Ereignisbasierte Dateninfrastrukturen

1. Einleitung

Ziele: Unterschied zwischen Event-basierten Ansätzen und Nachrichten-basierten Ansätzen erkläre

Architektur-Begriffe voneinander abgrenzen und vergleichen

Batch- und Stream-Processing abgrenzen

Cloud- und Edge-Processing erklären

Event und Message

* Nachrichtengesteuertes System: Empfänger warten auf Nachricht und reagieren darauf, andernfalls bleiben sie untätig (Sensor wartet auf Kommando um Informationen zu liefern)
* Ereignisbasiertes System: Listener werden an Quellen von Ereignissen angehängt, sodass sie aufgerufen werden, wenn Ereignis ausgelöst wird (Service registriert sich einmalig für Ereignisse und erhält bei Aktualisierung Werte); es wird nicht darauf geachtet wer die Nachricht empfängt

Architektur-Begriffe

* Synchrones Interaktionsmodell: wenn Prozedur aufgerufen wird, muss aufrufender Code blockieren und warten bis aufgerufene Code Ausführung beendet und Kontrolle zurückgibt
* Asynchrones Interaktionsmodell: ermöglicht es dem Aufrufer die Verarbeitung unabhängig vom Verarbeitungsstatus der aufgerufenen Prozedur fortzusetzen; erfordert einen Vermittler
* Message-Oriented Middleware: zentraler Hub zum Austausch von Nachrichten; Ausfallsicherheit durch Store-and-Forward-Mechanismus; Skalierbarkeit der Teilsysteme unabhängig voneinander; Nachteile: zusätzliche Komponente in der Architektur, bei Adressierung an nicht vorhandene Empfänger könnten Probleme entstehen (Puffer läuft voll oder Absender wartet ewig auf Antwort), Single-Point-of-Failure wenn System nicht korrekt betrieben wird; häufig Systeme existent die Messaging in Event-Verteilung ermöglichen
* Event-Driven Architecture: kein direkter Empfänger aber zentraler Punkt in der Architektur; Ausfallsicherheit gesteigert, da einzelne Services ausfallen können; Skalierbarkeit der einzelnen Komponenten; Nachteile: wie bei MOM ohne Adressierung an nicht vorhandene Empfänger; häufig Systeme existent die Messaging in Event-Verteilung ermöglichen
* Service-Oriented Architecture mit Enterprise Service Bus: einzelne Dienste übernehmen Aufgabe und arbeiten über zentrale Integrationsschicht in einem Geschäftsprozess; Vor- und Nachteile wie bei EDA; über Bus werden vergleichbare Ziele wie mit MOM/EDA erreicht
* Microservice Architecture: Services die unabhängig arbeiten und jeder für sich einen Teil des Systems realisiert; Skalierbarkeit, Ausfallsicherheit, Resilenz, Isolation, Technologieunabhängigkeit; Nachteile wie bei EDA; auch hier der Einsatz von MOM/EDA als Möglichkeit zum Austauschen von Nachrichten möglich

Batch- vs. Stream-Processing

* Batch-Processing: Daten werden nach Entstehung oder Erhalt von Eingangslogik sortiert und strukturiert; anschließend erfolgt Ablage in Datenbank bis Weiterverarbeitung oder Analyse; nicht echtzeitfähig (während Analyse erhaltene Daten fließen nicht in aktuellen Analyseprozess ein)
* Stream-Processing: Basis sind kontinuierliche Datenströme; verarbeitet Daten in Echtzeit; geeignet für Echtzeit-Analyse von Streaming-Daten
* Natives Streaming verarbeitet jeden Datensatz sofort, ohne auf weitere Datensätze zu warten
* Micro-Batching sammelt Daten über kurze Zeitabstände und führt anschließend Weiterverarbeitung durch

Bezug zu Cloud-Applications

* Cloud-Computing: Modell, dass bei Bedarf, jederzeit und überall bequem über ein Netz auf einen geteilten Pool von konfigurierbaren Rechnerressourcen zuzugreifen, die schnell und mit minimalem Managementaufwand zur Verfügung gestellt werden können
* Edge-Computing: nahe am physischen Standort der Datenquelle; schnellere und zuverlässigere Services; gemeinsamen Pool an Ressourcen über Vielzahl an Standorte verteilt; auch Fog-Computing genannt

2. Verteilen von Ereignissen und Nachrichten

Ziele: Point-to-Point und Publish/Subscribe Pattern erklären sowie mittels Diagramme die Funktionsweise darstellen

Verschiedene Formen der Nachrichtenzustellung unterscheiden und erklären

Message- und Event-Broker differenzieren

APIs und Protokolle wie JMS, MQTT und AMQP unterscheiden und (kurz) erklären

Software-Lösungen wie Mosquitto, ActiveMQ und Kafka unterscheiden, für ein konkretes Szenario gezielt wählen und Client-Anwendungen realisieren

