Processamento aproximado de grafos em grande escala

Renato David Silva Rosa

Instituto Superior Técnico
Lisboa, Portugal

Resumo Os grafos modelam muitas aplicações reais e apresentam desafios específicos em termos de processamento em grande escala. Nesse contexto, propõe-se o desenvolvimento duma extensão a um sistema de processamento de grafos que, por meio de resultados aproximados e de computação diferida, permita otimizar o uso dos recursos computacionais e dos tempos de resposta, e assim aumentar a escalabilidade do sistema. Os requisitos de qualidade da aproximação dos resultados serão definidos pelo utilizador, de acordo com um modelo de Quality-of-Data.
Desta forma, as atualizações ao grafo, sob a forma de *stream*, são monitorizadas, e a computação pode ser executada apenas quando é esperada a produção de novos resultados relevantes.
A fim de contextualizar e delimitar a solução proposta, examina-se o estado da arte em termos de sistemas de processamento em bloco e de *streams*, processamento especializado de grafos e soluções para otimização e escalabilidade dos sistemas, incluindo o processamento aproximado.
Por fim, é apresentado um esboço de arquitetura da solução projetada e são indicados os principais elementos para a sua avaliação.

Palavras-chave: processamento em bloco, *streams*, grafos, processamento aproximado, Quality-of-Data, Flink

1 Introdução

Muitos objetos e realidades do nosso quotidiano, quer naturais quer criados pelo homem, assumem a forma de redes, com elementos individuais que se relacionam entre si. Como veremos, essas redes, representadas na forma de grafos, apresentam particulares desafios computacionais. Neste trabalho, motiva-se e formula-se o problema do seu processamento em larga escala e, analisado o contexto em termos de trabalho já realizado nessa área, ensaia-se uma proposta de solução.

1.1 Desafios e dificuldades

As redes reais comportam importantes desafios em termos de representação, armazenamento e processamento. As suas características, bem como a sua importância, motivaram o aparecimento de grande número de técnicas, algoritmos e sistemas especialmente desenhados tendo em vista a eficiência e a rapidez de resposta.

Dimensão. A teoria computacional de grafos contempla diferentes formas de representação de redes, com diferentes implicações em termos do espaço ocupado e de eficiência das operações comuns num grafo. Para além das representações clássicas, matrizes de adjacências e listas de adjacências, existem ainda representações híbridas e com recurso a várias otimizações. Existem porém casos de grafos, como é o caso típico da World Wide Web (WWW), cuja dimensão é tal que requer técnicas avançadas de representação [15, 16].

A importância dos grafos e a necessidade de os armazenar e processar de forma rápida e eficiente veio dar origem a bases de dados especializadas, a sistemas distribuídos dedicados ao seu processamento em larga escala e em tempo real, e a *frameworks* bibliotecas para tratamento de grafos em sistemas de processamento distribuído mais genéricos, como veremos na Sec. 3.3.

Heterogeneidade. À grande dimensão das redes que encontramos na natureza e na sociedade acresce uma característica que lança importantes desafios: a sua heterogeneidade, em termos da distribuição do grau dos vértices. Em muitos grafos reais, como por exemplo, mais uma vez, a *WWW*, encontramos um grande número de vértices com grau reduzido mas também um número reduzido de vértices, conhecidos por *hubs*, com um número muito elevado de ligações. A distribuição de grau segue uma *lei de potência* nestes grafos, que são frequentemente *livres de escala* [10, Sec. 4.2-4.4] [11].

Esta heterogeneidade potencia consideráveis problemas para um sistema distribuído de processamento de grafos: seguindo uma estratégia simples de partição de vértices pelo nodos do sistema, a distribuição da carga de trabalho pode facilmente apresentar grandes desequilíbrios, pela presença de *hubs*, aos quais se associa, tipicamente, uma necessidade de processamento muito superior.

Mutabilidade. Um terceiro desafio prende-se com o facto de que a maior parte das redes evolui ao longo do tempo, quer na sua estrutura, quer nas propriedades que estão associadas aos seus elementos. Um objeto em constante mudança requer cuidados específicos. O problema da volatilidade dos grafos reais é uma das motivações para a contribuição que esperamos fornecer com este trabalho.

1.2 Grafos

Os grafos e o seu estudo estão presentes em muitos campos do conhecimento: não só na matemática, nomeadamente na teoria de grafos, que os estuda exaustivamente, mas também nas ciências sociais e naturais, como a biologia e a física. É na teoria matemática de grafos que encontramos porém a formalização do seu estudo, com importantes resultados teóricos e aplicações práticas. Por outro lado, a necessidade de resolver problemas que envolvem grafos em cada domínio levou a que as outras ciências contribuíssem para enriquecer o

domínio teórico com intuições, conceitos inovadores e importantes aplicações [12, pp. 1-2].

Existem abundantes aplicações dos grafos e diversas funções e medidas que podem ser computadas com base neles: caminhos entre vértices, distâncias, componentes, cortes, medidas de centralidade, identificação de comunidades (*clusters*), entre muitas outras.

1.3 *Streams* de grafos

A questão da mutabilidade das redes introduz a utilização de *streams* no contexto do processamento de grafos. A melhor forma de representar a evolução do grafo é através duma *stream* de atualizações que vão alterando propriedades do mesmo. As alterações tanto podem manter intacta a estrutura da rede, modificando apenas os dados associados a vértices e arcos, como proceder a alterações topológicas ao grafo, introduzindo e removendo vértices e arcos. Estes dois tipos de atualizações têm implicações diferentes em termos de processamento distribuído, representação, equilíbrio de carga e otimização. Ao invés de se construir o grafo estaticamente para depois o analisar, recebe-se uma *stream* de atualizações incrementais. Redes sociais bem conhecidas, tais como Facebook, Twitter, blogs, apresentam estas características, constituindo bons casos de uso de processamento de *streams* de grafos em tempo real [20].

Existe um outro tipo de *stream* de grafos, que se aplica quando o mesmo é demasiado grande para ser representado em memória. Neste caso, o que o sistema/algoritmo recebe é uma *stream* dos arcos do grafo, numa qualquer ordem arbitrária, tendo o processamento de ser realizado em apenas uma passagem da *stream*. Existem variantes, como múltiplas passagens, "anotação" da *stream*, ou a representação de parte do grafo no sistema [42].

2 Objetivos

2.1 Motivação e caso de uso

Como vimos, o processamento de grafos envolve estruturas de grande dimensão e, mais recentemente, também em constante mutação. Os sistemas distribuídos permitem, também neste contexto, efetuar processamento em larga escala, com resultados em tempo útil. No entanto, em qualquer um destes sistemas, as questões relacionadas com a otimização de recursos e de tempo de resposta são uma preocupação constante, e o contexto dos grafos não é exceção. É a questão da otimização de recursos e da eficiência no processamento de grafos em larga escala e com alterações contínuas que motiva a presente proposta de trabalho.

Vamos considerar, para efeitos de definição do nosso objeto de estudo, sistemas distribuídos de processamento de grafos com suporte a atualização dos mesmos. Consideraremos como caso típico aquele em que o sistema possui uma representação do grafo e recebe continuamente atualizações, que aplica ao

grafo. Consideramos que existe simultaneamente uma função de interesse, que deve ser computada a partir do grafo, e que as alterações ao grafo implicam a sua recomputação, seja ela realizada de raiz ou incrementalmente, sobre o resultado anterior.

2.2 Inspiração

A presente proposta de trabalho, na linha do tema apresentada a concurso, vai buscar a sua inspiração a trabalho que vem sendo realizado no campo da otimização de recursos/tempo de resposta em *workflows* de processamento [21, 22, 23].

Em *workflows* de processamento em grande escala, bem como nos componentes individuais que os constituem, existe consumo de CPU, memória, I/O, e também de tempo necessário para computar uma resposta atualizada. Ora, sucede que algumas vezes existe pouca diferença entre o resultado da recomputação e o resultado obtido anteriormente, não obstante todo o gasto decorrente do processamento. Ora, em certos domínios é tolerável admitir uma resposta não exata, mas suficientemente aproximada, se isso significar uma poupança significativa em termos de recursos computacionais e rapidez. Neste caso, o sistema poderia limitar-se a devolver a resposta anterior, adiando a recomputação para quando esta produzir resultados significativos [23]. Estamos perante uma necessidade de compromisso entre a correção da resposta dada pelo sistema e a otimização do uso dos recursos utilizados para obter a mesma resposta. Esse compromisso pode ser formalizado e expresso através do conceito de Quality-of-Data (QoD), que será explicitado na Sec. 3.4.

2.3 Problema a resolver

O objetivo do nosso trabalho é pois fornecer um meio de otimizar o uso de recursos computacionais e também o tempo de resposta em sistemas distribuídos de grafos. Partindo dum conceito já aplicado a *workflows* de processamento, que consiste na avaliação do impacto de novos dados e na recomputação adiada, segundo um modelo de QoD definido pelo utilizador, propõe-se, no presente trabalho, aplicar este mesmo conceito a um nível inferior e num âmbito mais específico, o do processamento de grafos.

Como vimos, os grafos, além de relevantes na representação de problemas, apresentam especificidades que os distinguem doutras classes de problemas, tanto que motivaram a criação de sistemas especializados no seu processamento. Assim, a aplicação deste modelo a grafos comporta desafios que justificam uma abordagem dedicada, e daí a presente proposta de trabalho.

2.4 Dificuldades e desafios

Este objetivo, como é evidente, não pode ser alcançado sem responder a algumas dificuldades que de imediato se apresentam.

O sistema terá de estar apto a prever quando a recomputação não vai produzir alterações significativas no resultado, e caso contrário, decidir efetuar o processamento. Isto implica a existência dum critério para decidir o que são ou não alterações significativas. Pode definir-se por exemplo, uma diferença máxima relativa entre os dados de entrada anteriores e os mais recentes ou, em alternativa, uma diferença máxima entre o estado atual e uma previsão dos novos resultados, a partir da qual a computação tem de ser repetida.

Para esse efeito, há que encontrar um modo de estimar a diferença relativa entre o resultado anterior e o resultado que se obteria se a recomputação fosse realizada ou, por outras palavras, estimar o erro acumulado com confiança suficiente. Para isso, porém, é necessário poder efetuar uma previsão da diferença relativa conhecendo apenas o *input*, ou seja, a *stream* de alterações ao grafo desde a última computação.

