Autores: André Y. Matsuda e Adriano de Souza Barbosa

Processamento de Streamings em Tempo Real para Big Data

1. Introdução

Através do avanço tecnológico e da crescente conectividade entre pessoas e dispositivos, a quantidade de dados disponíveis para empresas (web), governos e outras organizações está constantemente crescendo. A mudança para a presença de um conteúdo mais dinâmico e gerado pelo usuário na web e a onipresença de telefones inteligentes e outras modalidades tecnológicas levaram a uma abundância de informações que são valiosas apenas por um curto período de tempo, portanto, devem ser imediatamente processados. Empresas como a Amazon e a Netflix já se adaptaram e estão monitorando a atividade do usuário para otimizar as recomendações de produto ou vídeo. Twitter realiza análise de sentimento contínuo para informar os usuários sobre tendências e até mesmo o Google tem utilizado o processamento em lote/batch para indexar a web.

No entanto, a ideia de processar dados em movimento não é nova: Processamento de Eventos Complexos (CEP) [11 , 12] e DBMSs com recursos de consulta contínua [36] podem prover latência de processamento na ordem de milissegundos e geralmente expõem interfaces de alto nível, interfaces SQL e sofisticadas funcionalidades de consulta como joins. Mas, embora típica implantação desses sistemas não ultrapasse alguns nós, os sistemas focados neste artigo foram concebidos especificamente para implantações com 10 s ou 100 s de Nós. Muito parecido com MapReduce, o principal objetivo desses novos sistemas é a abstração de problemas de dimensionamento e possibilitar o desenvolvimento, implantação e manutenção de sistemas altamente escaláveis viáveis.

A seguir, apresentamos uma visão geral de alguns dos sistemas de processamento de fluxo distribuído mais populares atualmente disponíveis, destacando semelhanças, diferenças e trade-offs tomadas em seus respectivos projetos.

2. Análise em tempo real: Big Data em movimento

Em contraste com os sistemas tradicionais de análise de dados que coletam e processam periodicamente enormes - estáticos – volumes de dados, sistemas analíticos do tipo streaming evitam armazenar os dados e processam os mesmos à medida que se tornam disponíveis, minimizando o tempo que um único item de dados gasta no processamento em fila. Sistemas que rotineiramente conseguem latências de segundos ou mesmo sub segundos entre a recepção dos

dados e a produção de resultados são muitas vezes descritos como "real-time". No entanto, grandes partes da infraestrutura Big Data são construídos a partir de componentes distribuídos que se comunicam através de uma rede assíncrona e são projetados em cima de uma JVM (Java Virtual Machine). Assim, estes sistemas são apenas soft-real-time e nunca fornecem um tempo superior adequado para produzir uma saída.

A Figura 1 ilustra típicas camadas de um fluxo contínuo de dados. Dados como cliques do usuário, informações de faturamento ou conteúdo não estruturado, como imagens ou mensagens de texto são recolhidos em vários locais dentro de uma organização e então movidos para a camada de streaming (por exemplo, um sistema de enfileiramento como Kafka, Kinesis ou ZeroMQ) a partir do qual é acessível a um processador de fluxo que executa uma determinada tarefa para produzir uma saída. Esta saída é então encaminhada para o servidor que pode ser, por exemplo, uma web GUI analítica contendo tendências sobre tópicos no Twitter ou um banco de dados onde uma visão materializada é mantida.

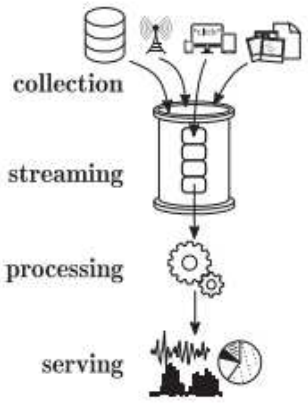
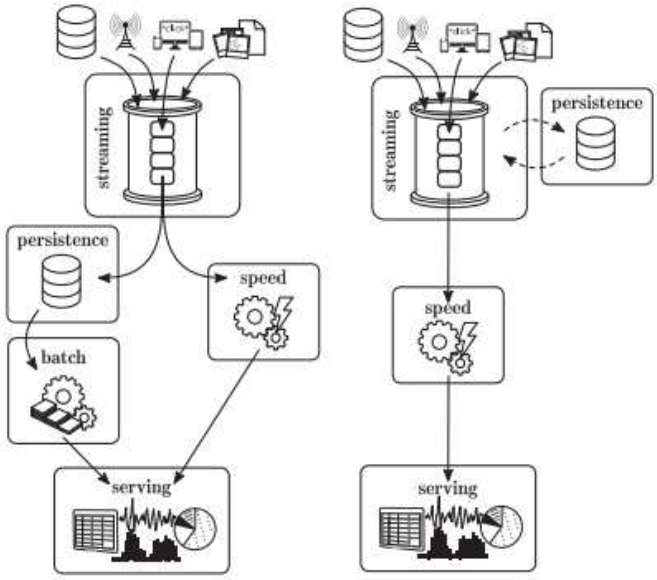