Grundlagen

* Publish/Subscribe: Nachrichten an Verbraucher veröffentlichen, die an Nachricht interessiert sind; Produzent veröffentlicht und Konsumenten abonnieren; Vorteile: geringe Kopplung (Anzahl und Art der Nachrichten muss nicht bekannt sein, leichte Skalierbarkeit ohne dass sich auf Publisher auswirkt), geringe kognitive Belastung für Subscriber (Innenleben des Publishers nicht von Interesse), Separation of Concerns (ermöglicht Trennung von Belangen), verbessere Testbarkeit, verbesserte Sicherheit (eignet sich gut für Least-Privilege-Prinzip)
* Point-to-Point-Messaging: Verteilungsmuster, das in Nachrichtenwarteschlangen mit einer Eins-zu-Eins-Beziehung zwischen Absender und Empfänger verwendet; jede Nachricht nur einmal gesendet und konsumiert; Beispiel ist Verarbeitung von Gehaltsabrechnungen, wo Verarbeitung nur einmal erfolgen darf
* At-most-once delivery: Nachricht wird einmal zugestellt; wenn fehlschlägt, kann sie verloren gehen
* At-least-once delivery: Nachricht wird mindestens einmal zugestellt; Fehlschläge werden damit vermieden
* Exactly-once delivery: Nachricht wird genau einmal verschickt; kann nicht verloren gehen oder dupliziert werden
* Event-Broker: drei Akteure (Ereignisproduzent erzeugt Ereignis, in geordneter Weise an bestimmte Verbraucher (Mediator) oder gleichmäßig an alle interessierten (Broker) verteilt, Verbraucher empfängt Ereignisse und verarbeitet sie
* Message Queue: einzelne Dienste können Queue haben, die Vielzahl von Aufrufen enthält, die sequentiell sortiert und an interessierte Parteien gesendet werden; Verbraucher können Nachricht aus Warteschlange nehmen, um sie weiterzuverarbeiten, zu behalten oder zu eigenen Prozessen hinzuzufügen

Datenformate, APIs und Protokolle

* JMS (Java Messaging Service): standartisierte API für Java; bekannte Implementierungen sind Apache ActiveMQ oder IBM MQ
* AMQP (Advanced Message Queuing Protocol): standardisiert keine APi, sondern ein Netzwerk-Protokoll auf TCP/IP-Ebene; ermöglicht verschlüsselte und kompatible Nachrichtenübertragung; Open-Source-Standard für asynchrones Messaging
* MQTT: Kommunikationsprotokoll mit Funktionen, die speziell auf IoT-Lösungen ausgerichtet sind; verwendet TCP-Verbindungen für Zuverlässigkeit, Fragmentierung und Ordnung; Ziel ist es Daten-Overhead eines jeden MQTT-Pakets zu minimieren; Benachrichtigungen, wenn Client unerwartet die Verbindung trennt; Publish-Subscribe-Routing (ermöglicht Hinzufügung weiterer Konsumenten und Produzenten)
* Vorteile von MQTT: einfacher zu konfigurieren; geringerer Byte-Verbrauch; Will- und Retained-Messages; Publish-Subscribe-Paradigma
* Vorteile von AMQP: mehr Möglichkeiten für Message Queuing; Konfiguration von Routing-Topologien ist fehleranfällig

JMS with Apache ActiveMQ Artemis

* Open-Source-Projekt für “multi-protocol, embeddable, high performance, clustered, asynchronous Messaging-Systems”
* Fungiert als MOM; in Java geschrieben

MQTT with Mosquito

* Will-Messages: Nachrichten die beim Abbruch der Verbindung hinterlassen werden sollen
* Retained-Messages: Nachrichten dieser Art bleiben als letzte bekannte Nachricht erhalten; über Flags realisiert

