**Kineograph**

* distributed system that takes a stream of incoming data to construct a continuously changing graph
* supports graph-mining algorithms to extract timely insights from the fast-changing graph structure
* Kineograph creates a series of consistent snapshots using a novel and efficient epoch commit protocol
* includes an **incremental** graph-computation engine
* First, new information (e.g., tweets) is continuously generated and is far more time-sensitive than mostly-static web pages
* Second, while each piece of information may be small and contains limited textual content, rich connections between entities such as users, topics, and tweets can be powerful in revealing important social phenomena
* Challenges:
  + First, Kineograph must handle continuous updates, and its computation must produce timely result
  + Second, Kineograph must maintain a graph structure that captures the relationships among various entities
  + Third, Kineograph must support graph-mining algorithms that extract insights from the graph structure

=> Kineograph addresses those challenges by designing a distributed in-memory graph storage system, along with a graph engine that supports incremental iterative propagation-based graph mining

* produces consistent snapshots periodically, so that existing graph-mining algorithms can be applied on a static snapshot
* graph mining algos und graph updates sind voneinander entkoppelt, sodass es keine Probleme gibt
* each ingest node reports the graph update progress in a global progress table maintained by a central service
* stores the graph-structure metadata separately from the application data associated with the graph (um eben Graph updates und Graph Berechnungen zu trennen)
* The storage layer of graph nodes implements a distributed key/value store:
* A graph is split into a fixed number (say 512) of logical partitions, which are further assigned to physical machines. Currently, graph partitioning is based on the hashing of vertex ids
* Each logical partition consists of a set of vertices, each with a set of directed weighted edges stored in a sorted list
* Each vertex also has a set of named vertex-fields that store the associated data for each configured graph mining algorithm. The type of values stored in vertex-fields is arbitrary, as long as it can be serialized
* Snapshot und epoch meachanism:
  + The snapshot mechanism is implemented through cooperation among ingest nodes, graph nodes, and a global progress table
  + Ingest nodes sind auch dafür zuständig neu ankommende Daten zu verarbeiten, diese Daten können ein Set von mehreren Update Operationen sein (z.B. v2 erstellen, Kante bei v1 und bei v2 einfügen)
  + Ingest node kreiert eine Transaktionsfolge mit kontinuierlichen Seq Nummern ( Nummern entscheiden auch zu welchem Snapshot eine Transaktion gehört)
  + Ingest nodes senden updates an graph nodes (zusammen mit der seq nummer einer transaktion, zu welcher das update gehört)
  + Eine progress table speichert den jeweiligen Fortschritt eines ingest node (anhand einer seq nummer für jeden ingest node)
  + Ingest node updatet seine seq nummer, sobald er mitbekommen hat, dass alle relevanten graph nodes die nötigen Updates bis zur jetzigen seq nummer ausgeführt haben. Die seq nummer s-i von Knoten i erhält der ingest node von der progress table
  + Der sog. Snapshooter nimmt den Vektor von seq nummern der ingest nodes bspw. Alle 10 Sekunden und legt so das Ende einer aktuellen „epoch“ fest.
  + Ein Update von ingest node i an einem Knoten wurde durchgeführt wenn die seq nummer a <= s-i ist, also in der aktuellen „epoch“ enthalten war.
  + The process of creating a snapshot does not stop incoming updates. Ingest nodes continuously send new graph updates into the system with higher sequence numbers
* ACID gegeben
* The computation layer of graph nodes in Kineograph is responsible for executing incremental graph-mining. Computation results are updated based on recent changes in the graph, reflected in new snapshots
* Knoten-basiertes Berechungsmodell, d.h. alle Daten in Feldern der Knoten gespeichert – Berechungen werden vollzogen, indem die Knoten iteriert werden
* User-defined rules, um zu prüfen, ob der Knoten Status significant verändert wurde im Vgl. zum vorherigen Snapshot.
  + Keine signifiakante Änderung: wert des Knoten wird erneuert
  + Signifikante Änderung:
    - update wird einem Set von anderen Knoten mitgeteilt (i.d.R. Nachbarn)
    - Optionale aggregation Phase, bei der Knoten bei einer Berechnung globaler Werte beteiligt ist
* Entweder beim Koten selbst ändert sich etwas oder eine Änderung ist nötig, da ein anderer Knoten verändert wurde. Beide Szenarien sind mögliche Trigger für ein Status Update. Eine Berechnung ist dann abgeschlossen, wenn bei allen Knoten kein Status Update mehr stattfndet.
* Um die Kommunikation zw. Den Knoten zu fördern, bietet Kineograph *push* und  *pull* Operationen, bei denen Knoten gezielt Infos an anderen Knoten senden bzw. diese von anderen erfragen können
* **Aggregators: wie bei Pregel (siehe Pregel Paper folgende Infos): jeder Knoten kann einen bestimmten Wert an einer Aggregator senden, das System sammelt diese Werte durch eine Reduction Funktion und gibt diesen globalen Wert dann im nächsten Snapshot/Superstep/epoch wieder an alle Knoten weiter. Bei default wird der aggregate value (min, max, sum denkbar) durch den Input von Werten zugehörig zum einem bestimmten Snapshot. Ein sog. *sticky* aggregator ist allerdings auch möglich, der input values aus mehreren Snapshots/Epochs nutzt**.
* **Außerdem wie bei GraphLab Sync Mechanism** (GraphLab Paper): ..
* Auch graph updates sind möglich, z.B. erhöhe die Kosten einer Kante um 10%
* Kineograph ist so gedacht, dass eine Applikation on top z.B. die user defined functions bereitstellt oder die IDs zu Partitions mapped
* Die application stellt dann auch die Liste der Felder eines Knotens bereit sowie mögliche graph-mining Algorithmen
* TunkRank: influence of certain users. Influence von X ist Summe aus (1+p \* Influence(Y)/(Anzahl Follower von Y) über alle Follower von X. p ist eine konstante Retweet Wahrscheinlichkeit.