Figura 1: Visualização do sequenciamento de um Stream

Na tentativa de combinar o melhor dos dois mundos, um Padrão arquitetônico chamado Arquitetura Lambda [32] tornou-se bastante popular e complementa o lento processado em lotes adicionando a capacidade de tempo real e, portanto, tratando, ao mesmo tempo, dois alvos do Big Data [29]: o volume e a velocidade.

Tal como ilustrado na Figura 2 A, a Arquitetura Lambda descreve um sistema que compreende três camadas: Os dados são armazenados em uma camada de persistência como HDFS a partir da qual é ingerida e processada em lote periodicamente (por exemplo, uma vez ao dia), enquanto a camada de velocidade controla a parte dos dados que ainda não foram processados pela camada de lote. A camada de serviço consolida ambos pela fusão da saída do lote e da camada de velocidade. O benefício óbvio de se ter um sistema em

tempo real para compensar a latência de processamento em lote é pago pelo aumento da complexidade do desenvolvimento, implantação e manutenção.
E se a camada de lote é implementada com um sistema que suporta processamento de lote e de Streaming (como por exemplo Spark), a camada de velocidade muitas vezes pode ser implementada usando a API de fluxo correspondente (por exemplo, Spark Streaming) para lidar com a lógica empresarial existente e a Implantação existente. Para sistemas baseados em Hadoop e outros que não fornecem uma API de streaming, no entanto, a camada de velocidade só está disponível como um sistema separado. Usando uma linguagem abstrata como Summingbird [18] para escrever a lógica de negócios, é possível a compilação automática de código tanto para o sistema de lote quanto para o processamento de streaming (por exemplo, Hadoop e Storm), facilitando o desenvolvimento nos casos em que o processamento em lote e a camada de velocidade usam a mesma lógica de negócios, mas ainda assim com a sobrecarga para implantação e manutenção.



(a) The Lambda Architecture. (b) The Kappa Architecture.

Figura 2: Arquitetura Streaming

Outra abordagem que, ao contrário, dispensa a camada de lote em favor da simplicidade é conhecida como a Arquitetura Kappa [25] e é ilustrada na Figura 2 B. A ideia básica é não recalcular periodicamente todos os dados na camada de lote, mas fazer todos os cálculos no sistema de processamento streaming sozinho e apenas executar o recálculo quando as regras de negócio mudarem, repondo assim os dados históricos. Para isso, a Kappa Architecture emprega um poderoso processador de stream capaz de lidar com a cópia de dados em uma taxa muito maior do que a de entrada e um sistema Streaming para retenção de dados. Um exemplo de um sistema de streaming é o Kafka que foi especificamente projetado para trabalhar com o processador de streaming Samza neste tipo de arquitetura. Arquivar dados (por exemplo, em HDFS)

ainda é possível, mas não faz parte do caminho crítico e muitas vezes não requerido, pois o Kafka, por exemplo, suporta tempos de retenção em ordem das semanas. Em contrapartida, no entanto, o esforço necessário para reprocessar todo o histórico aumenta linearmente com o volume de dados e a abordagem ingênua de manter o completo Fluxo de mudança pode ter um aumento significativo de armazenamento do que periodicamente processar novos dados e atualizar uma base de dados existente, dependendo de quão eficiente os dados são compactados no streaming. Como consequência, a arquitetura Kappa deve ser considerada uma alternativa à arquitetura Lambda em aplicações que não necessitem de retenção, ou permitem uma compactação eficiente (por exemplo, é razoável manter apenas o valor mais recente para cada combinação chave e dado).

Naturalmente, a latência exibida pela Camada de velocidade sozinha é apenas uma fração da latência end-to-end da aplicação devido ao impacto da rede ou outros sistemas no pipeline. Mas é obviamente um fator importante e pode determinar qual sistema escolher em aplicações com SLAs de tempo restrito. Este artigo concentra-se nos sistemas disponíveis para a camada de processamento de streaming.

3. Processadores em tempo real

Embora todos os processadores Stream compartilhem os mesmos conceitos e princípio de funcionamento, uma distinção importante entre os sistemas individuais é a latência, representada no modelo de processamento ilustrado na Figura 3: Manuseando itens de dados imediatamente à medida que eles aparecem, minimiza a latência, enquanto que o buffer e o processamento em batch produzem uma maior eficiência, mas obviamente aumenta o tempo que o item individual gasta no pipeline de dados. Sistemas puramente orientados como Storm e Samza fornecem latência muito baixa e custo por item relativamente alto, enquanto os sistemas de batch alcançam eficiência de recursos sem precedentes, mas à custa da uma maior latência, o que o torna proibitivamente alto para aplicações em tempo real.

