive Vertices and Edges: This allows for a much richer representation of your data:

- **Vertex Property:** A vertex representing a user could store a full profile: `(name: "Alice", age: 30, city: "Pune", interests: ["AI", "Data Science"])`.
- **Edge Property:** An edge representing a rating could store detailed context: `(stars: 5, date: "2025-10-20", comment: "Excellent!")`.

The Importance of Direction: All edges in GraphX are **directed**, meaning they have a source vertex and a destination vertex. This is crucial for modeling asymmetric relationships, like a user "following" another user or a payment being sent from one account to another.



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

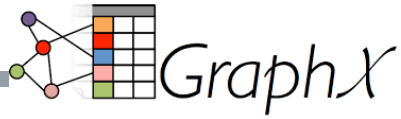
The Foundation: How GraphX Uses RDDs

Built on Spark Core: Under the hood, a GraphX graph is elegantly composed of two of Spark's fundamental data structures: **Resilient Distributed Datasets (RDDs)**.
The Two Core RDDs:

- **A Vertex RDD:** This is a distributed collection of all the graph's vertices, stored as tuples of `(VertexID, VertexProperty)`.
- **An Edge RDD:** This is a distributed collection of all the graph's edges, stored as `Edge` objects which contain the source ID, destination ID, and the edge's property.

Inherited Power: By being built on RDDs, GraphX automatically inherits Spark's most important features:

scalability across a cluster and **fault tolerance**. The "Resilient" aspect means that if a machine fails during a long computation, Spark can automatically recover the lost data partitions.



The Triplet View: A Key Optimization

The Problem with Joins in Distributed Systems: To analyze a relationship, you almost always need information about an edge *and* the two vertices it connects. In a relational database, this would require a `JOIN`. In a distributed system like Spark, joins are extremely expensive because they require shuffling massive amounts of data across the network.

GraphX's Efficient Solution: GraphX brilliantly avoids this costly operation by providing the **triplet view**.

What is a Triplet?: A triplet is a logical data structure that materializes an edge along with the properties of its source and destination vertices. It effectively pre-joins the necessary information, making it locally available for computation on each partition without any network shuffling.

Graph Operators: An Overview

A Rich Toolkit for Manipulation: GraphX provides a comprehensive "toolkit" of operators that allow you to explore, clean, and transform your graph data. These operators are the fundamental building blocks for any graph analysis task.

Three Main Categories: The operators can be grouped into three distinct categories based on their function:

- **Transformation Operators:** These modify the data *on* the vertices and edges without changing the graph's structure.
- **Structural Operators:** These change the *shape* of the graph itself by adding or removing vertices and edges.
- **Join Operators:** These enrich the graph by incorporating information from external data sources.

Transformation Operators

Purpose: To modify the properties of vertices or edges while preserving the graph's underlying structure. The number of vertices and edges remains constant.

Key Operators:

- `mapVertices`: Applies a user-defined function to every vertex's property. This is commonly used to initialize vertex states before an algorithm (e.g., setting an initial distance to infinity for a shortest path calculation).
- `mapEdges`: Applies a user-defined function to every edge's property. This is useful for tasks like re-calculating edge weights or normalizing relationship strengths.

Structural Operators

Purpose: To create a new graph that is a subset of the original by filtering out vertices and edges based on specific criteria.

Key Operators:

- `subgraph`: This is the primary filtering operator. It takes a predicate (a true/false function) and returns a new graph containing only the vertices and edges that satisfy the predicate. It's often used for data cleaning or focusing on a specific part of the network.
- `mask`: This operator creates a new graph that represents the intersection of two graphs. It returns a graph containing only the vertices and edges that are present in *both* the original graph and the one provided as an argument.

Join Operators

Purpose: To enrich the existing graph with new data from an external RDD, which could be a lookup table or a stream of new information.