Apache Kafka

* Kann als Message- bzw. Event-Broker verwendet werden und erlaubt perfomantes Verteilen von Nachrichte/Ereignissen in skalierbarer Umgebung
* In Java geschrieben; APIs unterscheiden sich in Producer API, Consumer API, Streams API, Admin API und Connect API
* Producer/Consumer: Erzeuger und Verbraucher von Nachrichten, Records: Ereignisse und Nachrichten in Kafka, Topics: zusammengefasste Topics, Partitionen: Unterteilung von Topic
* Zookeeper: zentraler Dienst zur Verwaltung von Konfigurationsinformationen, zur Namensgebung, zur verteilten Synchronisierung und zur Bereitstellung von Grouping Services; diese sind häufig notwendig in verteilten Anwednungen
* Records: Kafka organisiert Daten in Records; Record transportiert Werte als Value und hat Key und Timestamp
* Topics: fassen Records zusammen; Producer senden Records an das Ende eines Topics, es entsteht eine Art Log; Kafka erstellt Topics wenn Nachrichten in Topics platziert werden die nicht existieren; können bei Bedarf nachträglich konfiguriert werden
* Partitionen: Topics sind in Partitionen unterteilt; neuer Record wird an Partition angehängt; erlauben das Anhängen neuer Records; gehört zu den effizientesten Operationen auf einem Massenspeicher; solche Operationen sind sehr zuverlässig und einfach zu implementieren
* Keys: jeder Record kann einen Key haben; Kafka verarbeitet Records in der richtigen Reihenfolge, wenn Sie einen Key haben; Beispiel für Key: ID für einen Auftrag in einem Topic, damit Zustandsänderung in richtiger Reihenfolge verarbeitet wird; Wahl des Schlüssels relevantes Designkriterium, je nach Producer-Konfiguration und Eigenschaften des Schlüssels entscheidet sich Verteilung über Partitionen
* Commit: wenn Consumer Record verarbeitet haben, committen sie einen neuen Offset; so weiß Kafka immer, welche Records welcher Consumer bearbeitet hat und welche bearbeitet werden müssen; Consumer Records können committen, bevor sie tatsächlich bearbeitet sind; Consumer kann Batch von Records committen, wodurch bessere Performance möglich wird weil weniger Commits notwendig sind; Kafka unterstützt Exactly once
* Consumer aufbauen: Sammlung von Konfigurationen; Client-ID für den die Offsets verwaltet werden; Host und Port für die Server; Group-ID; Deserializer zum Lesen von Keys und Values; Erzeugung des Consumers; Consumer an verschiedene Topics registriert; je Verarbeitungsschritt nach Nachrichten gefragt; Nachrichten verarbeitet, ausgegeben und Commit realisiert um Offset zu verändern
* Producer aufbauen: Konfigurationen; Anzahl an ACK die Leader benötigt; bevor Nachricht-Anfrage als vollständig betrachtet; Serializer für Keys und Values; Producer erzeugt; Nachrichten in Records verpackt unter Angabe eines Topics, Key und eines Values; Record wird versendet; bei synchroner Verarbeitung kann mittels get gewartet werden; bei aynchroner wird Callback-Handler hinterlegt
* Polling: Consumer pollen Daten; Polling schützt davor, unter zu viel Last zu geraten; Consumer können selbst entscheiden wann sie Records verarbeiten; Spring Kafka pollt im Hintergrund neue Records
* Replikation: Partitionen speichern Daten und können über Server verteilt werden; jeder Server bearbeitet eigene Partitionen und erlaubt Lastverteilung; Partitionen können repliziert werden und Daten liegen auf mehreren Servern für Steigerung der Ausfallsicherheit; Replikation so umgesetzt dass ein Leader schreibt und restliche Replicas als Follower schreiben; Producer schreibt direkt an Leader; wenn Schreiboperation nicht erfolgreich war, kann Producer über API festlegen, dass Übertragung erneut versucht wird; Record kann trotz Fehler erfolgreich übertragen werden, Consumer muss dann mit Duplikaten umgehen können
* Consumer Groups: Consumer sind in Consumer Groups organisiert; jede Partition hat genau einen Consumer, ein Consumer kann für mehrere Partitionen zuständig sein; wenn Consumer Nachricht mit einem Key bekommt, erhält er auch alle Nachrichten mit demsleben Key, weil alle zur selben Partition gehören; maximale Anzahl der Consumer ist gleich Anzahl der Partitionen; idealerweise mehr Partitionen als Consumer, um bei Skalierung weitere Consumer hinzuzufügen
* Persistenz: Kafka eine Mischung aus Messaging-System und Datenspeicher; Records können von Consumern gelesen und von Producern geschrieben werden; werden dauerhaft gespeichert; Consumer speichern lediglich ihren Offset, Consumer kann alle Record verarbeiten, die jemals von Producer geschrieben worden sind, um so eigenen Zustand auf aktuellen Stand zu bringen
* Log Compaction: Kafka muss mit der Zeit immer mehr Daten speichern; einige Records werden jedoch irgendwann irrelevant; Log Compaction kann dazu genutzt werden, dass alle Records mit demselben Key bis auf den letzten entfernt werden; deswegen Wahl des Keys sehr wichtig
* Absicherung/Zugangskontrolle: Absicherung über Verschlüsselung mit SSL/TLS; Authentifizierung über SSL, SASL, HTTP Basic Authentication
* Scaling: Grundlage sind Partitionen und Replikationen; Topics in verschiedene Partitionen aufgeteilt; Partitionen repliziert im Cluster und Consumer auf Partitionen zugewiesen

3. Streaming APIs und Reactive Programming

Ziele: Reactive Programming beschreiben und von klassischem imperativem Vorgehen abgrenzen