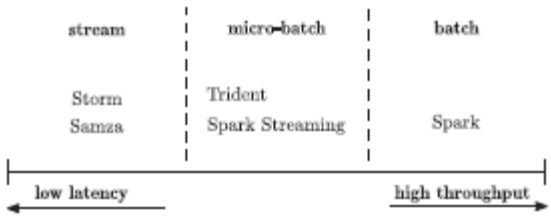


Figura 3: Relação de Latência para diferentes modelos de processamento

O espaço entre estes dois extremos é vasto e alguns sistemas como Storm Trident e Spark Streaming empregam estratégias de micro-batch. Grupos Trident agrupa tuplas em batches para amenizar o modelo de processamento one-at-a-time em favor do aumento da taxa de transferência, enquanto o Spark Streaming restringe o tamanho do lote em um processador para reduzir a latência. A seguir, analisamos detalhadamente as especificidades dos sistemas acima mencionados e destacamos os compromissos inerentes e as decisões tomadas em cada projeto.

3.1. Storm:

Storm (atual versão estável: 1.0.0) está em desenvolvimento desde o final de 2010, foi open-sourced em setembro de 2011 pelo Twitter e, tornou-se um projeto Apache em 2014. É o primeiro sistema de processamento de fluxo distribuído utilizado para pesquisa e prática e foi inicialmente promovido como o "Hadoop de tempo real" [30, 31], porque seu modelo de programação forneceu uma abstração para processamento Stream semelhante à abstração que o paradigma MapReduce prevê para processamento em batch. Mas além de ser o primeiro de seu tipo, Storm também tem uma ampla base de usuários devido à sua compatibilidade com praticamente qualquer idioma: Além da API Java, Storm também é compatível com Thrift e vem com adaptadores para várias linguagems como Perl, Python e Ruby.]

Storm pode correr em cima de Mesos [5], como um cluster dedicado ou até mesmo em uma única máquina. As partes vitais de uma implantação Storm são um cluster ZooKeeper para coordenação confiável, vários supervisores para execução e um servidor Nimbus para distribuir código em todo o cluster e tomar medidas em caso de falha ; Para se proteger contra uma falha em um servidor Nimbus, o Storm permite ter várias instâncias Nimbus em espera. Storm é escalável, tolerante a falhas e até elástico, pois o trabalho pode ser redistribuído durante o tempo de execução. A partir da versão 1.0.0, Storm fornece implementações confiáveis que sobrevivem e se recuperam da falha do supervisor. As versões anteriores focavam o processamento e portanto, requerem gerenciamento no nível de aplicação para obter tolerância a falhas e elasticidade em aplicações. Storm se destaca em velocidade e, portanto, é capaz de executar milissegundos de um dígito em determinadas aplicações. Por meio do impacto da latência da rede e da coleta de lixo, no entanto, as topologias do mundo real geralmente não exibem latência de ponta a ponta abaixo de 50ms [23, Capítulo 7].

Um pipeline de dados ou aplicação em Storm é chamado de topologia e como ilustrado na Figura 4 é um grafo direcionado que representa o fluxo de dados

como bordas direcionadas entre nós que novamente representam os passos de processamento individuais: Os nós que ingerem dados e assim iniciam o fluxo de dados na Topologia são chamados bicos (spouts) e emite tuplas para os nós downstream que são chamados parafusos (bolts) e fazem o processamento, escrevem dados para armazenamento externo. O Storm vem com vários agrupamentos que controlam o fluxo de dados entre nós, mas também permite agrupamentos personalizados arbitrários. Por padrão, o Storm distribui bicos e parafusos entre os nós do cluster, embora seja adaptável para cenários específicos em que uma determinada etapa de processamento deve ser executada em um nó específico, por exemplo por causa de dependência de hardware.

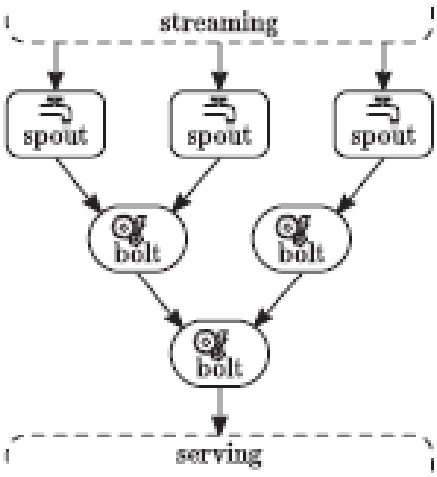


Figura 4: Fluxo de Dados na metodologia Storm

Embora a Storm não forneça nenhuma garantia na ordem em que os tuplos são processados, ela fornece a opção de pelo menos uma vez de processamento através de um recurso de reconhecimento que rastreia o status de processamento de cada tupla única em seu caminho através da topologia: Storm repetirá uma tupla, se algum parafuso envolvido no processamento dele explicitamente sinaliza falha ou não reconhece processamento bem-sucedido dentro de um determinado período de tempo.