Key Operators:

- `joinVertices`: This behaves like a SQL `INNER JOIN`. It takes an RDD of `(VertexID, Value)` and updates the properties of vertices in the graph that have a matching ID. Vertices without a match are left unchanged.
- `outerJoinVertices`: This behaves like a SQL `LEFT OUTER JOIN`. It also updates matching vertices but provides a mechanism to handle vertices in the graph that do *not* have a match in the input RDD, for instance, by setting a default value.

Iterative Algorithms & The Pregel API

- **The Nature of Graph Algorithms:** Many of the most powerful graph algorithms (like PageRank or Shortest Path) are **iterative**. This means the final answer cannot be calculated in a single pass; instead, information must be propagated step-by-step across the graph until a stable state is reached.
- **A Model for Iteration:** To manage this complexity, GraphX provides an implementation of the **Pregel API**. This is a well-established, "think like a vertex" model for graph computation.
- **Bulk-Synchronous Parallel (BSP):** Pregel operates in a series of synchronized rounds called **"supersteps."** In this model, all vertices perform their computation for a given step simultaneously, and the system waits for all of them to finish before proceeding to the next step.

How a Pregel "Superstep" Works

The Core Loop: In each superstep, every vertex in the graph performs the same three actions in parallel. This loop continues until the graph state stabilizes.

The Three Actions of a Vertex:

1. **Receives Messages:** The vertex collects all incoming messages that were sent to it by its neighbors in the *previous* superstep.
2. **Computes New State:** It then uses the information from these messages to update its own property or value.
3. **Sends New Messages:** Based on its new state, it sends new messages out to its neighbors, which will be processed in the *next* superstep.

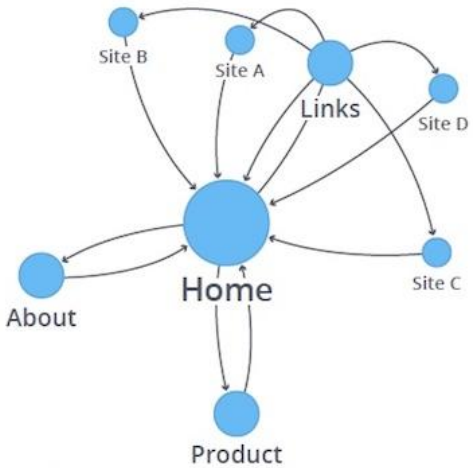
Convergence: The algorithm is said to have **converged** (and the process stops) when a superstep completes and no vertex sent any new messages.

The Built-in Algorithm Library

- **Avoiding Reinventing the Wheel:** Implementing distributed graph algorithms from scratch is notoriously complex and error-prone.
- **A Library of Experts:** GraphX includes a library of common, battle-tested, and highly optimized implementations of standard graph algorithms. This is a massive productivity boost for developers.
- **Focus on the Results:** This library allows data scientists and engineers to focus on interpreting the *results* of the analysis rather than the low-level implementation details. It makes advanced graph analytics accessible to a much broader audience of Spark users. Key examples include PageRank, Connected Components, and Triangle Counting.

Algorithm in Focus: PageRank

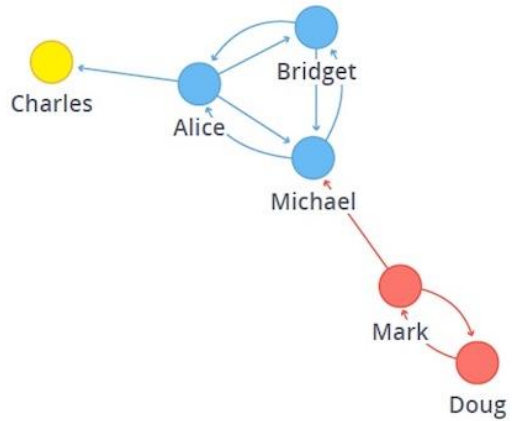
- **Measuring Influence:** PageRank is an iterative algorithm that assigns a numerical score to each vertex, representing its relative importance or influence within the network.
- **The Core Intuition:** The algorithm is based on a simple but powerful idea: a vertex is considered important if it is linked to by *other* important vertices. It's a system of endorsements, where an endorsement from a more influential source carries more weight.
- **Beyond Web Pages:** While it was made famous by Google for ranking web search results, its applications are vast. It is now widely used to find key influencers in social media, identify critical nodes in infrastructure networks, and even rank scientific papers based on citations.