O utilizador do sistema tem de poder exprimir o que considera um desvio aceitável em relação aos valores exatos, e estabelecer os seus objetivos em termos do compromisso entre correção e otimização, tendo por base que são como inversas uma da outra: maior otimização do uso de recursos implica maior erro e incerteza, e exatidão dos resultados exige maior processamento.

3 Trabalho relacionado

Nesta secção apresenta-se o trabalho relacionado com o objetivo a que nos propomos, numa perspetiva crítica, apontando as contribuições valiosas para o nosso trabalho, mas também em que medida essas contribuições não resolvem cabalmente o problema.

Começaremos por apresentar brevemente os mais importantes sistemas distribuídos para computação em grande escala, em bloco, focando de seguida o processamento especializado de *streams*. Versa-se de seguida o universo dos sistemas especializados em grafos, com referências aos mais importantes paradigmas de computação/programação por eles introduzidos. Por fim, explora-se a temática da escalabilidade e das técnicas existentes para, no contexto da computação distribuída, otimizar o uso dos recursos à custa dum erro aceitável no *output*. Conclui-se introduzindo o conceito de Quality-of-Data.

3.1 Processamento em bloco

A necessidade processar grandes quantidades de dados de forma rápida, fiável e eficiente deu origem a estratégias novas para abordar os problemas. A paralelização surgiu como o grande paradigma para enfrentar as tarefas que os problemas envolvem.

O modelo/sistema MapReduce foi pioneiro na difusão deste paradigma, e constitui ainda um ponto de referência. Os diferentes sistemas nele inspirados distinguem-se sobretudo pela forma como suportam computação iterativa e com dependências, e como potenciam a reutilização dos resultados intermédios.

Enquanto o MapReduce e o Hadoop olham o processamento como uma série de computações independentes, sistemas como o Spark ou o Flink[1] dão ênfase à computação iterativa e ao *caching* agressivo de resultados intermédios como meio de rentabilizar a computação.

MapReduce. O MapReduce [19] é um modelo de programação destinado ao processamento e geração de grandes conjuntos de dados. O nome também designa a implementação deste modelo, criada pela Google. Dado um conjunto de dados sob a forma de pares chave-valor, o utilizador define duas funções: a primeira, *map*, processa cada par chave-valor original, gerando um conjunto de pares chave-valor intermediários; a segunda, *reduce*, funde, ou sumariza, todos os valores intermediários associados à mesma chave.

Este modelo pode exprimir uma grande quantidade de tarefas reais de processamento de dados, de forma simplificada e especialmente apta à paralelização, por não estabelecer dependências entre os dados. O utilizador pode concentrar-se apenas na lógica específica do problema que pretende solucionar, abstraindo dos pormenores relativos ao paralelismo, distribuição, sincronização, tolerância a falhas e comunicação, que são assegurados pelo sistema.

Hadoop. O Hadoop[2] é uma *framework open-source* que permite o processamento distribuído de grandes conjuntos de dados em *clusters*, inspirado no MapReduce da Google. Apresentando um modelo de programação simples, permite estender a computação a milhares de nodos, garantindo tolerância a falhas e distribuição de carga, implementadas na camada de aplicação [55].

Do Hadoop fazem parte o Hadoop Distributed File System (HDFS), sistema de ficheiros distribuído onde os dados podem ser acedidos por todos os nodos do sistema, o YARN, para agendamento de tarefas e gestão de recursos do *cluster*, e o sistema MapReduce propriamente dito.

A importância do Hadoop torna-se evidente ao perceber que atua como uma base comum a um conjunto de projetos significativos no âmbito da computação distribuída[3].

Para o problema que temos em mãos, como veremos, o modelo de processamento MapReduce simples não é o mais indicado para o tratamento de grafos. Acresce ainda que, para os objetivos de otimização de recursos, por meio de admissão dum erro máximo, o Hadoop opera num nível demasiado básico para as necessidades.

[1] O Flink, sendo mais corretamente definido como um sistema de processamento de *streams* que também processa dados em bloco será abordado na parte correspondente, Sec. 3.2.

[2] https://hadoop.apache.org

[3] Cassandra e HBase (bases de dados NoSQL), Zookeeper (serviço de coordenação), Pig (linguagem e *framework* para análise de dados), e o Spark (abordado de seguida) são exemplos de tecnologias pertencentes ao "universo" Hadoop. Outros projetos, como por exemplo o Flink (Sec. 3.2) não estão diretamente ligados ao Hadoop, mas disponibilizam *drivers* para utilizar o HDFS como fonte dos dados a processar.

Spark. O Spark[4] surgiu como forma de dar solução a problemas para os quais a abordagem MapReduce é ineficaz: os que envolvem a computação repetida do mesmo conjunto de dados, e a análise interativa, *ad-hoc*, de *datasets*. Neste tipo de problemas, o Hadoop volta a recarregar todos os dados a partir do HDFS, incorrendo numa latência desnecessária[61]. No caso dos algoritmos em grafos, que são frequentemente de natureza iterativa, esta questão coloca-se de forma premente.

O Spark resolve este problema através duma abstração denominada Resilient Distributed Datasets (RDDs), que são estruturas de dados paralelas, tolerantes a falhas, que permitem manter explicitamente resultados intermédios em memória, a fim de serem reutilizados e manipulados por uma série de operadores. A eficiência dos RDDs provém do facto de apenas poderem ser obtidos por operações determinísticas a partir dos dados ou de outros RDDs. Desta forma, ao invés de se representarem os dados explicitamente, são armazenadas as operações que permitem obter o RDD a partir dos dados anteriores, permitindo evitar a replicação direta que é dispendiosa e mais difícil de manter, ao mesmo tempo que se assegura uma eficaz recuperação de falhas [60].

Os RDDs, juntamente com as operações suportadas sobre eles e uma forma de variáveis partilhadas entre as várias tarefas do programa, permitem que o Spark, com o seu modelo predominantemente em memória, possa correr programas 100 vezes mais rapidamente que o Hadoop, em memória, e 10 vezes mais rapidamente em disco[7].

Como veremos adiante, o Spark possui o seu próprio módulo de processamento de *streams* (Sec. 3.2) e de grafos (Sec. 3.3).

3.2 Processamento de *streams*

O MapReduce clássico é apropriado para processamento de grandes blocos de dados estáticos, e não tanto de fluxos de dados contínuos, vulgo *streams* de dados. Isso ocorre porque, como vimos (Sec. 3.1), o modelo MapReduce divide a computação em fases individuais e independentes, não estando otimizado para computação iterativa ou contínua.

Como forma de mitigar esta desvantagem, foram estudados modelos incrementais de MapReduce [35], dos quais um dos pioneiros foi o Google Percolator [43]. O processamento incremental consiste em computar resultados atualizados a partir de resultados obtidos anteriormente e dos novos dados entretanto recebidos, sem computar a função novamente no conjunto de dados inteiro, o que é frequentemente impraticável ou muito dispendioso. Se para certas funções uma versão incremental é trivial, para outras é muito difícil encontrar uma boa versão incremental, mesmo quando existe um algoritmo eficiente para dados estáticos [14].

Note-se porém que processamento incremental não é sinónimo de processamento de *streams*. Por um lado, pode haver processamento incremental sem

[4] http://spark.apache.org

streams, quando os dados mais recentes chegam em blocos, e não em contínuo; por outro, uma *stream* não tem de ser processada incrementalmente: pode recolher-se uma porção razoável da *stream* e proceder depois a uma recomputação de raiz. Contudo, quer num caso quer noutro existe uma considerável latência, associada a períodos de tempo em que os resultados apresentados pelo sistema não correspondem aos dados mais atuais.

Por este motivo, foram desenvolvidos sistemas que procuram aproximar-se o mais possível da computação em tempo real, fornecendo resultados com a menor latência possível em relação à chegada dos dados. Existem duas grandes abordagens: processamento da *stream* sob a forma de *mini-batches*, e processamento direto de *streams*. Outro elemento diferenciador é a tolerância a falhas oferecida. Apresentamos algumas implementações de maior relevância.

Storm. O Storm[5] foi originalmente criado pelo Twitter e é um sistema distribuído de processamento em tempo real de *streams* ilimitadas de dados. É indicado para casos em que é requerida uma latência muito baixa nos resultados.

A base duma aplicação em Storm é chamada *topology*, que se distingue duma aplicação MapReduce clássica pelo facto de, em princípio, correr por tempo indefinido. Uma *topology* consiste num grafo com vértices de dois tipos: *spouts*, fontes de *streams*, usualmente provenientes duma fonte externa; *bolts*, onde é realizada toda a computação, filtros, agregações, transformações, acesso a outras fontes externas, etc., e que podem por fim emitir uma ou mais novas *streams*. Operações complexas podem ser assim implementadas através de vários *bolts*, com um fluxo de dados definido pela topologia.

Cada *bolt* executa várias *tasks* distribuídas pelo *cluster*. Uma diferença essencial em relação ao MapReduce é que existe controlo explícito por parte do utilizador sobre a forma como as *streams* de entrada são distribuídas pelas várias *tasks* existentes [17].

Alguns *benchmarks* realizados [13, 29] comprovam que o Storm apresenta uma latência muito baixa quando comparado com outros sistemas (Spark e Flink), mas que em contrapartida possui taxa de débito (*throughput*) menos vantajoso. Outra possível desvantagem prende-se com as garantias em termos de tolerância a falhas: o Storm garante o processamento de cada elemento *at-least-once*, enquanto sistemas como o Spark Streaming e o Flink conseguem garantir *exactly-once* [59].

Para o nosso caso concreto, o Storm revela indiscutíveis capacidades em termos de processamento de *streams*, mas não possui suporte especializado a grafos, ao contrário dos exemplos que veremos de seguida.