Usando um sistema de fluxo apropriado, é mesmo possível proteger contra falhas de bico, mas o recurso de reconhecimento muitas vezes não é usado na prática, porque a sobrecarga de mensagens impostas pelo rastreamento da linhagem tupla, ou seja, uma tupla e todas as tuplas que são emitidos em seu nome, prejudica visivelmente a taxa de rendimento do sistema. [20]

Com a versão 1.0.0, a Storm introduziu um mecanismo de contrapressão para acelerar a ingestão de dados como último recurso sempre que os dados são ingeridos mais rapidamente do que podem ser processados. Se o processamento se tornar um gargalo em uma topologia sem esse mecanismo, o throughput se

degrada com tuplas eventualmente dando time-out e se perdendo (no máximo uma vez de processamento). Podem ocorrer repetições, mas que provavelmente colocarão ainda mais carga em um sistema já sobrecarregado.

3.2. Storm Trident

No Outono de 2012 e na versão 0.8.0, o Trident foi lançado como uma API de alto nível com garantias de sequenciamento e uma interface de programação mais abstrata com suporte integrado para junções, agregações, agrupamento, funções e filtros.

Em contraste com Storm, as topologias Trident são direcionadas a grafos acíclicos (DAGs), pois não suportam ciclos; isso os torna menos adequados para a implementação de algoritmos iterativos e também é uma diferença para as topologias planas Storm, que muitas vezes são erroneamente descritas como DAGs, pois na verdade podem introduzir ciclos.

Além disso, Trident não funciona em tuplas individuais, mas em micro lotes e, consequentemente, introduz o tamanho do lote como um parâmetro para aumentar a taxa de transferência ao custo de latência, que, no entanto, pode ser tão baixo quanto vários milissegundos para pequenos lotes. Todos os lotes são, por padrão processados em ordem sequencial, um após o outro, embora Trident também pode ser configurado para processar vários lotes em paralelo. Além da escalabilidade e da elasticidade da Storm, a Trident fornece sua própria API para gerenciamento de estado tolerante a falhas com semântica de processamento única. Mais detalhadamente, o Trident evita a perda de dados usando o recurso de reconhecimento do Storm e garante que cada tupla é refletida apenas uma vez em estado persistente, mantendo informações adicionais ao lado do estado e aplicando as atualizações de forma transacional.

A partir da escrita, estão disponíveis duas variantes de gerenciamento de estado: Um só armazena o número de sequência do último lote processado juntamente com o estado atual, mas pode bloquear toda a topologia quando uma ou mais tuplas de um lote falhado não podem ser reproduzidas (por exemplo, devido à indisponibilidade Da fonte de dados), enquanto o outro pode tolerar esse tipo de falha, mas é mais pesado pois também armazena o último estado conhecido.

Independentemente de os lotes serem processados em paralelo ou um por um, as atualizações de estado têm de ser mantidas em ordem para garantir a semântica correta. Como consequência, seu tamanho e frequência pode se tornar um gargalo e Trident pode, portanto, ser apenas viável para gerenciar pequenos estados.

3.3. Samza

Samza (atual versão estável: 0.10.0) é muito semelhante ao Storm, já que é um processador de fluxo com um modelo de processamento one-at-atime e uma semântica de processamento at-least-once. Foi inicialmente criado no LinkedIn, submetido à Incubadora Apache em julho de 2013 e foi concedido status de toplevel em 2015. Samza foi co-desenvolvido com o sistema de filas Kafka [4, 26] e, portanto, confia na mesma semântica de mensagens: Streams são particionados e as mensagens (ou seja, itens de dados) dentro da mesma partição são ordenadas, enquanto que não há ordem entre as mensagens de diferentes partições. Mesmo que Samza possa trabalhar com outros sistemas de enfileiramento, as capacidades de Kafka são efetivamente requeridas para usar Samza em todo seu potencial e, portanto, é assumido para ser implantado com Samza para o resto desta seção.

Em comparação com o Storm, Samza requer um pouco mais de trabalho para implementar, pois não depende apenas de um cluster ZooKeeper, mas também é executado em cima do Hadoop YARN [6] para tolerância a falhas: Em essência, a lógica da aplicação é implementada como um trabalho que é enviado através do cliente Samza YARN que tem o YARN, então inicia e supervisiona um ou mais contêineres.