Visualization of PageRank

Algorithm in Focus: Connected Components

- **Finding Disjoint Clusters:** The Connected Components algorithm is used to identify distinct, separate clusters within a graph where everything is interconnected.
- **What is a Component?:** A "component" is a subgraph where every vertex is reachable from every other vertex in that same subgraph (directly or indirectly), but there are no paths to any vertex outside of it.
- **Practical Use Cases:** This is extremely useful for network segmentation. For an e-commerce platform, it could reveal one large component of users who buy electronics and a completely separate component of users who only buy gardening supplies, enabling highly targeted and effective marketing campaigns.



Visualization of Strongly Connected Components

Performance Tuning: Partitioning & Caching

- **Partitioning is Critical:** How the graph is split across the cluster is the most critical factor for performance. A poor partitioning strategy leads to a high number of "edge cuts" and excessive data "shuffling" across the network, which is the biggest bottleneck in most distributed jobs. GraphX provides strategies like `EdgePartition2D` to intelligently co-locate related vertices.
- **The "Cache" Command (`.persist()`):** Iterative algorithms access the same graph data over and over again in each superstep. By default, Spark might recompute the graph from its source on each iteration. Caching tells Spark to explicitly keep the graph loaded in the cluster's memory, avoiding this massive re-computation cost and dramatically speeding up the workflow.

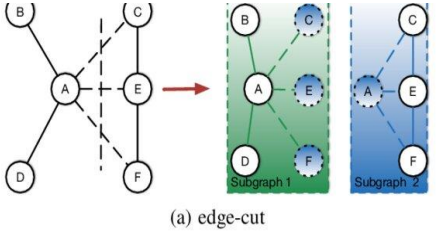
Advanced Topic: Graph Partitioning Heuristics

The Challenge of Distribution: How a graph is partitioned across the cluster is often the single most important factor for performance. The goal is to minimize the **edge cut** (the number of edges that span across different machines), as each cut represents a network communication hop.

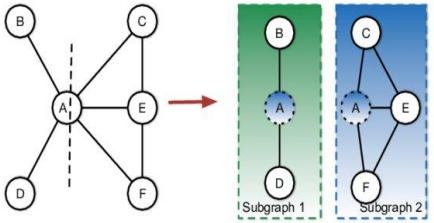
Vertex-Cut vs. Edge-Cut: GraphX employs a **vertex-cut** approach. Instead of cutting edges, it allows vertices to be replicated across multiple partitions, while ensuring every edge resides on exactly one partition. This is highly effective for power-law graphs (common in real-world scenarios) where a few vertices have a massive number of edges.

Partitioning Strategies:

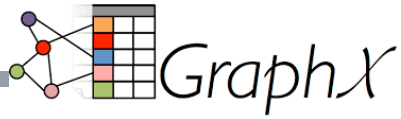
- `RandomVertexCut`: The default strategy. It partitions edges randomly, which can lead to a high edge cut and poor vertex locality. It's simple but often suboptimal.
- `EdgePartition1D`: Partitions edges based on a hash of the source or destination vertex ID. It attempts to group edges by source or destination, which can improve locality over random partitioning.
- **`EdgePartition2D`**: The most sophisticated strategy. It uses a 2D partitioning algorithm that tries to minimize both the number of vertex replicas and the communication cost. It often provides the best performance but has a higher initial partitioning overhead.



(a) edge-cut



(b) vertex-cut



The Evolution: Introduction to GraphFrames DataFrame API

Beyond RDDs: While GraphX is powerful, its RDD-based API can be verbose and lacks the automatic optimization of Spark's newer DataFrame APIs. **GraphFrames** is a library that provides graph analytics capabilities on top of Spark DataFrames.