Spark Streaming. O sistema Spark disponibiliza o Spark Streaming[6], uma biblioteca para processamento de *streams* sobre o Spark. A técnica base uti-

[5] http://storm.apache.org
[6] http://spark.apache.org/streaming

lizada são as *Discretized Streams*, que consistem em dividir a computação da *stream* numa série de *mini-batches*, pequenos blocos, como o nome indica, que são computados de forma determinística e *stateless*, em pequenos intervalos de tempo, tipicamente inferiores a 1 segundo. Cada um dos blocos em que a *stream* é dividida é transformado num RDD, o que traz vantagens em termos de tolerância a falhas (da forma descrita na secção 3.1), e também de resposta a *stragglers*[7], ao contrário doutros sistemas de *streams*, que têm dificuldade em lidar com este tipo de problemas [59].

Desta forma, são suportadas todas as operações padrão em *streams*, ao mesmo tempo que as questões relativas à consistência são clarificadas pelo facto de se utilizar uma noção discretizada de tempo. O uso de RDDs apresenta ainda a vantagem de permitir a integração com outros tipos de tarefas do Spark, nomeadamente o processamento em blocos tradicional ou consultas interativas [59].

A principal limitação desta abordagem consiste na latência que surge inevitavelmente por se efetuar *batch* dos dados, ainda que em intervalos pequenos [59]. Apesar de boa parte das aplicações suportarem essa latência sem problemas, existem outras para as quais tal demora não é desejável.

Flink. O Flink[8] é uma plataforma *open-source* para processamento distribuído de *streams* e dados em bloco. Tem a sua origem no projeto Stratosphere [3], criado na Universidade de Berlim. O seu núcleo é um sistema de processamento de *streams*, sendo o processamento em bloco tratado internamente como o caso especial duma *stream* limitada. Neste sentido, o Flink segue uma abordagem oposta ao Spark, e mais próxima do Storm. Esta estratégia permite também unificar o tratamento de blocos e *streams* na mesma *framework*.

Entre as funcionalidades suportadas, contam-se o processamento de eventos fora de ordem, janelas de *streaming* flexíveis, processamento contínuo e computações *stateful*. Quanto a tolerância a falhas, utiliza *snapshots* distribuídos e afirma garantir processamento *exactly-once*. No que respeita a otimizações, é de notar o suporte a computação iterativa e baseada em *deltas* (útil no caso dos grafos) e ainda uma gestão de memória própria dentro da Java Virtual Machine (JVM) [6].

O Flink afirma apresentar um alto desempenho em termos de taxa de débito, aliado a uma baixa latência, o que é confirmado por alguns *benchmarks* realizados recentemente [29].

Relativamente ao Spark, o sistema que mais se lhe assemelha em termos de objetivos, a sua principal desvantagem é a menor maturidade e popularidade (em termos de colaboradores e de utilizadores).

[7] Nodos que não falham, mas são lentos, atrasando a computação como um todo.
[8] http://flink.apache.org

Arquitetura Lambda. Terminamos esta secção com um contributo que dalgum modo se propõe reunir os benefícios de computação em bloco e computação de *streams*.

Existem três grandes propriedades que se deseja que um sistema analítico para grandes quantidades de dados possua: *correção*; baixa *latência*; elevado débito. Ora, as técnicas e sistemas que vimos até agora não logram satisfazer simultaneamente estas propriedades. Sistemas de processamento em bloco, como o Hadoop, conseguem correção máxima e bom débito, mas com latências elevadas; sistemas de processamento de *streams*, como o Storm, apresentam latências mínimas, mas falham no débito, na presença de grandes quantidades de dados, a não ser que recorram a processamento aproximado, sacrificando a correção.

A Arquitetura Lambda [41] é uma proposta de arquitetura de alto nível para responder a esta questão. A sua estratégia baseia-se na conceção dum sistema em camadas (*layers*), em que os requisitos em termos de correção, latência e débito são separados entre si, e satisfeitos por diferentes camadas do sistema [41, pp. 14-15].

O *batch layer* [41, p. 16] armazena um grande conjunto de dados em bruto, imutáveis e em constante crescimento, e computa funções arbitrárias sobre o mesmo. Os resultados são recomputados regularmente, à medida que chegam novos dados, e são produzidas *batch views*, conjuntos de resultados pré-calculados, que são utilizados pelo *serving layer* para responder em tempo mínimo a consultas sobre os dados [41, p. 17]. Para o *batch layer* utiliza-se geralmente o Hadoop, mas o Spark ou o Flink são também escolhas possíveis. Para o *serving layer* utilizam-se bases de dados otimizadas para leitura.

A recomputação a partir dos dados é usualmente demorada (de alguns minutos a várias horas), pelo que as consultas, nesse intermédio, apresentarão resultados desatualizados. Para compensar esta falha, é introduzida uma terceira camada, o *speed layer*, que tem como função processar apenas os dados mais recentes que chegam continuamente, de forma incremental, gerando *realtime views* em constante atualização, e possivelmente apenas aproximadas, que são descartadas assim que estiverem disponíveis resultados exatos provenientes do *batch layer* [41, pp. 18-20]. Para o *speed layer* utilizam-se sistemas de processamento de *streams*, como o Storm. O Spark Streaming e o Flink podem ser utilizados também para esta camada.

A Arquitetura Lambda não é um sistema em si, mas uma visão de alto nível dum sistema analítico completo. Para o presente trabalho, esta visão contribui para contextualizar o problema que nos propomos resolver. Na medida em que se pretende reduzir latência e otimizar recursos, por meio de respostas aproximadas e incrementais, o presente trabalho terá o seu lugar no *speed layer* desta arquitetura.

3.3 Processamento de grafos

Os sistemas que vimos até agora, para dados em bloco ou *streams* de dados, não são especializados no processamento de grafos, ainda que possam ser utilizados para esse fim. O modelo *MapReduce*, como referido, falha na representação de dependências computacionais próprias de certos algoritmos. Uma abordagem iterativa, por sua vez, além de não ser naturalmente exprimível em termos de MapReduce, não aproveita o seu potencial de processamento paralelo [38]. Por outro lado, tal abordagem acaba frequentemente por envolver demasiado movimento de dados, sem aproveitar as oportunidades de otimização decorrentes da peculiar estrutura dos dados em forma de grafo [26].

As bases de dados especializadas em grafos[9] permitem armazenar grafos duma forma especializada. São indicadas quando é importante uma lógica transacional e com foco nas operações CRUD, e quando as consultas se centram principalmente na própria estrutura do grafo. Não oferecem normalmente, porém, capacidades de processamento distribuído nem otimização para processamento de dados em *batch* ou em *stream*. Quando muito, podem funcionar como representação externa dos grafos (e dessa forma poderiam funcionar, por exemplo, em conjunto com um sistema de processamento como o Storm).

Praticamente todos os sistemas de processamento de grafos se baseiam na expressão dos algoritmos como um conjunto de pequenas funções a serem executadas de forma paralela e iterativa. A diferenciação entre os vários sistemas pode ser feita a partir de dois eixos principais: o elemento central da iteração (vértices ou arcos) e o modelo de comunicação (mensagens ou memória partilhada). Estas diferenças dão origem a outras características diferenciadoras, como por exemplo o suporte a esquemas assíncronos de processamento.

Computação centrada nos vértices. A distribuição de carga e paralelização, neste modelo, é efetuada através da partição dos vértices do grafo pelos vários nodos do sistema. Cada nodo recebe um igual número de vértices, que são computados em paralelo, assegurando-se a sincronização, comunicação e tolerância a falhas. A fim de otimizar o fluxo de dados entre diferentes nodos, a partição de vértices deve ser tal que minimize o número de arcos divididos entre diferentes máquinas, no que é conhecido como *edge-cut*.

Existem algumas ferramentas para efetuar a partição do grafo descrita mas, como observado por [27], o seu desempenho não é satisfatório para grafos cuja distribuição de grau segue uma lei de potência. A partição por meio de *hashing* simples, sendo rápida e fácil de implementar, tem o inconveniente de, em termos de valor esperado, cortar a maior parte dos arcos. Se a carga em termos de computação fica razoavelmente bem distribuída, o *overhead* em termos de comunicação aproxima-se do pior cenário possível [56].

[9] Dois exemplos líderes de mercado são o Neo4j (http://neo4j.com) e o Titan (http://thinkaurelius.github.io/titan)

Em modelos que tratam todos os vértices de forma simétrica, a distribuição de grau em forma de lei de potência introduz desequilíbrio na distribuição de trabalho, visto que a computação apresenta uma complexidade que é normalmente linear no grau do vértice [27].

Pregel. Apesar de já existirem sistemas e *frameworks* para processamento de grafos, as mesmas não tinham em conta as questões da escalabilidade e de tolerância a falhas, respondidas na maior parte dos sistemas distribuídos. Assim, com a finalidade de permitir o processamento distribuído de grafos em larga escala, foi criado e implementado pela Google, um sistema denominado Pregel [39].

O processo de computação de grafos utilizado pelo Pregel desenvolve-se numa sequência de passos, denominados *supersteps*. Em cada um desses passos, é executada, para cada vértice, uma função (*vertex-program*), definida pelo utilizador. Essa função pode modificar o estado do vértice e dos arcos nele originados. Para além disso, em cada passo, a função tem acesso às mensagens enviadas a cada vértice no passo anterior, e tem a possibilidade de enviar mensagens destinadas a outros vértices (normalmente os vértices adjacentes, mas não necessariamente), que serão processadas no passo seguinte.

Esta forma de conceber a computação centrada no vértice dá origem à expressão *think-like-a-vertex*, que caracteriza este modelo de computação, mais apta conceptualmente a ser distribuída, por ser dividida em pequenas unidades.

No fim de cada passo, cada vértice pode desativar-se, *votando para parar*. Um vértice inativo não é computado nos passos seguintes, a não ser que receba uma mensagem. O algoritmo, no seu conjunto, termina quando todos os vértices votarem para parar e não existirem mensagens pendentes.

Este modelo é, dalgum modo, uma adaptação do *MapReduce*, na medida em que se foca na computação local a cada vértice, tomando o sistema a responsabilidade de integrar e unificar tudo da forma adequada. Por outro lado, este modelo centrado no vértice e baseado em mensagens presta-se a exprimir boa parte dos algoritmos de forma intuitiva e acessível.

O modelo de computação do Pregel foi implementado de forma *open-souce* através do projeto Apache Giraph[10], um dos sistemas de processamento de grafos com maior popularidade.