ReactiveManifesto und die vier Kernaussagen kurz erklären

Konzepte des Reactive Programming erklären

Marble Diagrams verwenden, um Verarbeitungsvorgänge zu visualisieren

Typische Operatoren des Reactive Programmings anwenden, um Problemstellungen zur Datenstromverarbeitung funktional zu lösen

Basics

* Imperative Programmierung: z.B. Java; Folge von Anweisungen die in der Reihenfolge ausgeführt werden in der sie geschrieben sind
* Funktionale Programmierung: Bausteine sind keine Objekte, sondern Funktionen und Prozeduren; Merkmale: Funktionen höherer Ordnung nehmen andere Funktionen als Argumente; unveränderliche Daten (statt bestehenden Werten wird oft mit Kopien der ursprünglichen Werte gearbeitet); Concurrency (Gleichzeitigkeit wird unterstützt und ist sicherer zu implementieren); referentielle Transparenz (Berechnungen können jederzeit durchgeführt werden und liefern immer das gleiche Ergebnis); Lazy Evaluation (Werte können nur bei Bedarf berechnet werden)
* Reactive Programming: befasst sich mit Ereignisströmen; wenn Ereignis eintritt wird darauf reagiert, indem etwas getan wird
* Reactive Manifesto: definiert Kernprinzipien des Reactive Programming; Reaktionsfähig (System reagiert zeitnah; Probleme werden schnell erkannt und effektiv behandelt); Widerstandsfähig (System auch im Falle eines Ausfalls reaktionsfähig; wird durch Replikation, Eingrenzung, Isolierung und Delegation erreicht; Ausfälle werden innerhalb jeder Komponente eingegrenzt); Elastisch (auch bei wechselnder Arbeitslast reaktionsfähig; reagieren auf Änderung in Inputrate, indem Ressourcen erhöht oder verringert werden); Measseg-Driven (stützen sich auf asynchrones Message-Passing um Grenze zwischen Komponenten zu schaffen; diese Grenzen bietet die Möglichkeit, Fehler als Nachrichten zu delegieren); Vorteile: bessere Kontrolle über Reaktionszeiten; Konsistenz im Softwaredesign für Echtzeitsysteme, um Kosten und Aufwand zu reduzieren; Lastausgleich und Ausfallsicherheit; Herausforderungen: Hinzufügen von Beobachterprozessen zu aktueller Software kann schwierig oder unmöglich sein; reaktives Design bedeutet für Entwickler Bewusstseinswandel; reaktive Systeme können durch zu große Anzahl an Prozessen zu Verzögerungen führen; es gibt eine Lernphase, bei der mehr Validierung und Überwachung von Design und Implementierungen erforderlich sein kann
* Reactive Streams: Standard für asynchrone Stream-Verarbeitung mit nicht blockierendem Backpressure; z.B. RxJava oder Akka-Streams; Publisher (Methode subscribe()); Subscriber (kann als Senke beschrieben werden, vier Methoden: onSubscribe(), onNext(), onError(), onComplete()); Subscription (Klasse die zur Steuerung des Nachrichtenflusses verwendet wird); Processor (wandelt Nachrichten um und gibt sie weiter, fungiert als Publisher und Subscriber)
* Bibliotheken: ReactiveX (verwendet Observables um asynchrone Datenströme darzustellen, implementiert funktionale reaktive Implementierung in vielen Sprachen, RxJava, RXJs, RxScala); Webflux (kann direkt in Spring-Anwendungen verwendet werden)
* Backpressure: definiert wird Übertragung von Stream-Elementen reguliert werden kann; es wird gesteuert wie viele Elemente der Empfänger konsumieren kann

Stream API

* Stream-Konzept in Java erlaubt eine Art Datastream-Processing auf dem Operationen auf alle Elemente einer Datenstruktur angewendet werden
* Drei Typen von Operationen auf Streams unterschieden: Daten und Stream (Create-Operation), Operation1 … Operation n (Intermediate-Operation), Ereignis (Terminal-Operation)
* Create-Operationen: Umwandlung Array -> Stream; Umwandlung Collection -> Stream
* Intermediate-Operationen
* Terminal-Operationen
* Abgrenzung von Reactive Programming: Stream-API ist blockierend; Verarbeitung erfolgt synchron; Ausführung ist auf eine Verarbeitungskette beschränkt; Stream-API basiert nicht auf Reactive Streams; hier ist kein Observieren von Änderungen im Vordergrund