A escalabilidade é conseguida através da execução de um trabalho Samza em várias tarefas paralelas cada uma das quais consumindo uma partição separada de Kafka; O grau de paralelismo, ou seja, o número de tarefas, não pode ser aumentado dinamicamente em tempo de execução.

Semelhante a Kafka, Samza se concentra no suporte para linguagem JVM, particularmente Java. Contrastando Storm e Trident, Samza é projetado para lidar com grandes quantidades de estado de uma forma tolerante a falhas, persistindo estado em um banco de dados local e replicando atualizações de estado para Kafka. Por padrão, a Samza emprega um armazenamento de valor-chave para essa finalidade, mas outros mecanismos de armazenamento com recursos de consulta mais ricos podem ser conectados.

Conforme ilustrado na Figura 5, um trabalho de Samza representa um passo de processamento num pipeline analítico e assim corresponde aproximadamente a um parafuso na topologia Storm. Em contraste com Storm, no entanto, onde os dados são enviados diretamente de um parafuso para outro, o resultado produzido por um trabalho Samza é sempre escrito de volta para o Kafka de onde ele pode ser consumido por outros trabalhos Samza.

Embora um único trabalho Samza ou uma única persistência Kafka possa atrasar uma mensagem em apenas alguns milissegundos [8, 24], a latência

acrescenta-se em pipelines de análise complexa (compreendendo várias etapas de processamento), o que eventualmente pode causar latência de ponta a ponta mais elevada quando comparáveis a implementações com Storm.

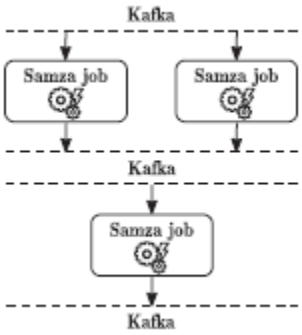


Figura 5: Fluxo de Dados na metodologia Samza. Samza Jobs não podem se comunicar diretamente, mas precisam utilizar um sistema de fila como o Kafka

No entanto, este projeto também desacopla etapas de processamento individuais e, assim, facilita o desenvolvimento. Outra vantagem é que o armazenamento em buffer de dados entre etapas de processamento torna os resultados (intermediários) disponíveis para partes não relacionadas, ou seja, para outras equipes na mesma empresa, e ainda elimina a necessidade de um algoritmo de contrapressão, uma vez que não há perda de backlog de um determinado trabalho temporário, dado um tamanho razoável na implementação do Kafka.

Uma vez que a Samza processa as mensagens em ordem e armazena resultados de processamento após cada etapa, é capaz de impedir a perda de dados checando periodicamente o progresso atual e reprocessando todos os dados desse ponto em diante em caso de falha;

3.4. Spark Streaming

A estrutura Spark [39] (atual versão estável: 1.6.1) é um quadro de processamento batch que é muitas vezes mencionado como o sucessor não oficial de Hadoop, pois oferece vários benefícios em comparação, mais notavelmente um API mais conciso resultando em menos desenvolvimento de aplicação lógica e melhoria significativa de desempenho através do armazenamento em cache: Em particular os algoritmos iterativos (por exemplo, algoritmos de aprendizado de máquina como clustering *k*-means ou regressão logística) são acelerados por ordens de magnitude, porque os dados não são necessariamente escritos e carregados dos discos entre os passos do processamento.

Além desses benefícios de desempenho, o Spark fornece uma variedade de algoritmos de aprendizado de máquina out-of-the-box através da biblioteca MLlib. Originário da UC Berkeley em 2009, o Spark foi de fonte aberta em 2010 e foi doado para a Apache Software Foundation em 2013, onde se tornou um projeto de alto nível em fevereiro de 2014. É principalmente escrito em Scala e tem APIs Java, Scala e Python.

A abstração central do Spark são coleções distribuídas e imutáveis chamadas RDDs (conjuntos de dados distribuídos resilientes) que só podem ser manipuladas através de operações determinísticas. Spark é resistente a falhas de máquina, controlando a linhagem de qualquer RDD, isto é, a sequência de operações que a criou, e marcando pontos RDDs que são caros para recalcular, HDFS [35]. Uma implantação Spark consiste em um gerenciador de cluster para gerenciamento de recursos (e supervisão), um programa de driver para agendamento de aplicativos e vários nós operacionais para executar a lógica do aplicativo. Spark é executado em cima do Mesos, YARN ou no modo standalone, caso em que pode ser usado em combinação com o ZooKeeper para remover o nó mestre, ou seja, o gerenciador de cluster, como um único ponto de falha.