Chronos

* Chronos is a storage and execution engine designed and optimized specifically for running in-memory iterative graph computation on temporal graph
* Understanding the evolution of graphs over time often involves running those graph mining algorithms on a series of snapshots, which we refer to as temporal graph mining
* Chronos für multi-core machines und verteilte systeme designed
* Rather than taking the straightforward approach of applying an existing graph engine on each snapshot of a temporal graph, Chronos proposes *locality-aware batch scheduling* (LABS) with the following two key observations
  + Zwei mögliche Formen der Lokalität: einmal zeitlicher Natur (zwei aufeinanderfolgende Zeitpunkte eines Knoten-Status werden aneinandergereiht) und zum anderen struktureller Natur (Status zweier Nachbarknoten zum selben Zeitpunkt (z.B. innerhalb eines Snapshots) werden als nebeneinander gelegt.
  + The design of Chronos favors time-locality when it lays out multiple snapshots of a graph in memory
  + Zweites Merkmal von LABS: Chronos makes a decision to batch operations associated with each vertex (or each edge) across multiple snapshots
* Chronos also examines the interplay with incremental graph computation that might help speed up iterative graph mining on multiple snapshots
* Chronos assumes that the temporal graph data is initially available in persistent storage such as disk or flash
* Chronos kann Berechnung zum Zeitpunkt t durchführen und alle anderen können das nur „jetzt“
* Fokus von Chronos ist zu untersuchen, wie sich ein Graph zeitlich verändert
* Existierende Systeme wie Pregel, GraphLab, PowerGraph: ein Knoten scatter (lokalen Wert) an Nachbarn, dann gather von den Nachbarn und mit den gesammelten Updates wird neuer lokaler Wert des Knotens berechnet -> statisch anhand eines Snapshots, Chronos will scatter-gather über eine Serie von Snapshots ermöglichen
* run iterative computation on multiple snapshots together. That is, he propagation from one vertex to its neighbors on multiple snapshots can be scheduled together
* One obvious option is to assign different snapshots to different cores, while an alternative is to assign different graph partitions (aligned across multiple snapshots) to different core (zweite Variante die bessere Wahl)
* LABS gruppiert die unterschiedlichen Ausprägungen eines Knoten zusammen
* In-Memory Data Structure:
  + Zu Beginn werden die interessanten Snapshots in den Hauptspeicher/Arbeitsspeicher geladen, um wiederholten Zugang zu ermöglichen
  + Aufteilung in Knoten und Kanten Array
  + Time-locality: Daten eines Knotens hintereinandern im Array, dh mehrere Versionen (Snaphots) hintereinander eines Knoten. In structure-locality sind Konten eins, zwei, drei von Snapshot 1, dann eins, zwei, drei von Snapshot 2 usw.
  + Kanten Array, eine Kante zwischen zwei Knoten kommt in diesem Array einmal vor, und über Snapshot bitmap lässt sich identfizieren, ob Kante in einem Snapshot vorhanden war oder nicht. Z.B. bitmap 110: in Snapshot 0 und 1 war Kante vorhanden, in Snapshot 2 zum Bsp. Nicht
  + Zusätzlich im Kanten Array: Kante kann in den unterschiedlichen Snapshots unterschiedliche Daten gehabt haben. D.h. z.B. w-i-j-0 = 50: in Snapshot 0 hatte Kante zwischen Knoten i und j das Gewicht von 50.
  + Wahl fiel auf zeitliche statt strukturelle Dimension, da hier oft lineare Daten gegeben
* Locality Aware Batch Scheduling:
  + Auf einmal werden alle Knoten-Zustände aller Snapshots in die Berechnung mit aufgenommen
* Incremental computation:
  + Am Bsp SSSP: nicht immer incremental computation verringert die Berechungszeit. Fällt bsp eine kante nahe source node weg, dann müssen viele knoten ihre distanzen neu berechnen, was dauert
  + Chronos verbessert incremental computation in two ways:
    - First, to compute N snapshots from S0 to SN−1, Chronos first computes the result for snapshot S0. After having the result for S0, Chronos then computes the rest of N−1 snapshots (S1 to SN−1) in a batch using LABS
    - Snapshots sind bereits bekannt, Chronos kann den Schnittpunkt der N Snapshots berechnen. Dh jeder echte Snapshot fügt hinzu oder entfernt Kanten und Knoten zu diesem Schnittpunkt. Dadurch der Umfang, der incremental berechnet werden kann, wird vergrößert.