Webflux

* Spring WebFlux ist Framework für Entwicklung reaktiver und nicht-blockierender Webanwendungen
* Es unterstüzt und verwendet reaktive Streams
* Mono: spezieller Typ von Publisher; Mono-Objekt repräsentiert einen einzelnen oder leeren Wert
* Flux: Publisher, der 0 bis N asynchrone Sequenzwerte darstellt; er kann 0 bis viele, möglicherweise unendlich Werte ausgeben
* Terminal-Operationen: zum Lauschen auf Ereignisse; z.B. subscribe()
* Typische Operatoren: Creation (Just; Interval; From; Create und Generate; Sink; Hot Publisher (emittieren Ereignisse bevor es einen Subscriber gibt); Cold Publisher (emittieren Ereignisse erst wenn es einen Subscriber gibt); Filtering (Filter; Sample und SampleFirst; Distinct; Skip); Math (Count; Reduce); Combination (Concat; Merge; Zip; Combine Latest); Transformation (Map; Buffer; Window; FlatMap; concatMap; groupBy)
* Scheduler: Abstraktion in Reactor, die dem Benutzer Kontrolle über Threading gibt
* Debugging: nicht einfach Breakpoints in Lambda-Ausdrücke zu platzieren; was hilft: Umbau des Quellcodes; Hooks.onOperatorDebug(); Log-Operator

Vor- und Nachteile:

* Vorteile: Lazyness (Berechnungen werden nur bei Bedarf/Subscription ausgführt); Prägnanter Code (Implementierungen werden prägnanter und kompakter); Backpressure (damit kann gesteuert werden, wie viele Ereignisse verarbeitet werden sollen); Reduktion von Calbacks (asynchrone Programmierung ist vereinfacht und kann mittels reaktiver Programmierung einfach umgesetzt werden); Maintainability (Wartbarkeit von Quellcode kann sich verbessern)
* Nachteile: Lernkurve (Verständnis von Operatoren und kreativer Einsatz nötig); Debugging und Maintainability (Fehler unter Umständen schwierig zu verfolgen); Speicher (Speicher kann voll laufen aufgrund verschiedener Operatoren und Mechanismen); Fehlerbehandlung (Behandlung von Fehlern die während Laufzeit auftreten können)

Pitfalls

* Aspekte zu berücksichtigen: Verarbeitung startet mit Subscription; Exceptions (werden anders behandelt als zu erwarten wäre); Verschachtelte Datenreihen; State Management; Parallelism (Operatoren können zu ungewollter Parallelität führen)

4. Enterprise Integration Patterns

Ziele: Erklären was Enterprise Integration Patterns sind

Beispiele für typische Integrationsaufgaben beschreiben

Notation zur Darstellung von Integrationsansätzen verwenden, um Konzepte darstellen zu können

Notation verwenden, um damit Ansätze umzusetzen

Basics

* Enterprise Integration Patterns: bieten eine Menge von Patterns organisiert in Pattern-Language; jedes Pattern steht für Entscheidung die getroffen werden muss und für Überlegungen, die zu dieser Entscheidung führen; Pattern-Language ist Netz zusammenhängender Patterns
* Beispiele: Informationsportale (fassen Informationen aus mehreren Quellen in einer einzigen Anzeige zusammen); Data Replication (mehrere Geschäftssysteme benötigen Zugriff auf dieselben Daten); Shared Functions (redundante Funktionen zu implementieren ist nach Möglichkeit zu vermeiden); Service Oriented Architecture (gemeinsam genutzte Geschäftsfunktionen werden oft als Dienste bezeichnet und in einer Architektur organisiert); Distributed Business Process (Geschäftsvorgange sind oft über viele verschiedene Systeme verteilt); Business-to-Business Integration (Geschäftsfunktionen können für externe Anbieter oder Geschäftspartner zur Verfügung stehen)