Spark Streaming [40] desloca a abordagem de processamento em lote (batch) da Spark visando atender aos requisitos em tempo real, através da fragmentação do fluxo de itens de dados de entrada em pequenos lotes, transformando-os em RDDs e processando-os como de costume.

Ele também cuida do fluxo de dados e distribuição automaticamente. Spark Streaming está em desenvolvimento desde o final de 2011 e tornou-se parte da Spark em fevereiro de 2013. Sendo uma parte da estrutura Spark, a Spark Streaming tinha uma grande comunidade de desenvolvedores e também um enorme grupo de usuários potenciais desde o primeiro dia, já que os dois sistemas compartilham a mesma API e são executados em cima de um cluster Spark comum. Assim, ele pode ser resiliente à falha de qualquer componente [37] como Storm e Samza, além de suportar dimensionamento dinâmico dos recursos alocados para um aplicativo. Os dados são ingeridos e transformados em uma sequência de RDDs que é chamado DStream (fluxo discretizado) antes de serem processados através dos trabalhadores.

Todos os RDDs em um DStream são processados em ordem, ao passo que os itens de dados dentro de um RDD são processados em paralelo sem garantias de sequenciamento. Uma vez que existe um determinado atraso de programação ao processar um RDD, os tamanhos de lote abaixo de 50ms tendem a ser inviáveis [10, seção "Ajuste de Desempenho"]. Conseqüentemente, o processamento de um RDD leva cerca de 100ms no melhor dos casos, embora

o Spark Streaming seja projetado para latência da ordem de alguns segundos [40, Sec. 2]. Para evitar a perda de dados, mesmo para fontes de dados não confiáveis, o Spark Streaming concede a opção de usar um write-ahead (WAL) log partir do qual os dados podem ser reproduzidos após falha. A administração do estado é realizada através de um DStream de estado que pode ser atualizado através de uma transformação DStream.

3.5. Discussão

A Tabela 1 resume as propriedades desses sistemas em comparação direta. Storm fornece baixa latência, mas não oferece garantias de sequenciamento e é frequentemente implementada sem garantias de entrega, uma vez que o reconhecimento de por-tupla necessário para processamento at-least-once efetivamente duplica a sobrecarga de mensagens.

Processamento de estado exaclty-once está disponível no Trident através de atualizações de estado, mas tem impacto notável no desempenho e tolerância a falhas em alguns cenários de falha.

Samza é outro processador de stream nativo que não foi orientado para baixa latência tanto quanto Storm e coloca mais foco em fornecer semântica rica, em particular através de um conceito embutido de gestão de estado.

Tendo sido desenvolvido para uso com Kafka na Arquitetura Kappa, Samza e Kafka são fortemente integrados e compartilham a mesma semântica de mensagens; Assim, Samza pode explorar plenamente as garantias de ordenação fornecidas por Kafka.

O Spark Streaming unifica o processamento em lote (batch) e stream e oferece uma API de alto nível, garantindo o processamento exactly-once e um rico conjunto de bibliotecas, o que pode reduzir a complexidade do desenvolvimento de aplicativos. No entanto, sendo um processador de lote nativo, Spark Streaming perde para seus concorrentes em relação à latência [20].

Table 1: Apache Storm/Trident, LinkedIn's Samza and Apache Spark Streaming in direct comparison.

	Storm	Trident	Samza	Spark Streaming
strictest guarantee	at-least-once	exactly-once	at-least-once	exactly-once
achievable latency	≪ 100 ms	< 100 ms	< 100 ms	< 1 s
state management	yes	yes (small state)	yes	yes
processing model	one-at-a-time	micro-batch	one-at-a-time	micro-batch
backpressure mechanism	yes	yes	not required (buffering)	yes
ordering guarantees	no	between batches	within stream partitions	between batches
elasticity	yes	yes	no	yes

Todos esses diferentes sistemas mostram que a baixa latência está envolvida em uma série de trade-offs com outras propriedades desejáveis, como throughput, tolerância a falhas, garantias de processamento de confiabilidade e facilidade de desenvolvimento. O throughput pode ser otimizado armazenando dados em buffer e processando-os em lotes para reduzir o impacto de mensagens e outros custos indiretos por item de dados, enquanto isso obviamente aumenta o tempo de vôo de itens de dados individuais. As interfaces abstratas escondem a complexidade do sistema e facilitam o processo de desenvolvimento de aplicativos, mas às vezes também limitam as possibilidades de ajuste do desempenho.

Da mesma forma, as ricas garantias de processamento e tolerância a falhas para operações com estado aumentam a confiabilidade e tornam mais fácil raciocinar sobre a semântica, mas exigem que o sistema faça trabalhos adicionais, ou seja reconhecimentos e replicação de estado (síncrona).