GraphLab. Um outro sistema, o GraphLab [37], distingue-se do Pregel pelo facto de não utilizar mensagens, mas assentar a comunicação num modelo de memória partilhada. Cada vértice tem total acesso aos seus dados, bem como aos dos vértices adjacentes e dos arcos, independentemente da sua orientação. Pode ainda agendar programas a serem executados por vértices vizinhos. Desta forma, é possível utilizar um modelo de comunicação assíncrono, em

[10] http://giraph.apache.org

que o momento em que o fluxo de dados se realiza é definido no programa do utilizador.

O assincronismo permite uma melhor utilização de recursos, e pode mesmo acelerar a convergência dalguns algoritmos, mas introduz problemas relacionados com a emergência de não-determinismo na computação. Para fazer face a esta dificuldade, o GraphLab assegura que qualquer execução é serializável, ou seja, que possui uma execução sequencial correspondente. Para esse efeito, utiliza um mecanismo de sincronização que impede vértices adjacentes de correrem simultaneamente o programa [27].

Modelo GAS e Powergraph Sistemas como o Pregel não distinguem conceptualmente, pelo menos de forma explícita, a computação do vértice em si e os aspetos ligados à comunicação de dados e à sumarização. É nesse contexto que surge uma proposta de abstração a nível superior dos algoritmos em grafos, o modelo Gather-Apply-Scatter (GAS) [27]:

Gather É recolhida, para cada vértice, informação dos vértices e arcos a ele adjacentes. Essa informação é sumarizada através duma função definida pelo utilizador, que deverá ser comutativa e associativa (de modo a que a ordem de execução não interfira no resultado final). Existem várias funções de sumarização possíveis, consoante o objetivo pretendido, entre as quais soma, máximo, mínimo, ou simplesmente a união de todos os valores.

Apply Utiliza-se o resultado da fase anterior para atualizar o valor associado ao vértice que está a ser processado.

Scatter Usa-se o novo valor associado ao vértice para atualizar a informação associada aos arcos que lhe são adjacentes.

Este modelo abstrato foi introduzido no âmbito dum outro sistema de processamento distribuído de grafos, o PowerGraph [27]. A decomposição GAS permite pensar a computação das mensagens duma forma *pull-based*, ou seja, é o sistema que se encarrega, através da invocação da fase *Scatter*, de obter as mensagens destinadas aos vértices adjacentes a determinado vértice, ao invés de ser o programa do utilizador a enviar as mensagens diretamente. Deste modo, é possível gerir a distribuição de trabalho e o fluxo de mensagens de forma mais eficiente e oportuna, permitindo uma série de otimizações.

Computação centrada nos arcos. O PowerGraph, mantendo uma lógica de programação orientada aos vértices, procura eliminar a dependência em relação ao grau dos vértices através da distribuição da execução do programa *pelos arcos*, divididos pelos vários nodos do sistema. A fase *Gather* é invocada paralelamente em cada arco e os resultados intermédios são reunidos e fundidos, passando o resultado final à fase seguinte. Do mesmo modo, a função *Scatter* é também executada em paralelo.

O foco na computação sobre os arcos induz outra forma de distribuir o processamento e os dados pelos vários nodos do sistema. Agora, ao invés dos

vértices, são os arcos que são distribuídos uniformemente pelas diversas máquinas. Disso resulta também que são agora os vértices que podem ser divididos por diferentes nodos. Assim, alterações aos dados dum vértice têm de ser comunicadas a todos os nodos que possuem uma cópia do mesmo.

O objetivo passa a ser portanto minimizar o número de vértices divididos entre máquinas diferentes. Cada arco é atribuído a um nodo do sistema de modo a minimizar o número de vértices divididos, mas sem que o desequilíbrio entre o número de nós atribuídos a diferentes nodos ultrapasse um fator constante predefinido. Existem diferentes técnicas para efetuar um *vertex-cut* nas condições descritas [27].

Chaos. O Chaos[11] é um sistema distribuído para grafos que resulta da extensão dum outro sistema, o X-Stream [40, 46], e que apresenta uma abordagem semelhante com iteração sobre os arcos.

O ponto distintivo deste sistema, porém, resulta do modelo de representação e armazenamento dos elementos do grafo (vértices, arcos e atualizações entre iterações): foca-se a utilização de armazenamento secundário em disco como forma de suportar a computação de grafos de grande dimensão, que podem chegar ao bilião de arcos. Outro ponto chave é a verificação de que, em *clusters* médios, a largura de banda em rede acaba frequentemente por suplantar o próprio acesso aos dados mantidos em armazenamento secundário. Assim, este sistema ignora simplesmente a localidade dos dados, concentrando os seus esforços em otimizar a leitura sequencial a partir do armazenamento secundário, otimizando, por exemplo, a dimensão dos blocos (*chuncks*) de dados. Como forma de equilibrar a distribuição de carga, este sistema implementa ainda uma forma de *work stealing* [47].

Bibliotecas para grafos de sistemas mais genéricos. Até agora foram abordados sistemas especializados no processamento de grafos. Porém, alguns sistemas de processamento de dados genéricos incluem também subsistemas especializados neste tipo de processamento, sob a forma de bibliotecas e APIs. Uma das principais vantagens deste modelo é a facilidade em integrar dados externos na computação dos grafos, associando-os a vértices e arcos, de forma quase transparente.

GraphX. O GraphX[12] é a biblioteca de processamento de grafos do Apache Spark. Esta biblioteca/API tem como base a verificação de que frequentemente a análise e processamento de grafos está inserida em tarefas de processamento muito maiores, em que para além dos grafos existem dados em formato tabular ou de chave-valor. Assim, o GraphX propõe-se unificar, na mesma *framework*, a análise de grafos e de dados não-estruturados e tabulares. Para esse efeito, permite-se efetuar a composição entre os dois tipos de dados, conforme as

[11] http://labos.epfl.ch/hpgp#chaos
[12] http://spark.apache.org/graphx

especificidades do objetivo pretendido, com os grafos podendo ser tratados na sua forma natural de grafos ou então como simples coleções de vértices e arcos[26].

Outra vantagem oferecida por esta estratégia reside na maior facilidade em construir grafos a partir de fontes distintas, em integrar informação externa no grafo e em comparar propriedades em grafos diferentes [57].

A representação dos grafos, na biblioteca GraphX, é feita através de tabelas, para vértices e arcos, baseadas em RDDs, já descritos na secção 3.1. As coleções de vértices e arcos podem ser obtidas através da API na forma de RDDs com algumas funcionalidades adicionais [28]. A tolerância a falhas é obtida através do uso de grafos *imutáveis* [56] e duma série de *operadores* [26, 28] que dão origem a novos grafos.

O particionamento do grafo pelos vários nodos do sistema distribuído é realizada através de *vertex-cuts*, na esteira da estratégia seguida pelo PowerGraph (secção 3.3), a fim de minimizar o *overhead* de comunicação. É possível porém controlar a forma como os arcos são distribuídos pelos nodos, através de funções de particionamento expressamente definidas, ou utilizar a estratégia por omissão, que tem em conta a forma como o próprio *input* se encontra particionado [56, 26].

Os operadores do GraphX, que operam sobre os grafos em termos de transformações efetuadas às coleções de vértices e arcos, permitem exprimir um grande número de algoritmos. Para simplificar o trabalho do utilizador/programador, é ainda oferecida uma versão otimizada da abstração introduzida pelo Pregel (secção 3.3), a *Pregel API*, que permite expressar a computação em termos de *vertex-programs* e trocas de mensagens. Algumas restrições, tais como reduzir a troca de mensagens apenas a vértices adjacentes, embora possam limitar alguma expressividade, permitem introduzir otimizações específicas [28, 57].

Gelly. O Flink disponibilizou também a sua biblioteca de processamento de grafos, o Gelly[13], em 2015.

A abordagem seguida pelo Gelly, bem como as funcionalidades que exibe, têm evidentes semelhanças com o GraphX. Tem-se em conta a necessidade de integrar a análise de grafos em *pipelines* analíticas, com diferentes fontes de dados, pelo que se oferece uma forma de integração facilitada. Vértices e arcos do grafo podem ser manipulados e consultados na forma de *DataSets*, com possibilidade de lhes associar qualquer tipo de dados [33].

É possível efetuar um grande número de operações sobre os grafos: transformação de valores (*map*), filtros, junção com outras fontes de dados por meio de chaves, e operações baseadas no conjunto dos vértices; operações para adicionar ou remover vértices e arcos; operações sobre a vizinhança dos vértices (outros vértices e arcos), não só de iteração, mas também de *reduce*.

[13] https://ci.apache.org/projects/flink/flink-docs-master/libs/gelly_guide.html

À semelhança do GraphX, o Gelly oferece também uma API para exprimir algoritmos sobre grafos no popular modelo *think-like-a-vertex* introduzido pelo Pregel (Sec. 3.3), a que se chamou *vertex iterations*. É oferecida ainda uma variante do modelo GAS (Sec. 3.3), denominada Gather-Sum-Apply (GSA), ainda que com algumas limitações [5].

Uma das importantes vantagens do Flink/Gelly no processamento de grafos, em relação ao MapReduce simples, reside no seu suporte nativo a computação iterativa, típica em problemas de grafos, com *caching* de resultados intermédios e das estruturas/elementos do grafo utilizados. Este sistema oferece ainda um meio de otimização adicional, as *delta iterations*, em que se separa claramente o conjunto dos elementos a atualizar em cada iteração dos elementos que não requerem computação, o que pode conduzir, como facilmente se depreende, a ganhos significativos em termos de tempo despendido e uso de recursos [32].

Sendo recente, o Gelly tem muitos pontos de expansão potenciais. Um dos projetos envolve o suporte a *streams* de grafos (secção 1.3), disponibilizando operadores que possibilitem a expressão de algoritmos neste tipo de dados. Este projeto encontra-se já parcialmente implementado [9]. Outros projetos presentes no roadmap do Gelly incluem operadores especializados em grafos fortemente heterogéneos/livres de escala, técnicas avançadas de particionamento de grafos, e operações otimizadas para vértices bipartidos, entre outras [24].