Patterns

* Message Channel: Anwendungen über Nachrichtenkanal verbunden, eine schreibt Info in den Kanal, andere liest aus
* Message: Informationen sind in Nachrichten verpackt, die über Kanäle übertragen werden können (besteht aus Header und Body)
* Pipes and Filter: wird verwendet um größere Verarbeitungsaufgaben in eine Abfolge kleinerer. Unabhängiger Schritte zu unterteilen
* Message Translator: spezieller Filter zwischen anderen Filtern oder Anwendungen, um ein Datenformat in ein anderes zu übersetzen
* Message Endpoint: verbindet eine Anwendung mit einem Kanal den die Anwendung zum Senden oder Empfangen von Nachrichten verwenden kann
* Point-to-Point Channel: Nachricht über Punkt-zu-Punkt-Kanals zu versenden, der sicherstellt, dass nur ein Empfänger eine Nachricht erhält
* Publish-Subscribe Channel: Ereignisse werden über Publish-Subscribe Channel versendet, der jedem Empfänger eine Kopie eines bestimmten Ereignisses liefert
* Message Router: konsumiert Nachricht aus Nachrichtenkanal und veröffentlicht sie in Abhängigkeit von einer Reihe von Bedingungen in einem anderen Kanal wieder; mit mehreren Ausgangskanälen verbunden; verändert nicht den Inhalt, sondern kümmert sich nur um das Ziel der Nachricht
* Content-Based Router: Nachricht wird auf Grundlage ihres Inhalts weitergeleitet
* Message Filter: eliminiert unerwünschte Nachrichten aus einem Kanal auf Grundlage einer Reihe von Kriterien
* Recipient List: Empfängerliste verwenden, um eingehende Nachricht zu prüfen, Liste der gewünschten Empfänger zu bestimmten und Nachricht an alle Kannäle weiterzuleiten, die mit den Empfängern in der Liste verbunden sind
* Splitter: Nachricht in Einzelnachrichten aufteilen, die jeweils Daten zu einem Element enthalten
* Aggregator: sammelt einzelne Nachrichten bis er vollständigen Satz zusammengehöriger Nachrichten erhält; dann wird einzelne zusammengesetzte Nachricht veröffentlicht
* Invalid Message Channel: Empfänger verschiebt unzulässige Nachricht in Kanal für ungültige Nachrichten; spezieller Kanal für Nachrichten, die von Empfänger nicht verarbeitet werden können
* Guaranteed Delivery: sicherstellen, dass Nachrichten auch bei Ausfall des Nachrichtensystems nicht verloren gehen
* Command Message: Befehlsnachricht, um Prozedur in einer anderen Anwendnung aufzurufen
* Document Message: Dokumentnachricht, um Datenstruktur zuverlässig zwischen Anwendungen zu übertragen
* Event Message: Verwendung für zuverlässige, asynchrone Ereignisbenachrichtigung zwischen Anwendungen
* Request-Reply: verwenden, um Antwort von Empfänger zu erhalten
* Message Sequence: große Datenmenge als Nachrichtenfolge versenden und jede Nachricht mit Sequenzidentifikationsfeldern kennzeichnen

5. Data-Stream-Processing

Ziele: Erklären was ein Stream und in diesem Kontext eine Relation ist

Erklären was es mit Stream-to-Relation, Relation-to-Relation, Relation-to-Stream-Operatoren auf sich hat und an einem Beispiel die Unterschiede darstellen

Verschiedene Arten von Fenstern und deren Notwendigkeit erklären

Stateful- und Staeteless-Operatoren unterscheiden

Kafka Streams anwenden können (z.B. in Pseudocode), um Data-Stream-Processing an einem Beispiel umzusetzen

Kafka Streams und Apache Flink differenzieren, insbesondere hinsichtlich der Art der Ausführung von Topologien und der Skalierung von Tasks

Motivation

* Stream: Menge von Tupeln in einem zeitlichen Kontext
* Basisdatenströme: Quelldatenströme, die beim Datenstrom-Management-System ankommen
* Abgeleitete Datenströme: Zwischenströme, die von Operatoren aus anderen Basisdatenströmen erzeugt werden
* Relation: ungeordnete Menge von Tupeln zu jedem Zeitpunkt
* Stream-to-Relation-Operator: Stream als Eingabe und erzeugt Relation als Ausgabe
* Relation-to-Relation-Operator: eine oder mehrere Relationen als Eingabe und erzeugt eine Relation als Ausgabe
* Relation-to-Stream-Operator: Relation als Eingabe und erzeugt Stream als Ausgabe mit demselben Schema wie die Relation
* Window: jüngsten Daten werden in einem Berechnungsfenster berücksichtigt, um ein Ergebnis auf Grundlage einer Datenuntermenge zu erhalten
* Stateless-Operatoren: z.B. Filtern auf Grundlage eines Attributwerts, verarbeiten Datenströme Element für Element; liefern ein neues Ergebnis mit unvorhersehbarer Häufigkeit; haben keine Informationen über die aktuelle Iteration des Berechnungsfensters oder früherer Ergebnisse, wenn sie Ergebnis aus Datenstromelement berechnen
* Stateful-Operatoren: verwenden als Eingabe logischen Block von Stream-Elementen, um einziges Ergebnis für gesamte Menge zu berechnen; Stream-Element-Block kann durch Bereich von Zeitstempeln oder Anzahl von zu berücksichtigenden Elementen definiert werden; speichern Informationen wie aktuelle Iteration des Fensters oder Liste der verschiedenen Elementwerte, um ein Ergebnis aus allen berücksichtigten Elementen zu verarbeiten; während der Laufzeit erzeugten Informationen werden als sein Zustand bezeichnet
* MapReduce: Verarbeitung großer Datenmengen lass sich einfacher für parallele Umgebungen auf- und verteilen