Uma semântica exactly-once é particularmente desejável e pode ser implementada através da combinação das garantias do processamento at-least-once com atualizações de estado transacionais ou idempotentes, mas elas não podem ser alcançadas para ações com efeitos colaterais, como enviar uma notificação a um administrador.

4. Outros sistemas:

Nos últimos dois anos, surgiu um grande número de processadores de Stream que visam fornecer alta disponibilidade, tolerância a falhas e escalabilidade horizontal. Flink [2] (anteriormente conhecida como Estratosfera [15]) é um projeto que tem muitos paralelos com o Spark Streaming, pois também se originou da pesquisa e anuncia a unificação de processamento em lote e stream no mesmo sistema, fornecendo garantia exactly-once para a programação de um modelo Stream e uma API de alto nível comparável à do Trident.

Em contraste com Spark Streaming, no entanto, Flink não depende de processamento em lotes para processamento de Stream internamente e, portanto, oferece latência baixa na ordem de Storm ou Samza.

O Projeto Apex [1] é um software livre / aberto com motor DataTorrent RTS. Muito parecido com Flink, Apex promete alto desempenho em Stream e processamento em lote com baixa latência.

Como Spark / Spark Streaming, Flink e Apex são complementados por uma série de banco de dados, sistema de arquivos e outros conectores, bem como correspondência de padrões, aprendizagem de máquinas e outros algoritmos

através de bibliotecas adicionais. Um sistema que não é publicamente disponível é a Heron do Twitter [27].

Desenhando para substituir o Twitter, o Heron é completamente compatível com API para Storm, mas melhora em vários aspectos, tais como backpressure, eficiência, isolamento de recursos, facilidade de depuração e monitoramento de desempenho, mas, segundo se informa, não fornece garantias exacly-once.

MillWheel [13] é um processador de Stream extremamente escalável que oferece qualidades semelhantes como Flink e Apex, ou seja, gestão de estado e semântica exaclty-once. Millwheel e FlumeJava [19] são os motores de execução por trás do serviço de nuvem Google Dataflow para processamento de dados.

Como outros serviços do Google e ao contrário da maioria dos outros sistemas discutidos nesta seção, o Dataflow é totalmente gerenciado e, assim, alivia seus usuários da carga de implantação e de todos os problemas relacionados. O modelo de programação Dataflow [14] combina o processamento de lote e de Stream e é também agnóstico do sistema de processamento subjacente, desacoplando assim a lógica de negócios da implementação real.

A API tornou-se aberto (software livre) em 2015 e evoluiu para o projeto Apache Beam (abreviatura de Batch e Stream, atualmente em incubação) [7] para agrupá-lo com os correspondentes mecanismos de execução (runners): Até a escrita deste artigo, Flink, Spark e o serviço de nuvem proprietário do Google Dataflow são suportados.

O único outro sistema totalmente gerenciado de stream além do Google Dataflow que estamos cientes é IBM Infosphere Streams [17]. No entanto, em contraste com o Google Dataflow que é documentado para ser altamente escalável (limite de cota para os clientes: 1000 nós de computação [9]), é difícil encontrar evidência de alta escalabilidade de IBM Infosphere Streams; As avaliações de desempenho feitas pela IBM [21] apenas indicam que ele funciona bem em pequenas implantações com até alguns nós.

O Apache Flume é um sistema para agregação e coleta de dados eficientes que é freqüentemente usado para ingerir dados no Hadoop, pois se integra bem com HDFS e pode lidar com grandes volumes de dados de entrada. Mas Flume também suporta operações simples como filtragem ou modificação em dados de entrada através de Interceptores Flume [23, Capítulo 7], que podem ser encadeados juntos para formar um pipeline de processamento de baixa latência.

A lista de processadores de fluxo distribuídos continua [16,28, 33, 38], mas uma vez que uma discussão completa de processamento de Big Data Streaming está fora do escopo deste artigo, não entraremos em mais detalhes aqui.

5. Conclusão

Os sistemas orientados por lotes/batch fizeram o trabalho pesado em aplicações com uso intensivo de dados por décadas, mas não refletem a natureza ilimitada e contínua dos dados, como produzidos em muitas aplicações do mundo real.

Os sistemas de Streaming, por outro lado, processam os dados à medida que chegam e, portanto, são, muitas vezes, mais naturais, embora inferiores em relação à eficiência.

Enquanto um número crescente de implantações utilizando a Arquitetura Lambda e sistemas híbridos emergentes como Dataflow / Beam, Flink ou Apex documentam esforços significativos para fechar a lacuna entre o processamento de lote/batch e de streaming, tanto na pesquisa quanto na prática, a arquitetura Kappa evita completamente a abordagem tradicional e até anuncia o advento analítico de Big Data, puramente orientada para streaming.