Críticas e outras abordagens. Uma das críticas que pode fazer-se aos paradigmas e sistemas expostos é que reduzem a expressividade dos algoritmos que podem ser expressos, e consequentemente dos problemas que podem ser resolvidos eficientemente em grafos. Um sintoma significativo é o facto de grande parte da investigação nesta área recorrer aos mesmos tipos de algoritmos para avaliar as soluções[14].

Existem classes de problemas em grafos que não se enquadram nos paradigmas vistos até agora, centrados em vértices ou em arcos, como os de *graph mining* (deteção de subgrafos frequentes, a contagem de padrões ou a enumeração de cliques, por exemplo). Problemas como estes envolvem a enumeração e a exploração exaustiva de subgrafos, o que torna abordagens como a do Pregel ineficientes pelo facto de envolverem demasiado movimento de mensagens e causarem sobrecarga em vértices de grau elevado [53].

Pode também observar-se que, no modelo centrado nos vértices, não existe controlo explícito sobre a forma como o grafo é particionado pelos vários nós de processamento, não se tomando assim partido de possíveis otimizações algorítmicas caso esse particionamento pudesse ser controlado pelo programador [54].

[14] Um exemplo típico de algoritmo já clássico é o PageRank.

3.4 Escalabilidade

Tradicionalmente, a necessidade de otimização do uso de recursos de processamento, em bases de dados, por exemplo, inseria-se num contexto em que os dados são relativamente estáveis e a fonte de instabilidade provém das diferentes análises e consultas aos mesmos dados. Nestes casos, havendo conhecimento completo dos dados envolvidos, os sistemas podem formular *query plans* a fim de otimizar a forma como as respostas são obtidas. Já no caso do processamento de *streams*, estamos perante a situação inversa: muitas vezes as consultas são sempre as mesmas, processadas de forma permanente, e o que varia é o afluxo de novos dados, frequentemente imprevisível. Facilmente surgem picos de afluência de dados na *stream*, que podem tornar o sistema instável e mesmo incapaz de efetuar o processamento requerido no tempo desejável [58].

A resposta mais imediata a este problema, reservar mais recursos computacionais, seja por ampliação física do *cluster*, seja por alocação de recursos virtuais num ambiente de *cloud*, não é escalável, conduz a uma utilização subótima dos recursos e pode comportar custos indesejáveis.

Assim, assumindo um contexto em que os recursos, físicos ou virtuais, são fixos, podemos distinguir duas grandes formas de responder a esta problemática: otimização do uso dos recursos existentes, por meio duma distribuição inteligente do processamento e/ou alteração dinâmica de parâmetros da computação (*tuning*); e o recurso ao processamento aproximado, descartando parte do trabalho excessivo em troca dum erro aceitável.

Otimização de recursos e *tuning* Face à imprevisibilidade do afluxo de dados ao sistema, já referida, mas também à natureza heterogénea quer das computações a efetuar, quer dos dados, quer dos recursos disponíveis [44], surgiram diferentes abordagens de resposta. A nível de *cluster*, foram apresentadas propostas para gerir eficientemente recursos entre diferentes sistemas e *frameworks* analíticas [30].

Ao nível dos sistemas individuais, caso que interessa mais para o presente trabalho, existe um grupo de estratégias que assenta na distribuição, ou redistribuição, da carga pelos nodos e recursos do sistema da forma que mais favorece o débito e a estabilidade do sistema como um todo. Analisando o grafo de tarefas e nodos, bem como o fluxo de dados, podem utilizar-se algoritmos que determinam, em *runtime*, a alocação de recursos mais eficiente [58, 4]. O uso de algoritmos que permitem ao sistema adaptar-se rapidamente a mudanças no fluxo de dados, e tomar partido dos recursos disponíveis é também conhecido como *computational elasticity* [48]

Outras estratégias recorrem à alteração de parâmetros da computação (*tuning*) durante a execução, como forma de promover a estabilidade e o desempenho do sistema. Um dos exemplos paradigmáticos é o tamanho do *batch* em que os dados são processados, em sistemas de processamento em bloco ou de *streams* em *mini-batch* [18].

Estas técnicas são úteis para fazer face a picos súbitos do fluxo de dados e manter a estabilidade e a capacidade de resposta do sistema, mas, além de não serem infalíveis, têm em conta apenas o aspeto quantitativo do fluxo de dados. Não oferecem resposta satisfatória à questão qualitativa, que formulámos nos inícios deste trabalho, de avaliar a diferença introduzida pelos novos resultados a fim de evitar que o custo da computação seja superior ao benefício da resposta.

Processamento aproximado. Podemos falar de processamento aproximado quando se admite uma determinada quantidade de erro no resultado que se obtém dos dados em troca de outras vantagens, tais como a rapidez na resposta ou poupança/otimização no uso de recursos utilizados para a obter. Existem inúmeras aplicações reais em que a completa exatidão da resposta não é tão importante como a rapidez da obtenção da mesma; considere-se, por exemplo, um sistema de recomendação de produtos *online*: uma resposta tardia, ainda que exata, pode ser desprovida de utilidade.

Podemos considerar duas grandes formas de utilização de processamento aproximado. A primeira delas ocorre quando os resultados são sempre aproximados, isto é, quando o sistema fornece sempre aproximações aos resultados reais, com erro controlado, o que pode acontecer quando a obtenção da resposta exata é muito dispendiosa, demorada, ou mesmo impossível. Uma segunda forma de processamento aproximado insere-se em sistemas que garantem *correção futura* (*eventual accuracy*) dos resultados: o sistema computa regularmente resultados exatos, mas nos intervalos da recomputação apresenta apenas resultados com uma certa aproximação (à semelhança do que foi visto na Sec. 3.2 a propósito da arquitetura Lambda).

O processamento aproximado surgiu no contexto de sistemas de apoio à decisão [36] e inclui diferentes técnicas, conforme o problema concreto a resolver e as estratégias empregadas na sua resolução. Na medida em que permite ter em conta não apenas os aspetos quantitativos, mas também os qualitativos, dos dados, e no seguimento do que foi apresentado como os objetivos da presente proposta de trabalho, o processamento aproximado apresenta-se como a linha de investigação a seguir, pelo que examinamos as suas principais técnicas mais de perto.

Load shedding. No contexto de processamento de *streams*, o *load shedding* consiste em descartar uma parte dos dados a processar, quando, por via dum súbito aumento do fluxo de dados, o sistema não tem capacidade de os processar a todos, ou quando fazê-lo introduz uma latência inaceitável. O problema de *load shedding* pode ser assim formalizado como um problema de otimização em que se procura maximizar a exatidão dos resultados, com a restrição de que a taxa de entrada de novos dados não pode ser superior à taxa a que eles são processados [8]. No caso dos sistemas distribuídos, a questão apresenta complexidade adicional, pois cada um dos nodos pode ter os seus próprios re-

quisitos e limitações. Podem conceber-se estratégias centralizadas, em que os diversos nodos concordam num plano de *load shedding* comum, ou locais, em que cada nodo procede, de forma controlada, à seleção de *input* a desprezar [51].

Esta técnica também é utilizada em contexto de bases de dados/*data warehouses*, quando para responder a determinada consulta não se tem em conta uma parte dos dados. Neste contexto, esta estratégia recebe o nome de *data skipping*, sendo porém idêntica, na sua essência [50].

As questões que se impõem são *quando* descartar dados (detetar as condições em que o mesmo deve de ser realizado), *onde* os descartar (em que ponto da *pipeline* de processamento devem ser os dados/resultados descartados), e que *quantidade* e *quais* os dados a desprezar [52]. A deteção do excesso de carga do sistema envolve algoritmos específicos que não importa discutir aqui. Quanto ao lugar onde os dados devem ser descartados, é fácil de ver que quanto mais cedo na *pipeline* analítica forem eliminados, menos processamento desnecessário terá sido realizado; contudo, frequentemente o fluxo de dados não é linear entre *streams*, e uma eliminação precoce pode implicar erros intoleráveis [52].

Quanto à quantidade e quais os dados a não ter em conta, a abordagem mais simples é meramente quantitativa e consiste em descartar *input* de forma aleatória. Com esta técnica, conhecendo a distribuição dos dados, pode estimar-se e limitar-se o erro resultante através de métodos estatísticos [8].

Existem porém técnicas mais avançadas, que têm em conta a importância dos dados para a computação em causa, e procuram efetuar um *load shedding semântico*, inspirado no conceito de Quality-of-Service (QoS), bastante utilizado no contexto de tráfego em redes. Para esse efeito, há que determinar a importância relativa dos dados e seus valores para os resultados finais, seja através de regras decorrentes da lógica do problema, seja através de métodos estatísticos ou de aprendizagem computacional baseados no histórico de dados processados. Estas técnicas permitem obter predicados que determinam que dados remover do processamento [52].

Amostragem. O *load shedding* aleatório é muito semelhante, na sua técnica, à amostragem do *input*, com a possível diferença de que, quando se fala em amostragem, a amostra ter uma dimensão geralmente bastante inferior ao conjunto de dados original; no caso do *load shedding*, são os dados descartados a minoria. Por outro lado, *load shedding* faz sentido apenas no contexto de *streams*, ao contrário da amostragem, que é uma técnica mais geral e que é aplicada frequentemente a dados estáticos ou pouco variáveis.

A amostragem foi proposta já há décadas como meio de otimizar o tempo de resposta de consultas a bases de dados, ou como forma de previsão de resultados. No caso dos sistemas distribuídos, a amostragem é uma técnica útil para reduzir, por vezes de forma considerável, os recursos computacionais

utilizados na obtenção da resposta e o tempo da mesma, à custa dum erro aceitável.

Existem diferentes formas de efetuar amostragem em sistemas de processamento em larga escala. Para além da amostragem do *input* de dados, é possível também recorrer a *task dropping*, quando se descartam, ao invés do *input*, algumas das tarefas em que o processamento global é particionado (por exemplo, por nodo). Esta técnica pode servir também como estratégia de tolerância a falhas: descartam-se as tarefas que falharam por algum motivo ou as que estão demasiado demoradas (*stragglers*), estimando o erro introduzido [45]. Existem também técnicas mais avançadas, ajustadas à lógica do problema, e que recorrem a funções definidas pelo utilizador [25]. Técnicas e modelos estatísticos, tais como amostragem multi-nível ou teoria de valores extremos permitem efetuar o cálculo a partir da amostra e extrapolar o resultado para o conjunto dos dados, com um erro máximo previsto que pode ser calculado [25].