Key Advantages of GraphFrames:

- **Unified API:** Leverages the familiar DataFrame API, making it easy for those comfortable with Spark SQL to perform graph operations.
- **Performance Optimization:** Queries are automatically optimized by Spark's **Catalyst Optimizer**, which can often generate more efficient execution plans than manually coded RDD operations.
- **Multi-Language Support:** Full support for Scala, Java, Python, and R, whereas GraphX's primary API is Scala-based.

Core Representation: A GraphFrame is simply represented by two DataFrames: one for vertices (`v`) and one for edges (`e`). The vertices DataFrame must have an `id` column, and the edges DataFrame must have `src` and `dst` columns.

Advanced Technique: Declarative Motif Finding

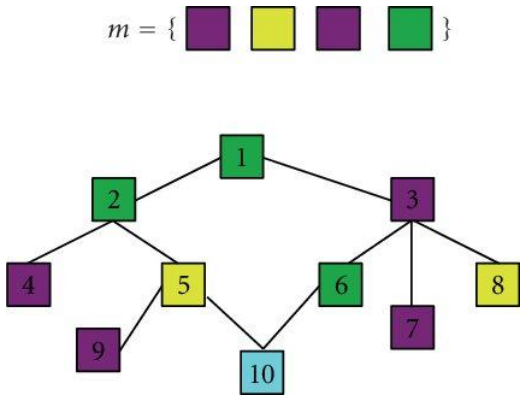
What is a Motif?: A motif is a structural pattern or subgraph that you want to find within a larger graph. For example, "find all triangles of users" or "find all users who sent a message to another user who then replied."

Imperative vs. Declarative: In GraphX, finding complex motifs often requires writing a complex, imperative algorithm using Pregel or multiple joins. GraphFrames introduces a powerful, **declarative API**.

The Motif Finding DSL: You specify the structural pattern you're looking for using a simple string-based Domain-Specific Language (DSL).

- Example: Find users (a) who are friends with (e1) a user (b), who in turn is friends with (e2) a third user (c).
  Scala Code
  ```
  val motif = "(a)-[e1]->(b); (b)-[e2]->(c)"
  val results = g.find(motif)
  ```

- The Catalyst Optimizer translates this declarative query into an efficient physical execution plan, abstracting away the complex join and filter logic.



Example of a graph and a motif. The motif m occurs three times in the graph, at positions {2, 4, 5, 9}, {1, 3, 7, 8}, and {3, 6, 7, 8}.

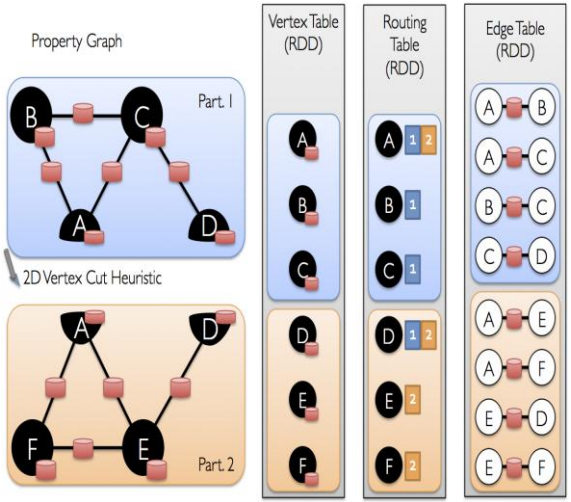
GraphX Internals: The aggregateMessages API

The Engine Behind Pregel: The Pregel API is a high-level abstraction. The core engine that powers it and most of GraphX's algorithms is the `aggregateMessages` function. It is a more flexible and fundamental operation.

Two-Phase Process: `aggregateMessages` works in two phases to optimize communication:

1. **Map Phase (`sendMsg`):** A user-defined function is executed on each edge triplet in parallel on its local partition. This function can choose to send a message (of type `A`) to either the source or destination vertex of the edge.
2. **Reduce Phase (`mergeMsg`):** For each vertex that received messages, a user-defined commutative and associative function is used to combine all incoming messages into a single message.