GraphTau

* Berechnungen auf Stream von konsistenten und stabilen Graphen Snapshots anhand eines kleinen Sets von Operatoren, die solche Streams manipulieren können
* Built on top of Spark
* Neuer Graph Snapshots jedes Mal, wenn neue Daten eintreffen
* Frameworks generell sollten folgende Fähigkeiten besitzen:
  + Müssen iterative Graph Algorithmen in Echtzeit ausführen können
  + Analytische Aufgaben verlangen häufig eine Kombination von graph-structured data mitunstrukturierten Daten aus Tabellen
  + Analyse über Windows of input data laufen lassen
* Bei Kineograph und Chronos kein Zusammenspiel von structured und unstructured data
* Und die beiden (kineo und chronos) bieten keine window operations auf mehreren Graphen
* Naiad bietet keine window operations auf graph snapshots
* 2 Kernmerkmale von GraphTau:
  + first, it provides a way to create and manipulate consistent graph snapshots in user defined window
  + Second, it presents an incremental computation model that allows graph computations to "shift" from a stale (alten) snapshot to a new snapshot even in between iterations of the underlying algorithm
* The main idea in GraphTau is to treat time-evolving graphs as a series of consistent graph snapshots, and dynamic graph computations as a series of deterministic batch computations on discrete time intervals
* Computation Model:
  + Annahme PageRank läuft, währenddessen ändert sich zugehöriger Graph, dann Algo läuft durch und liefert „veraltetes“ Ergebnis, für das „neue“ Ergebnis muss der Algo nochmals komplett durchlaufen.
  + Dafür GraphTau bietet Pause-Shift-Resume computation model: sobald erster Snapshot vorhanden, Berechung startet, wenn neuer Snapshot kommt, dann pausiert die Berechnung auf aktuellem Graph, shiftet meta-data zum neuen Snapshot und macht dann weiter mit der Berechung auf dem neuen Graph.
  + Für Algorithmen, bei denen sich das PSR-Modell nicht anwenden lässt, führt GraphTau Online Rectification ein (Online Nachbesserung). Anhand des connected component algos: 3 knoten sind in einer cc, dann fällt während dem laufenden algo ein knoten weg, dann geht das Modell zurück an eine stelle, bei der der wegfallende knoten noch keinen einfluss auf die anderen beiden hatte. Es müssen Status gemerkt und gekannt werden.
* Consistency wird geschaffen, indem man z.B. timestamps einführt, so kann bspw. Keine Kante an einen nicht existierenden Knoten angefügt werden
* Zeit wird in nicht überlappende Intervalle eingeteilt.
* Wichtige Punkte bei den Systemen allgemein: Snapshot Generierung, Koordination zwischen Generierung und Berechnungsstufen und Berechnungen zwischen mehreren Snapshots