O erro, nestes casos, é habitualmente apresentado na forma de margens de erro para um determinado intervalo de confiança. O utilizador do sistema tanto pode estabelecer o erro máximo admissível, devendo o sistema selecionar a amostra conveniente para essa finalidade, como, de modo inverso. determinar o tamanho da amostra, sendo o erro estimado apresentado pelo sistema [25].

Em bases de dados, ou sistemas em que os dados são estáveis, podem pré-computar-se amostras de diferentes tamanhos e estratificações diferentes, a selecionar conforme a consulta pretendida e o erro máximo admissível [1]. Quando os dados sofrem constante atualização, como é o caso dum sistema de processamento de *streams*, esta estratégia não pode ser aplicada diretamente, mas podem encontrar-se soluções mistas, com atualização de amostras pré-computadas, por exemplo. O mesmo tipo de solução pode servir para um sistema em arquitetura Lambda.

Um outro problema a considerar neste âmbito prende-se com o cálculo do erro e a confiança que a esse cálculo também está associada. Não existe um método de estimação do erro ideal, e algumas experiências mostraram que o erro reportado, em cenários reais, peca frequentemente ora por otimismo (devolvendo resultados na verdade mais distantes do resultado correto do que seria expectável), ora por pessimismo (selecionando amostras demasiado grandes e alocando recursos desnecessariamente, para o nível de erro que seria aceitável). Para fazer face a este problema, têm sido também desenvolvidas técnicas de diagnóstico para determinar a confiança que pode ser dada ao erro calculado por determinado método (o *erro do erro*) [2].

A amostragem de grafos constitui por si só um vasto campo de estudo, no qual existe ainda muito trabalho em aberto. Existem variadas técnicas de amostragem para grafos, baseadas por exemplo nos vértices, nos arcos ou em *random walks*, cuja finalidade é preservar, para o subgrafo obtido como amostra, propriedades semelhantes ao grafo original, a fim de que a amostra possa ser usada como estimador. Diferentes técnicas de amostragem permitem

preservar diferentes propriedades, ou são mais indicadas para distintos tipos de grafos [31].

Outras técnicas. Para além da aproximação por métodos estatísticos, existem ainda outras abordagens, baseadas por exemplo em aprendizagem automática. O objetivo é, a partir do histórico de processamento, inferir a correlação entre os novos dados de entrada e as diferenças no output calculado. Quando essa relação não é direta, existem métodos de aprendizagem, tais como redes neuronais, *fuzzy logic*, métodos probabilísticos, etc., que permitem obter eficazmente previsões mais acertadas sobre as potenciais alterações que novos dados introduzem nos resultados [23].

Quality-of-Data. A otimização do uso dos recursos computacionais e do tempo de resposta, por meio do atraso da recomputação, quando não se prevê que haja diferença de *output* relevante, exige uma forma de exprimir os requisitos, diferentes para cada problema e aplicação, em termos de tolerância ao erro e do nível de alterações que podem desencadear a recomputação, assim como as exigências mínimas de latência e débito para a aplicação. Estes requisitos podem ser definidos através do conceito de Quality-of-Data (QoD).

A QoD pode definir-se, para cada problema, através de três dimensões [21, 23]:

Tempo O tempo máximo entre recomputações sucessivas. Durante esse intervalo, podem ser devolvidos resultados aproximados, mas após um *timeout*, os resultados exatos devem ser recalculados.

Sequência Quantidade máxima de atualizações recebidas para um determinado objeto, que ao ser excedida implica a recomputação de resultados.

Valor Divergência relativa máxima entre o estado dum objeto na última computação e o seu estado atual. É nesta dimensão que é importante determinar o potencial efeito dos novos dados de entrada nos resultados a obter.

Os requisitos de QoD podem ser expressos por meio de regras em qualquer um destes eixos, passíveis de serem combinadas por meio de operadores lógicos.

Uma outra possibilidade consiste em recorrer a amostragem para estimar o efeito das alterações entretanto ocorridas na computação global. Para além de se poder utilizar o resultado da aproximação na resposta a devolver (ao invés de simplesmente repetir a última resposta), poderia utilizar-se o erro estimado da amostra, ou a divergência em relação ao último valor exato obtido, como indicador para a necessidade de recomputar.

4 Solução proposta

Passamos agora a descrever a solução que propomos. O trabalho consistirá na criação duma biblioteca que possa ser facilmente integrada com um sistema de processamento de *streams*, e com um sistema de processamento de grafos.

A *stream* pode conter atualizações topológicas ou de propriedades dos vértices e arcos do grafo, e flui de forma contínua, ou em intervalos não especificados.

4.1 Arquitetura

O esboço da arquitetura da solução é apresentado na Fig. 1.

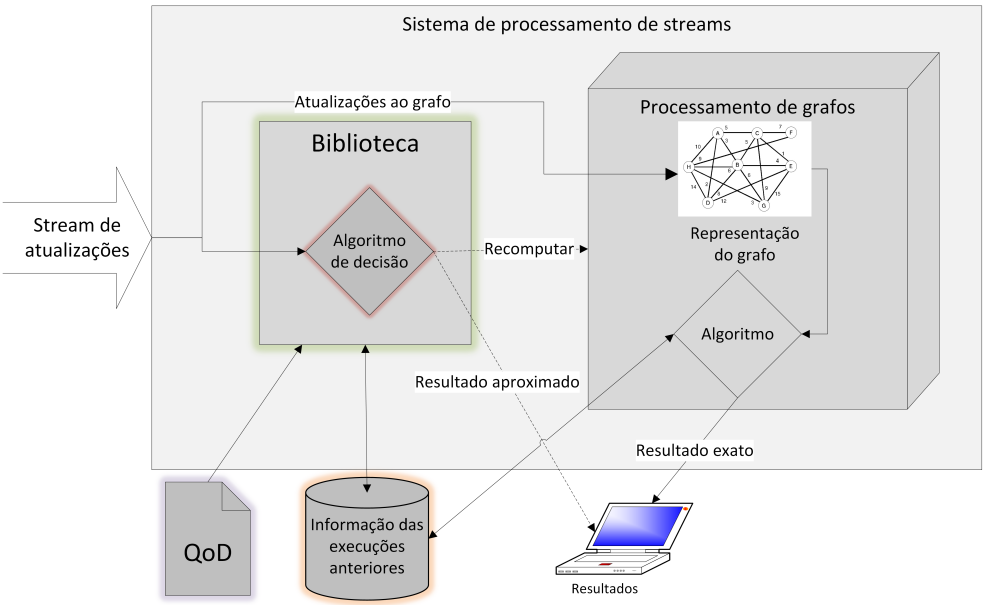

Figura 1. Esboço da arquitetura da solução

A presente solução distingue a aplicação das alterações ao grafo, que podem ser refletidas de imediato no grafo, de forma eficiente, e a computação das funções pretendidas sobre o grafo, que é o que aqui se pretende otimizar. Assim, a atualização de propriedades é efetuada pelo sistema de processamento de grafos, e ao mesmo tempo avaliada pela biblioteca a criar, a fim de determinar se as alterações efetuadas justificam a recomputação da função.

Como elementos de entrada para a decisão, para além da *stream* de atualizações ao grafo, existe uma definição da QoD apropriada, fornecida pelo utilizador do sistema, e ainda dados relativos aos últimos processamentos efetuados sobre o grafo (resultado, *timestamp* e outros elementos que se venham a considerar relevantes).

4.2 Algoritmo

O aspeto central da solução é o algoritmo de decisão, que determina qual o resultado a devolver, se o proveniente duma recomputação, se o baseado numa

Algoritmo 1 Algoritmo para decidir a recomputação, com base em QoD

```
 1: function RECOMPUTAR                                  ▷ Recomputar os resultados de raiz
 2:     r ← executar algoritmo...
 3:     info.timestamp ← tempo atual
 4:     info.pendentes ← 0
 5:     return r
 6: end function

 7: function DECIDIR-RECOMPUTAÇÃO
 8:     if tempo-atual - info.timestamp > QoD.tempo-máximo
 9:      ou info.pendentes > QoD.máximo-pendentes then
10:         return RECOMPUTAR
11:     end if
12:     for cada r ∈ info.regras do
13:         if r é verdadeira then
14:             return RECOMPUTAR
15:         end if
16:     end for
17:     info.pendentes ← info.pendentes + quantidade de novos dados
18:     if QoD.usar-amosta then
19:         r ← executar algoritmo na amostra do grafo e extrapolar...
20:         if erro estimado da amostra > QoD.erro-máximo
21:           ou r-info.último-resultado > QoD.diferença-máxima then
22:             return RECOMPUTAR
23:         end if
24:         return r
25:     end if
26:     return info.último-resultado
27: end function
```

amostra, se o último calculado. O algoritmo 1 apresenta o pseudo-código do procedimento.

De acordo com a especificação da QoD definida pelo utilizador, caso o tempo desde a última computação, ou o número acumulado de atualizações pendentes, ultrapassar os limites definidos, a função é recomputada (linha 9). O mesmo sucede se qualquer uma das regras referentes à divergência máxima entre valores associados ao grafo e aos seus elementos for verificada (linha 12).

Por fim, a solução que propomos vai, à partida, prever a possibilidade de recorrer a amostragem do grafo, se tal for possível e desejado. Nesse caso, a amostra (cujo modo de extração será alvo de atenção futura, e envolverá possivelmente intervenção do utilizador) é utilizada para calcular a função pretendida de forma mais rápida e eficiente do que procedendo à recomputação (linha 18). O resultado é extrapolado/ajustado para o grafo inteiro, se necessário, e com ele é também devolvido o erro estimado, dentro dum intervalo de confiança definido. No caso de o erro estimado superar o máximo definido pela QoD, ou a diferença, absoluta ou relativa, entre o resultado estimado e o último resultado exato conhecido, ser superior a um limite também definido

pela QoD, procede-se mais uma vez à recomputação de resultados (linha 21). Senão, pode devolver-se o resultado estimado.

Em último caso, quando nenhuma das condições anteriores se verifica, é devolvido o último resultado exato calculado, armazenado no sistema (linha 26).