Kafka Streams

* Arbeiten als Schicht im Kafka-Ökosystem damit Daten aus verschiedenen Quellen zusammenlaufen; stellt High-Level-DSL bereit
* Skalierung: über Consumer-Groups und Partitionierung; Stream-Knoten erlaubt Verarbeitung von Daten innerhalb zugewiesener Partitionen; Consumer-Groups können parallel und unabhängig auf denselben Daten arbeiten
* Zuverlässigkeit: bringt über Partitionen und automatisierte Neuverteilung bei Ausfall eines Consumers innerhalb einer Gruppe Failover-Strategie mit; fällt ein Knoten aus, werden Partitionen auf verbleibende Knoten verteilt
* Wartbarkeit: Stream-Anwendungen als eigener Java-Prozess gestartet; Quellcode durch DSL ähnlich zu Streaming- und Reactive-Ansätzen; Ansätze zur Fehlerbehebung und Überwachung können hier wie üblich Anwendung finden
* Processing Topologies: Source-Prozessoren (über Source fließen Infos in die Kafka-Streams-Anwendungen), Stream-Prozessoren (für Datenverarbeitungs-/Transformationslogik auf Input-Stream verantwortlich), Sink-Prozessoren (verarbeitete Datensätze werden zurück nach Kafka geschrieben; wie Quellprozessoren auch mit einem Kafka-Topic verbunden)
* Sub-Topologies: möglich Topologien zu unterteilen; Zwischenergebnisse werden in Queues geschrieben, die anschließend mit eigener Verarbeitungslogik behandelt werden
* Depth-First Processing: neuer Datensatz wird durch jeden Stream-Prozessor in Topologie geleitet, bevor weiterer Datensatz verarbeitet wird
* SERDES: Serializer und Deserializer; für die Verarbeitung von Daten und Topics und Veröffentlichung auf Topics notwendig
* Stateless-Processing: Filter (filter); Verzweigen (split); Transformieren (map oder mapValues); Transformieren in ein oder mehrere Records (flatMap oder flatMapValues); Zusammenführen (merge)
* Sink-Processors: To (aktuellen Stream in einem Topic platzieren); Repartition (aktuelle Stream in einer auto-generated Topic platzieren und direkt in einem neuen Stream verwenden)
* Stateful-Processing: State Stores (Zustandsspeicher); KStream (meist bei schküssellosen Ereignissen, um Verarbeitungsketten auf Records zu realisieren), KTable und GlobalKTable (Zugriff auf Datenströme in Form einer Tabelle, KTable erstellt Partitionen und nutzt Keys, GlobalKTable gibt allen Consumern kompletten Datenbestand (bietet sich nur für überschaubare Menge an))
* Join: Möglichkeit verteilte Daten zusammenzuführen
* Co-Partitioning: jede Partition wird genau einem Consumer zugewiesen; Datensätze auf beiden Seiten müssen durch dasselbe Key-Feld und unter Verwendung derselben Partitionierungsstrategie partitioniert werden; Co-Partitioning notwendig außer bei KStream-GlobalKTable (Join des Streams mit einer Tabelle die vollständig in jeder Instanz läuft)
* Rekeying: selectKey; KStream kann in richtige Partition organisiert werden
* GroupBy und GroupByKey: repartitionierende Operation
* Aggregation: aggregate-Operator; besteht aus Initializer und Aggregator

Windows (Arten und Notwendigkeit):

* Tumbling Windows: fester Größe, überlappen sich nicht; werden nur mit Fenstergröße definiert und haben vorhersehbare Zeitbereiche
* Overlapping Windows: fester Größe, können sich überlappen; mit Fenstergröße und Vorrückungsintervall definiert
* Session Windows: variabler Größe, durch Aktivitätsperioden bestimmt; Inaktivitätslücke wird zur Definition verwendet
* Sliding Join Windows: fester Größe für Joins verwendet; zwei Datensätze fallen ins selbe Fenster, wenn Unterschied zwischen ihren Zeitstempeln kleiner oder gleich der Fenstergröße ist
* Grace: Ereignisse können verzögert im Stream empfangen werden; Grace-Angabe kann im Code gemacht werden
* Suppression: verhindert, dass jedes Ereignis einzeln weitergegeben wird; stattdessen lieber Gesamtergebnis weitergeben

Apache Flink

* 1:1-Streams: behalten Aufteilung und Reihenfolge der Elemente bei; Subtask des map()-Operators sieht gleichen Elemente in gleicher Reihenfolge wie von Subtask des Source-Operators erzeugt
* Redistributing: ändert Aufteilung der Datenströme; jede Operator-Teilaufgabe sendet Daten an verschiedene Ziel-Teilaufgaben, abhängig von gewählter Transformation
* Datenformate: verschieden Formate können von DataStream-APIs von Flink verarbeitet werden: Grundtypen (String, Long, Integer, Boolean und Array) und zusammengesetzte Typen (Tuples, POJOs und Scala Fallklassen); für andere Typen greift Flink auf Kryo zurück
* Zustandsverwaltung: lokal; dauerhaft: wird in regelmäßigen Abständen überprüft und bei Ausfall wiederhergestellt; vertikal skalierbar: kann in eingebetteten RcoksDb-Instanzen gehalten werden; horizontal skalierbar: wird umverteilt, wenn Cluster wächst oder schrumpft; abfragbar: kann von außen über Queryable State API abgefragt werden
* Watermarks: zum Ordnen von Ereignissen nach Zeitstempeln; definieren wann das Warten auf frühere Ereignisse beendet werden soll; Flink verfügt über mehrere eingebauten Typen von Fensterzuweisern
* Process Functions: kombiniert Ereignisverarbeitung mit Zeitgebern und Zuständen; Grundlage für Erstellung ereignisgesteuerter Anwendungen mit Flink; sehr änhlich zu RichFlatMapFunticon aber mit Zusatz des Zeitbezugs