Tegra

2 Haupttechniken, die auf snapshots angewandt werden

* Jeder mögliche snaphot, der den Nutzer interessieren könnte, wird gespeichert [hoher Speicheraufwand]
* Alle Veränderungen zwischen 2 Snapshots werden als sog. Deltas geloggt (aus den Unterschieden wird dann der benötigte snapshot berechnet) [teure Rechenoperationen und mit der Zeit langsam]

Grundidee von Tegra ist es, das Speichern, die Berechnung und Kommunikation zwischen den Snapshots in einem time-evolving graph zu teilen. Mithilfe von *Timelapse* versteckt Tegra Feinheiten des Zustandsmanagements vom Endnutzer. Timelapse ist als eine Serie von Snapshots zu verstehen, beginnend beim ursprünglichen Graphen. Dadurch können Nutzer unabhängig von einander auf den verschiedenen Versionen arbeiten.

Tegra ist das erste Streaming Graph Verarbeitungssystem, das einheitlich Speicherung, Berechnung und Kommunikation nutzt.

Sharing Storage: radix tree, um ganze Snapshots ohne Duplikate zu speichern.

Sharing Computation: inkrementelle Berechnung, speichert Berechnungen vergangener Iterationen, die für Teile wiederverwendet werden können, die keiner Neuberechnung bedürfen.

Sharing Communication:

Tegra unterstützt 1 Million Kanten Updates pro Sekunde pro Maschine.

NAIAD (in Grafik: dashed = gestrichelt)

* Naiad is a distributed system for executing data parallel, cyclic dataflow programs
* the ability to perform iterative and incremental computations
* the model enriches dataflow computation with timestamps that represent logical points in the computation
* Naiad is our prototype implementation of timely dataflow for data parallel computation in a distributed cluster
* interactive graph mining
* he application performs iterative processing on a real-time data stream
* interactive queries on fresh results possible
* computation model *timely dataflow* with its core features:
  + structured loops allowing feedback in the dataflow
  + stateful dataflow vertices capable of consuming and producing records (Datensätze) without coordination
    - (die zwei ersten dienen iterative and incremental computations with low latency)
  + notifications for vertices once they have received all records for a given round of input or loop iteration (macht es möglich, konsistente Ergebnisse (als Output und für Zwischenstufen bei Berechnungen) zu erzeugen – sowohl beim Streaming als auch bei Iterationen
* Graphen bei Timely Dataflow sind **gerichtet und können Kreise enthalten**
* Stateful vertices asynchronously receive messages and notification of global progress
* Kanten tragen Daten mit logischen timestamps, wodurch es möglich ist den globalen Fortschritt zu messen
* Model supports concurrent execution of different epochs and loop iterations
* Input vertices receive a sequence of messages from external producer and output vertices emit sequence of messages back to the external producer
* Input and output vertices are labeled with integer epoch
* Ingress, feedback and egress nodes in nested context
* Timestamp: epoch e, loop counter c
* One loop counter for each of the k loop contexts that contain the associated edge // loop counters distinguish different iterations
* Timely dataflow vertices send and receive timestamped messages, and may request and receive notification that they have received all messages
* U sendet eine Nachricht an v ( u.SendBy(e, m, t) und korrespondierend v.onRecv(e, m, t)
* onNotify(t) is an indication that all onRecv(e, m, t) invocations have been delivered
  + d.h. timestamp t dann „fertig“
* Naiad is our high-performance distributed implementation of timely dataflow
* Workers können local über eine shared memory kommunizieren oder remote über TCP
* ….
* (F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In Proceedings of the 6th Conference on Innovative Data Systems Research (CIDR) , Jan. 2013) wenn mehr Infos notwendig
* All programs follow one pattern: first define dataflow graph (input stages, computational stages, output stages), then repeatedly supply the input stages with data