4.3 Tecnologia envolvida e plano de realização

Resta definir sobre que base concreta, em termos de tecnologias, será esta solução desenvolvida.

Como é sugerido pelo esboço de arquitetura apresentado da Fig. 1, é vantajoso que o processamento de *streams* e o processamento de grafos se integrem da forma mais natural e direta possível. Assim, a escolha restringe-se aos sistemas que apresentam soluções de processamento de grafos inseridas em sistemas mais gerais: Spark/GraphX (Sec. 3.1, 3.2 e 3.3) e Flink/Gelly (Sec. 3.2 e 3.3.

Dada a inconveniência de trabalhar nas duas bases simultaneamente, a nossa escolha recai sobre o Flink/Gelly. As razões para a escolha prendem-se com a versatilidade do modelo de *streaming* empregado, que no Flink é híbrido, podendo utilizar-se streaming puro, elemento a elemento, ou processar em bloco (no Spark apenas os *mini-batches* estão disponíveis). Apesar de ser menos utilizado, e se encontrar num estágio de maturidade inferior ao Spark, o Flink é um projeto de topo do Apache, criado na Europa, e com uma comunidade crescente de colaboradores e de utilizadores. O facto de se encontrar praticamente ainda nos seus inícios acentua o potencial de contribuição que se pode obter com o presente projeto.

5 Metodologia de avaliação

Apresentam-se agora os pontos a avaliar na solução a desenvolver, com alusão aos *benchmarks* que podem ser utilizados para o efeito.

Existem vários *datasets* de grafos de grande dimensão disponíveis, que podem ser utilizados para explorar, exercitar e avaliar sistemas, soluções e algoritmos. Destacamos os grafos obtidos a partir de *crawls* da WWW, de diferentes dimensões, incluindo *datasets* com dezenas de milhões de vértices e centenas de milhões de arcos, que dada a sua dimensão requerem técnicas especializadas de representação, como o WebGraph [15]. Exemplares de menor dimensão, mas ainda assim na casa dos milhões de vértices incluem por exemplo as redes de ligações entre páginas da Wikipédia em várias línguas [34].

Todos estes exemplos são de grafos estáticos, pelo que é necessária alguma adaptação para os colocar na forma de *streams*, o objeto que nos interessa avaliar, recorrendo, por exemplo, a construção incremental.

Existem também *datasets* adaptados para *streaming*, disponíveis online [49], que reproduzem o crescimento de diferentes redes reais, e das quais desta-

camos: citações em trabalhos científicos; filmes; redes sociais (Facebook, Flickr e Youtube).

A fim de ter uma visão global sobre o desempenho do sistema, e evitar fenómenos de *overfitting*, o sistema será avaliado com diferentes tipos de problemas em grafos. Assim, explorar-se-ão problemas de centralidade (PageRank, *betweeness*), caminhos/distâncias, *clustering*/comunidades, dinâmicas de difusão (como epidemias), cortes, dentro do exequível. Se possível, utilizar-se-ão também diferentes técnicas para a resolução do mesmo problema.

As métricas a avaliar serão:

Latência Comparação da latência da nossa solução com a apresentada pelo sistema de processamento de grafos sem otimizações.

Débito (*Throughput*) Avaliar o débito do sistema em diferentes cenários de carga, em termos de quantidade de itens a processar.

Escalabilidade Desempenho e estabilidade do sistema na presença de grandes quantidades de dados e com diferentes números de nodos de processamento.

Uso de recursos Comparar o uso dos recursos computacionais (CPU, memória, disco, rede) no sistema com e sem as alterações introduzidas pela solução que propomos.

Qualidade de resposta Comparar os resultados aproximados obtidos pela nossa solução com os resultados exatos e verificar se satisfazem os requisitos para diferentes especificações de QoD.

Exatidão Verificar se o erro estimado para as aproximações corresponde ao erro real, para intervalos de confiança definidos.

6 Conclusão

Neste projeto de dissertação começámos por contextualizar o problema do processamento de grafos, apontando a sua importância e os desafios que comporta. Referimos de seguida a motivação e a inspiração que estão na base desta proposta de trabalho, formulando o problema a resolver em termos de otimização do uso de recursos e de tempo de resposta no processamento de *streams* de grafos.

Na análise do trabalho relacionado, sintetizámos os principais aspetos dos atuais sistemas de processamento em bloco e de *streams*, com destaque para os sistemas e bibliotecas especializados em grafos, terminando com referências a técnicas de otimização e estabilização e de processamento aproximado, com uma explicitação do conceito de Quality-of-Data.

A arquitetura e algoritmo base da solução proposta, bem como as linhas e métricas orientadoras da avaliação da mesma solução terminam o presente documento.

Referências

[1] Sameer Agarwal et al. «BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data». Em: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 29–42. ISBN: 9781450319942. URL: http://doi.acm.org/10.1145/2465351.2465355.

[2] Sameer Agarwal et al. «Knowing when You'Re Wrong: Building Fast and Reliable Approximate Query Processing Systems». Em: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 481–492. ISBN: 9781450323765. URL: http://doi.acm.org/10.1145/2588555.2593667.

[3] Alexander Alexandrov et al. «The Stratosphere platform for big data analytics». Em: *The VLDB Journal* 23.6 (mai. de 2014), pp. 939–964. ISSN: 1066-8888. URL: http://link.springer.com/10.1007/s00778-014-0357-y.

[4] L. Amini et al. «Adaptive Control of Extreme-scale Stream Processing Systems». Em: *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE, 2006, pp. 71–71. ISBN: 0769525407. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1648858.

[5] *Apache Flink 1.0-SNAPSHOT Documentation: Gelly: Flink Graph API*. URL: https://ci.apache.org/projects/flink/flink-docs-master/libs/gelly_guide.html (acedido em 27/12/2015).

[6] *Apache Flink: Features*. URL: http://flink.apache.org/features.html (acedido em 22/12/2015).

[7] *Apache Spark™ - Lightning-Fast Cluster Computing*. URL: http://spark.apache.org (acedido em 15/12/2015).

[8] Brain Brian Babcock et al. «Load Shedding Techniques for Data Stream Systems». Em: *In Proc. of the 2003 Workshop on Management and Processing of Data Streams (MPDS*. 2003, pp. 1–3. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.1941.

[9] János Dániel Bali. «Streaming Graph Analytics Framework Design». Tese de mestrado. KTH, School of Information e Communication Technology (ICT), 2015. URL: http://kth.diva-portal.org/smash/get/diva2:830662/FULLTEXT01.pdf.

[10] Albert-László Barabási. *Network Science*. Cambridge University Press, 2016. ISBN: 9781107076266. URL: http://barabasi.com/networksciencebook/. Disponível online. Publicação física prevista para abril de 2016.

[11] Albert-László Barabási e Réka Albert. «Emergence of Scaling in Random Networks». Em: *Science* 286.5439 (out. de 1999), pp. 509–512. URL: http://www.sciencemag.org/content/286/5439/509.abstract.

[12] Alain Barrat, Marc Barthélemy e Alessandro Vespignani. *Dynamical Processes on Complex Networks*. 1st. New York, NY, USA: Cambridge University Press, 2008. ISBN: 9780521879507. URL: http://www.cambridge.org/us/academic/subjects/physics/statistical-physics/dynamical-processes-complex-networks.

[13] *Benchmarking Streaming Computation Engines at... | Yahoo Engineering*. URL: http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at (acedido em 21/12/2015).

[14] Pramod Bhatotia et al. «Incoop: MapReduce for Incremental Computations». Em: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. SOCC '11. New York, NY, USA: ACM, 2011, 7:1–7:14. ISBN: 9781450309769. URL: http://doi.acm.org/10.1145/2038916.2038923.

[15] P Boldi e S Vigna. «The Webgraph Framework I: Compression Techniques». Em: *Proceedings of the 13th International Conference on World Wide Web*. WWW '04. New York, NY, USA: ACM, 2004, pp. 595–602. ISBN: 158113844X. URL: http://doi.acm.org/10.1145/988672.988752.

[16] Paolo Boldi e Sebastiano Vigna. «The webgraph framework II codes for the world-wide web». Em: *Data Compression Conference, 2004. Proceedings. DCC 2004*. IEEE, 2004, pp. 528–528. ISBN: 0769520820. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1281504.

[17] *Concepts*. URL: http://storm.apache.org/documentation/Concepts.html (acedido em 21/12/2015).

[18] Tathagata Das et al. «Adaptive Stream Processing Using Dynamic Batch Sizing». Em: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. New York, NY, USA: ACM, 2014, 16:1–16:13. ISBN: 9781450332521. URL: http://doi.acm.org/10.1145/2670979.2670995.

[19] Jeffrey Dean e Sanjay Ghemawat. «MapReduce: Simplified Data Processing on Large Clusters». Em: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 1–13. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[20] D Ediger et al. «Massive streaming data analytics: A case study with clustering coefficients». Em: *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. 2010, pp. 1–8. URL: http://www.cc.gatech.edu/~bader/papers/StreamingCC.html.

[21] Sérgio Esteves, João Nuno Silva e Luís Veiga. «Fluchi: a quality-driven dataflow model for data intensive computing». Em: *Journal of Internet Services and Applications* 4.1 (2013), p. 12. ISSN: 1869-0238. URL: http://www.jisajournal.com/content/4/1/12.

[22] Sérgio Esteves e Luís Veiga. «WaaS: Workflow-as-a-Service for the Cloud with Scheduling of Continuous and Data-Intensive Workflows». Em: *The Computer Journal* (2015). URL: http://comjnl.oxfordjournals.org/content/early/2015/01/08/comjnl.bxu158.abstract.

[23] Sérgio Esteves et al. «Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets». Em: *Journal of Parallel and Distributed Computing* 79-80 (mai. de 2015), pp. 52–66. ISSN: 07437315. URL: http://www.sciencedirect.com/science/article/pii/S0743731515000507.

[24] *Flink Gelly - Apache Flink - Apache Software Foundation*. URL: https://cwiki.apache.org/confluence/display/FLINK/Flink+Gelly (acedido em 28/12/2015).