Why It's Efficient: This design significantly reduces network traffic. Instead of sending thousands of individual messages for a vertex across the network, all messages are first aggregated locally on each partition, and then only one final, combined message per vertex is sent to its destination partition.



Advanced Topic: Serialization and Memory Tuning

Serialization Overhead: In any distributed system, data must be serialized (converted to bytes) to be sent over the network or stored. The default Java serialization is slow and verbose.

- **Kryo Serializer:** For performance-critical applications like graph analytics, switching to the **Kryo serializer** (`spark.serializer = org.apache.spark.serializer.KryoSerializer`) is essential. It is significantly faster and more compact.
- **Class Registration:** For maximum performance, you must explicitly register the custom classes you will be sending in your vertex and edge properties. If Kryo encounters an unregistered class, it falls back to a slower method.

Garbage Collection (GC) Tuning: Iterative graph algorithms create billions of short-lived objects (messages, vertex states). This puts immense pressure on the JVM's Garbage Collector.

- Poor GC performance can cause long "stop-the-world" pauses, killing performance.
- Tuning the JVM's GC settings (e.g., using the G1GC garbage collector with appropriate heap settings) is an advanced but often necessary step for optimizing large-scale GraphX jobs.

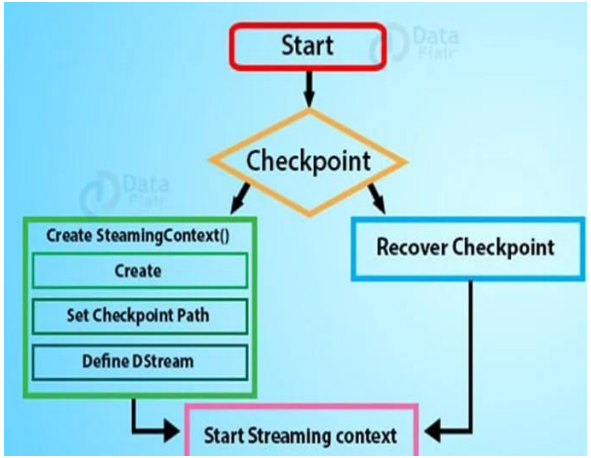
Advanced Topic: Graph Checkpointing

The Problem: Iterative algorithms (like PageRank or custom Pregel jobs) can run for hours. If an executor fails or the job is interrupted, all computational progress is lost, and the job must restart from scratch.

The Solution: Checkpointing: Checkpointing is a fault-tolerance mechanism. You can periodically save the current state of the graph's RDDs to a reliable distributed file system (like HDFS or S3).

How it Works:

- You must set a checkpoint directory (`sc.setCheckpointDir(...)`).
- Inside your iterative loop, you can call `graph.checkpoint()` every N iterations (e.g., every 10 supersteps).
- This severs the RDD's lineage, which can also help prevent stack-overflow errors in very long iterative jobs, but it comes with a significant I/O performance cost.



GraphX vs. GraphFrames: Deep Dive

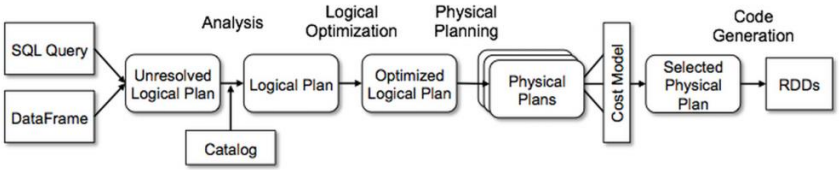
GraphX (RDDs):

- **Pros:** Low-level control, high performance for complex iterative algorithms, strongly typed Scala API.
- **Cons:** Verbose, RDDs lack the automatic optimization of DataFrames, primarily a Scala/Java API (Python support is limited).

GraphFrames (DataFrames):

- **Pros:** Leverages the **Catalyst Optimizer** for automatic query optimization. Full API parity in Python, Scala, and SQL. Uses a simple, declarative API for motif finding.
- **Cons:** Can have higher overhead for purely iterative algorithms compared to the highly optimized GraphX operations.