No entanto, seja no núcleo de novos designs de sistemas ou como complemento das arquiteturas existentes, os processadores de Streaming escalonáveis horizontalmente estão ganhando impulso, já que a exigência de baixa latência se tornou uma força motriz nos pipelines de análise Big Data modernos.

6. Referências:

1. Apache apex. http://apex.incubator.apache.org/.

2. Apache flink. https://flink.apache.org/.

3. Apache Flume. https://flume.apache.org/.

4. Apache Kafka. http://kafka.apache.org/.

5. Apache Mesos. http://mesos.apache.org/.

6. Apache Yarn. http://hadoop.apache.org/docs/stable/ hadoop-yarn/hadoop-yarn-site/YARN.html.

7. Beamproposal – incubator wiki. https://wiki.apache.org/incubator/BeamProposal.

8. Comparisons: Spark streaming. http://samza.apache.org/ learn/documentation/0.7.0/comparisons/spark-streaming.html. Accessed: 2016-01-12.

9. Resource quotas. https://cloud.google.com/dataflow/quotas. Accessed: 2016-01-14.

10. Spark streaming programming guide. https://spark.apache. org/docs/1.6.0/streaming-programming-guide.html. Accessed: 2016-01-12.

11. D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In In CIDR, pages 277–289, 2005.

12. D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. The VLDB Journal, 12(2):120–139, Aug. 2003.

13. T. Akidau, A. Balikov, K. Bekiroglu, et al. Millwheel: Faulttolerant stream processing at internet scale. In Very Large

Data Bases, pages 734–746, 2013.
14. T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massivescale, unbounded, out-of-order data processing. Proceedings of the VLDB Endowment, 8:1792–1803, 2015.
15. A. Alexandrov, R. Bergmann, S. Ewen, et al. The stratosphere platform for big data analytics. The VLDB Journal, 2014.
16. R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, et al. Photon: Fault-tolerant and scalable joining of continuous data streams. In SIGMOD '13, 2013.
17. A. Biem, E. Bouillet, H. Feng, et al. Ibm infosphere streams for scalable, real-time, intelligent transportation services. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 2010.
18. O. Boykin, S. Ritchie, I. O'Connell, et al. Summingbird: A framework for integrating batch and online mapreduce computations. PVLDB, 2014.
19. C. Chambers, A. Raniwala, F. Perry, et al. Flumejava: Easy, efficient data-parallel pipelines. In PLDI 2010, 2010.
20. S. Chintapalli, D. Dagit, B. Evans, et al. Benchmarking streaming computation engines at yahoo! http://yahooeng.tumblr. com/post/135321837876/benchmarking-streamingcomputation-engines-at, December 2015. Accessed: 2016-01-11.
21. I. Corporation. Of streams and storms. Technical report, IBM Software Group, 2014.

22. Ericsson. Trident – benchmarking performance.
http://www.ericsson.com/research-blog/data-knowledge/
trident-benchmarking-performance/. Accessed: 2016-01-12.
23. M. Grover, T. Malaska, J. Seidman, and G. Shapira. Hadoop
Application Architectures. O'Reilly, Beijing, 2015.
24. J. Kreps. Benchmarking apache kafka: 2 million writes per
second (on three cheap machines). https://engineering.
linkedin.com/kafka/benchmarking-apache-kafka-2-millionwrites-
second-three-cheap-machines. Accessed: 2016-01-12.
25. J. Kreps. Questioning the lambda architecture.
http://radar.oreilly.com/2014/07/questioning-the-lambdaarchitecture.
html, 7 2014. Accessed: 2015-12-17.
26. J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging
system for log processing. In NetDB'11, 2011.
27. S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal,
J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron:
Stream processing at scale. In Proceedings of the 2015 ACM
SIGMOD International Conference on Management of Data,
SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.
28. W. Lam, L. Liu, S. Prasad, et al. Muppet: Mapreduce-style processing
of fast data. VLDB 2012, 2012.
29. D. Laney. 3D data management: Controlling data volume, velocity,
and variety. Technical report, META Group, February 2001.
30. N. Marz. Preview of storm: The hadoop of realtime processing.
http://web.archive.org/web/20120509023348/http://tech.
backtype.com/preview-of-storm-the-hadoop-of-realtime-proce,
5 2012. Accessed: 2015-12-17.
31. N. Marz. History of apache storm and lessons learned.

http://nathanmarz.com/blog/history-of-apache-storm-andlessons-learned.html, 10 2014. Accessed: 2015-12-17.

32. N. Marz and J. Warren. Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.

33. L. Neumeyer, B. Robbins, and A. Kesari. S4: Distributed stream computing platform. In In Intl. Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms (KDCloud, 2010.

34. D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, 2010.

35. K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.