[25] Inigo Goiri et al. «ApproxHadoop: Bringing Approximations to MapReduce Frameworks». Em: *SIGPLAN Not.* 50.4 (2015), pp. 383–397. ISSN: 0362-1340. URL: http://doi.acm.org/10.1145/2775054.2694351.

[26] Joseph E Gonzalez et al. «GraphX: Graph Processing in a Distributed Dataflow Framework». Em: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 599–613. ISBN: 9781931971164. URL: http://dl.acm.org/citation.cfm?id=2685048.2685096.

[27] Joseph E Gonzalez et al. «PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs». Em: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 17–30. ISBN: 9781931971966. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez.

[28] *GraphX - Spark 1.6.0 Documentation*. URL: http://spark.apache.org/docs/latest/graphx-programming-guide.html (acedido em 07/01/2016).

[29] *High-throughput, low-latency, and exactly-once stream processing with Apache Flink | data Artisans*. URL: http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink (acedido em 21/12/2015).

[30] Benjamin Hindman et al. «Mesos: A Platform for Fine-grained Resource Sharing in the Data Center». Em: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

[31] Pili Hu e Wing Cheong Lau. «A Survey and Taxonomy of Graph Sampling». Em: *CoRR* abs/1308.5 (ago. de 2013). URL: http://dblp.org/rec/journals/corr/HuL13.

[32] Vasiliki Kalavri. *Gelly: Large-Scale Graph Processing with Apache Flink*. 2015. URL: http://pt.slideshare.net/vkalavri/gelly-in-apache-flink-bay-area-meetup (acedido em 28/12/2015).

[33] Vasiliki Kalavri. *Large-scale graph processing with Apache Flink @GraphDevroom FOSDEM'15*. 2015. URL: http://pt.slideshare.net/vkalavri/largescale-graph-processing-with-apache-flink-graphdevroom-fosdem15 (acedido em 28/12/2015).

[34] *Laboratory for Web Algorithmics*. URL: http://law.di.unimi.it/datasets.php (acedido em 05/01/2016).

[35] Daewoo Lee, Jin-Soo Kim e Seungryoul Maeng. «Large-scale incremental processing with MapReduce». Em: *Future Generation Computer Systems* 36 (jul. de 2014), pp. 66–79. ISSN: 0167739X. URL: http://www.sciencedirect.com/science/article/pii/S0167739X13001891.

[36] Qing Liu. *Approximate Query Processing*. English. Ed. por LING LIU e M.TAMER ÖZSU. Boston, MA, 2009. URL: http://www.springerlink.com/index/10.1007/978-0-387-39940-9_534.

[37] Yucheng Low et al. «Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud». Em: *Proc. VLDB Endow.* 5.8 (2012), pp. 716–727. ISSN: 2150-8097. URL: http://dx.doi.org/10.14778/2212351.2212354.

[38] Yucheng Low et al. «GraphLab: A New Parallel Framework for Machine Learning». Em: *Conference on Uncertainty in Artificial Intelligence (UAI)*. Catalina Island, California, 2010. URL: http://arxiv.org/abs/1006.4990.

[39] Grzegorz Malewicz et al. «Pregel: A System for Large-scale Graph Processing». Em: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. ISBN: 9781450300322. URL: http://doi.acm.org/10.1145/1807167.1807184.

[40] Jasmina Malicevic, Amitabha Roy e Willy Zwaenepoel. «Scale-up Graph Processing in the Cloud: Challenges and Solutions». Em: *Proceedings of the Fourth International Workshop on Cloud Data and Platforms*. CloudDP '14. New York, NY, USA: ACM, 2014, 5:1–5:6. ISBN: 9781450327145. URL: http://doi.acm.org/10.1145/2592784.2592789.

[41] Nathan Marz e James Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. 1ª ed. Greenwich, CT, USA: Manning Publications Co., 2015. ISBN: 9781617290343. URL: https://www.manning.com/books/big-data.

[42] Andrew McGregor. *Graph Mining on Streams*. Ed. por LING LIU e M.TAMER ÖZSU. Boston, MA, 2009. URL: http://www.springerlink.com/index/10.1007/978-0-387-39940-9_184.

[43] Daniel Peng e Frank Dabek. «Large-scale Incremental Processing Using Distributed Transactions and Notifications». Em: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. 2010. URL: http://research.google.com/pubs/pub36726.html.

[44] Charles Reiss et al. «Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis». Em: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. New York, NY, USA: ACM, 2012, 7:1–7:13. ISBN: 9781450317610. URL: http://doi.acm.org/10.1145/2391229.2391236.

[45] Martin Rinard. «Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks». Em: *Proceedings of the 20th Annual International Conference on Supercomputing*. ICS '06. New York, NY, USA: ACM, 2006, pp. 324–334. ISBN: 1595932828. URL: http://doi.acm.org/10.1145/1183401.1183447.

[46] Amitabha Roy, Ivo Mihailovic e Willy Zwaenepoel. «X-Stream: Edge-centric Graph Processing Using Streaming Partitions». Em: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. New York, NY, USA: ACM,

2013, pp. 472–488. ISBN: 9781450323888. URL: http://doi.acm.org/10.1145/2517349.2522740.

[47] Amitabha Roy et al. «Chaos». Em: *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*. New York, New York, USA: ACM Press, out. de 2015, pp. 410–424. ISBN: 9781450338349. URL: http://dl.acm.org/citation.cfm?id=2815400.2815408.

[48] S Schneider et al. «Elastic scaling of data parallel operators in stream processing». Em: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 2009, pp. 1–12. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5161036.

[49] *Streamed Graph Datasets*. URL: http://www.eecs.wsu.edu/~yyao/StreamingGraphs.html (acedido em 05/01/2016).

[50] Liwen Sun et al. «Fine-grained partitioning for aggressive data skipping». Em: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*. SIGMOD '14. New York, New York, USA: ACM Press, 2014, pp. 1115–1126. ISBN: 9781450323765. URL: http://dl.acm.org/citation.cfm?doid=2588555.2610515.

[51] Nesime Tatbul, Uğur Çetintemel e Stan Zdonik. «Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing». Em: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. VLDB Endowment, 2007, pp. 159–170. ISBN: 9781595936493. URL: http://dl.acm.org/citation.cfm?id=1325851.1325873.

[52] Nesime Tatbul et al. «Load Shedding in a Data Stream Manager». Em: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. VLDB Endowment, 2003, pp. 309–320. ISBN: 0127224424. URL: http://dl.acm.org/citation.cfm?id=1315451.1315479.

[53] Carlos H C Teixeira et al. «Arabesque: A System for Distributed Graph Mining». Em: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. New York, NY, USA: ACM, 2015, pp. 425–440. ISBN: 9781450338349. URL: http://doi.acm.org/10.1145/2815400.2815410.

[54] Yuanyuan Tian et al. «From 'think like a vertex' to 'think like a graph'». Em: *Proceedings of the VLDB Endowment* 7.3 (nov. de 2013), pp. 193–204. ISSN: 21508097. URL: http://dl.acm.org/citation.cfm?doid=2732232.2732238.

[55] Tom White. *Hadoop: The Definitive Guide*. 4ª ed. O'Reilly Media, Inc., 2015. ISBN: 9781491901632. URL: http://hadoopbook.com/.

[56] Reynold S Xin et al. «GraphX: A Resilient Distributed Graph System on Spark». Em: *First International Workshop on Graph Data Management Experiences and Systems*. GRADES '13. New York, NY, USA: ACM, 2013, 2:1–2:6. ISBN: 9781450321884. URL: http://doi.acm.org/10.1145/2484425.2484427.

[57] Reynold S Xin et al. «GraphX: Unifying Data-Parallel and Graph-Parallel Analytics». Em: *CoRR* abs/1402.2 (2014). URL: http://arxiv.org/abs/1402.2394.

[58] Ying Xing, Stan Zdonik e Jeong-Hyon Hwang. «Dynamic Load Distribution in the Borealis Stream Processor». Em: *Proceedings of the 21st International Conference on Data Engineering*. ICDE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 791–802. ISBN: 0769522858. URL: http://dx.doi.org/10.1109/ICDE.2005.53.

[59] Matei Zaharia et al. «Discretized Streams: Fault-tolerant Streaming Computation at Scale». Em: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13 1. New York, NY, USA: ACM, 2013, pp. 423–438. ISBN: 9781450323888. URL: http://doi.acm.org/10.1145/2517349.2522737.

[60] Matei Zaharia et al. «Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing». Em: *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), pp. 2–2. ISSN: 00221112. URL: http://dl.acm.org/citation.cfm?id=2228301.

[61] Matei Zaharia et al. «Spark : Cluster Computing with Working Sets». Em: *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), p. 10. ISSN: 03642348. URL: http://dl.acm.org/citation.cfm?id=1863113.

Anexo - Planeamento do trabalho

Tabela 1. Tempo estimado para o trabalho restante

Tarefa	**Descrição**	**Tempo previsto**
Introdução ao Flink/Gelly	- Fork do projeto - Tutoriais e exemplos	Janeiro (1 semana) Fevereiro (2 semanas)
Datasets e algoritmos	- Recolha e preparação dos datasets - Execução de algoritmos - Escrita de algoritmos no Gelly	Fevereiro (2 semanas)
Implementação	- Modelo e implementação da QoD - Implementação de QoD de tempo e sequência - Implementação da previsão de alterações e do processamento adiado - Amostragem de grafos	Março (4 semanas) Abril (3 semanas)
Testes e conclusão da implementação	- Testar e validar a solução - Concluir as funcionalidades - Corrigir erros	Maio (4 semanas)
Avaliação da solução	- Avaliação da solução com diferentes algoritmos, datasets e cenários - Comparação com outras soluções	Junho (3 semanas) Julho (2 semanas)
Escrita da tese de dissertação	- Agregação dos materiais escritos - Redação do texto final	Julho (2 semanas) Agosto (4 semanas)
Revisões e submissão	- Revisão da dissertação e da documentação do código	Setembro (2 semanas)
Investigação e documentação	- Leitura de artigos mais recentes - Acompanhamento do estado da arte - Tópicos avançados - Documentação da arquitetura, das opções, da implementação e da avaliação	Janeiro - setembro
Reuniões semanais	- Análise do progresso e orientação do trabalho	Janeiro - setembro