Vergleich

* Apache Storm: älteste Open-Source-Streaming-Framework (Vorteile: sehr niedrige Latenz, echtes Streaming, ausgereift, hoher Durchsatz, hervorragend geeignet für nicht komplizierte Streaming-Anwendungsfälle; Nachteile: keine Zustandsverwaltung, keine erweiterten Funktionen, mindestens-einmal-Garantie)
* Apache Spark Streaming: erstes Framework, das die Lambda-Architektur vollständig unterstützt (Vorteile: unterstützt Lambda-Architektur, hoher Durchsatz, standardmäßige Fehlertoleranz, einfach zu verwendende APIs auf höherer Ebene, exactly once, große Community; Nachteile: nicht geeignet für niedrige Latenzanforderungen, zu viele Parameter zum Abstimmen, von Natur aus zustandslos, bleibt in viele fortgeschrittenen Funktionen hinter Flink zurück)
* Apache Flink: unterstützt auch Lambda-Architektur (erstes Streaming-Framework mit allen fortschrittlichen Funktionen, niedrige Latenz mit hohem Durchsatz, automatische Anpassung, exactly once; Nachteile: Community nicht so groß wie bei Spark, keine Annahme von Flink Batch bekannt, nur beliebt für Streaming)
* Kafka Streams: leichtgewichtige Bibliothek (gut für Microservices, vererbt alle guten Eigenschaften von Kafka, exaclty once, unterstützt Stream-Joins; Nachteile: kann nicht ohne Kafka verwendet werden, neu im Anfangsstadium, nicht für schwere Arbeiten wie Spark Streaming, Flink)

6. Query Languages

Ziele: Unterschiede zwischen SQL und ksqlDb erklären

Grundlegenden Aufbau von DSL und DML Queries in ksqlDB erklären

Vorgehen zur Verarbeitung von Datenströmen mit ksqlDB darstellen

ksqlDB Queries und deren Bestandteile erklären sowie in Kafka Stream übersetzen können

ksqlDB

* Open-Source-Event-Streaming-Datenbank
* Es wird eine High-Level-SQL-Schnittstelle für Interaktion mit Kafka angeboten
* SQL-Grammatik lehnt sich an ANSI SQL an, jedoch spezielle SQL-Dialekt abgeleitet, um Daten sowohl in Streams als auch in Tabellen zu modellieren
* Gemeinsamkeiten zwischen ksqlDb und SQL: SQl-Schnittstelle; DDL- und DML-Anweisungen; Netzwerkdienste und Clients zur Übermittlung von Abfragen; Schemas; Materialisierte Ansichten; eingebaute Funktionen und Operatoren für Datentransformation; Datenreplikation
* Unterschieden zwischen ksqlDb und SQL: verbesserte DDL- und DML-Anweisungen (beziehen sich auf Tabellen und Datenströme); Push-Abfragen; einfache Abfragefunktionen; ANSi-inspiriertes SQL, aber nicht vollständig konform; lokale und entfernte Speicherung; Konsistenzmodell
* ksqlDb Server: verantwortlich für Ausführung Ihrere Stream-Processing-Anwendungen; Arbeitslasten können über mehrere Server mit derselben Konfiguration verteilt werden; jeder Server vergleichbar mit einzelner Instanz einer Kafka-Streams-Anwendung
* ksqlDB-Cluster: Gruppe von kooperierenden ksqlDb-Servern
* SQL-Engine: verantwortlich für Parsen einer SQl-Anweisung, Konvertierung in eine oder mehrere Kafka-Sterams-topologien und Ausführung der Kafka-Streams-Anwendung
* REST service: ksqlDB enthält eine REST-Schnittstelle, über die Clients mit SQL-Engine interagieren können
* ksqlDB CLI: Befehlszeilenanwendung, mit der man mit einem laufenden ksqlDB-Server interagieren kann
* ksqlDB UI: Feature von Confluent für die Interaktion mit ksqlDB
* Deployment Mode (Interactive Mode): im interaktiven Modus, können Clients jederzeit neue Abfragen über REST-API übermitteln; Standardmodus
* Deployment Mode (Headless Mode): REST-API ist deaktiviert