Key Takeaway: Use **GraphX** for performance-critical, complex iterative algorithms in a Scala environment. Use **GraphFrames** for its powerful declarative API (like motif finding), ease of use with Python/SQL, and integration with the DataFrame ecosystem.



Catalyst Optimizer

Pregel API Internals: Active Set Optimization

The Naive Approach: A simple implementation of Pregel would require every vertex to be "awake" and run its computation in every single superstep, even if it has no new information. This is extremely wasteful.

The "Active Set" Optimization: The GraphX Pregel implementation maintains an "active set" of vertices. A vertex is only in the active set for a superstep if it received a message in the *previous* superstep.

How it Works:

1. **Superstep N:** Only vertices in the active set execute the `vprog` (vertex program).
2. These active vertices send messages.
3. **Superstep N+1:** The "active set" is now comprised *only* of the vertices that received a message in Superstep N.

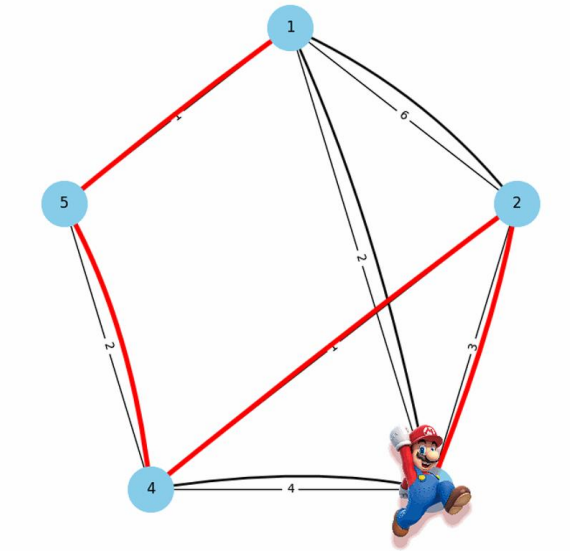
Benefit: As an algorithm converges (like SSSP), fewer and fewer vertices find shorter paths, so they stop sending messages. The active set shrinks, dramatically reducing computation in later supersteps.

Custom Algorithm: Single Source Shortest Path (SSSP)

The Problem: Find the shortest path (lowest cumulative edge weight) from a single "source" vertex to all other vertices in the graph.

Pregel Implementation: This is a classic Pregel use case.

- **Initialization:** The graph is initialized with all vertex distances set to `Double.PositiveInfinity`, except for the source vertex, which is set to `0.0`.
- **Superstep Logic (`vprog`):** The vertex program receives incoming path-length messages. If an incoming message (e.g., `5.0`) is *shorter* than the vertex's current distance (e.g., `10.0`), the vertex updates its distance to `5.0` and sends new messages to all its neighbors with its new distance plus the edge weight (e.g., `5.0 + edge_weight`).
- **Convergence:** The algorithm converges when no vertex can find a shorter path, and thus no new messages are sent.



Super Mario using Dijkstra's algorithm to find shortest path

Graph-Aware RDD Operations: mapReduceTriplets

A Powerful Generalization: `aggregateMessages` (which powers Pregel) is a highly optimized and specialized function. The more general-purpose "workhorse" operator is `mapReduceTriplets`.

How it Works: This operator is a direct combination of MapReduce and the triplet view:

1. **Map Phase:** It runs a user-defined `map` function on every single triplet (`EdgeTriplet`) in the graph. This function can extract information and produce a set of key-value pairs.
2. **Reduce Phase:** It then aggregates all the generated pairs by key using a user-defined `reduce` function.

Use Case: This is extremely powerful for performing vertex-centric computations in a single pass. A classic example is computing the `out-degree` of every vertex:

- **Map:** For each triplet, send a message (count of 1) to the `srcId` vertex.
- **Reduce:** Sum all the 1s for each `srcId`