Chapter 5

Algoritmos e Modelos de Programação em Big Data

Fabio Porto (LNCC)

Abstract

Big Data is a phenomenon that currently attracts a huge attention both in academia and industry. It is understood as an important asset as it provides data about all sorts of activities, in different granularity from individuals to huge groups and from all sorts of phenomena. The data deluge resulting from the Big Data phenomenon needs to be processed and interpreted. The later however is challenging as data do not fit in memory of a single computer and the type of analysis usually involves scanning almost all the available data. In order to deal with these challenges new parallel data processing models have emerged enabling the distribution of data and processing, potentially through thousands of machines. In this course, we wil present the MapReduce programming model and its implementation in the Spark Framework. We will also discuss algorithms that can help implementing applications on top of this model.

Resumo

Big Data é um fenômeno que atrai grande atenção no meio acadêmico e nas empresas. É considerado um importante ativo, presente em todas as atividades, e envolvendo dados sobre indivíduos, assim como sobre grupos, como aqueles em redes sociais. Para ser transformado em informação, o grande volume de dados proporcionado pelo fenômeno Big Data precisa ser processado e interpretado. O processamento, no entanto, impõe um grande desafio, uma vez que a quantidade de dados é tal que não cabe na memória principal de uma máquina, ainda que potente, e as análises via de regra varrem todo o conjunto disponível. De forma a lidar com esse desafio, foram propostos frameworks de processamento paralelo, como o MapReduce, que oferece uma interface simples de programação e um sistema de processamento que escala a até milhares de máquinas. O framework Apache Spark é um dos que se destaca neste panorama. Neste curso, apresentaremos o modelo de programação MapReduce e sua implementação Apache Spark. Discutiremos ainda algoritmos que auxiliam no desenvolvimento de aplicações neste paradigma.

5.1. Introdução

O fenômeno Big Data se apresenta como uma das novidades dessa década com maior impacto na sociedade. Realçado pela mídia [Economist, 2010, Nature, 2008], Big Data passou a ser um modelo de negócios e ciência em quase todos os domínios. Na ciências, podemos citar a astronomia como uma das primeiras a adotar a abordagem baseada em dados para avaliação de hipóteses científicas. Levantamentos digitais, tais como o Sloan Digital Sky Survey [SDSS,], se baseiam inteiramente em dados sobre corpos celestes extraídos de imagens capturadas pelo telescópio Sloan Foundation, localizado no Novo México. Igualmente, na bioinformática, bases de dados como as do GeneBank, no NCBI, crescem continuamente com a publicação de dados de sequenciamento genético de novos organismos, permitindo a investigação por cientistas e empresas. É claro que não se pode deixar de mencionar as grandes empresas ligadas a Web, como a Google, e as redes sociais, como o Facebook. Atualmente, essas empresas são os maiores investidores em processos automatizados puramente baseadas na análise de dados de registros de operações realizadas em seus sites. Processos como os de recomendação comercial, seguindo característica de seu perfil navegacional são usados na tomada de decisão sobre que anuncio apresentar para cada um dos usuários desses sistemas. Bom, é oportuno, no entanto, indagar o que é Big Data e, principalmente, como essa abordagem afeta o modelo de desenvolvimento de aplicações que pretende extrair conhecimento deste grande volume de dados? O grupo Gartner [1] caracterizou Big Data como um fenômeno ligado a produção de dados nas atividades em que aparecem 5 V's: Volume, Variedade, Velocidade, Valor, Veracidade. Ainda que todas essas características sejam de fato relevantes no contexto de Big Data, o volume de dados aparece como a que ganhou maior destaque inicialmente, principalmente pelo desafio que arquivos com volumes muito grandes impõem para a quantidade de memória principal disponível em um único computador. Além dos desafio impostos pelos 5V's, um ponto de suma importância a considerar no contexto de Big Data é o tipo de acesso aos dados. Em aplicações tradicionais, o modelo de processamento principal tem por alvo um indivíduo ou grupos de indivíduos. Em Big Data,informações sobre um indivíduo, em geral, tem pouca importância, que não seja para sua contribuição a um agregado de interesse. As aplicações que se debruçam sobre grandes volumes de dados estão interessadas em tendências, ou em padrões que emergem a partir do dado agregado. Neste contexto, os padrões de acesso envolvem a varredura de grandes partes dos dados, combinada com algum tipo de investigação específica complementar à informações disponível no arquivo de base da exploração. Notem o termo *exploração* que denota o tipo de experiência de acesso em dados neste contexto.

Dessa forma, neste curso, vamos considerar a seguinte definição para Big Data:

Definition 5.1.1. Big data é o desafio imposto para se processar grandes arquivos, cujo tamanho extrapola em muito a quantidade de memória disponível em uma única máquina. Além disso, consideramos que o processamento de dados em grandes volumes se interessa à características que emergem ao se varrer uma boa parte de seus dados.

Uma vez definida a classe de problemas que nos é de interesse, podemos enriquecer nosso entendimento do problema avaliando elementos relevantes para o desenvolvimento de aplicações que respondam ao desafio Big Data. Claramente, um primeiro ponto

[1] www.gartner.com

seria avaliar a arquitetura computacional a abrigar o ambiente de processamento. A premissa de que os dados a serem processados não cabem na memória principal de um único computador se baseia no fato de que o crescimento do volume de dados disponível para exploração não segue a lei de Moore [Moore, 2000] e, ainda que tenhamos computadores cada vez com maior volume de memória a ser alocado, o conjunto de dados disponível tende a superá-lo. Neste sentido, o grupo da Google que propôs o arcabouço de software baseado no modelo *MapReduce* (MR), considerou aplicações implementadas sobre um cluster com até milhares de máquinas simples [Dean and Ghemawat, 2008]. Apesar da maioria das aplicações não utilizar um número tão grande de máquinas, a possibilidade de escalar o processamento sobre a quantidade de nós disponíveis é bastante atraente, sem contar com a alternativa de uso da nuvem com alocação de recursos de grandes provedores. Desta forma, a partir do arcabouço Apache Hadoop, considerou-se a plataforma de cluster de servidores sem compartilhamento como o padrão de fato para processamento Big Data. Dois outros elementos principais, entre vários componentes do que se passou a chamar de Ecosistema Big Data, merecem realce neste momento. O primeiro diz respeito ao gerenciamento dos arquivos por entre as máquinas componentes do cluster. Os arcabouços MR [2] priorizam a execução dos procedimentos da aplicação de forma a minimizar a transferência de dados, optando se possível pela alocação do procedimento no mesmo nó em que os dados estejam distribuídos. Para esse fim, foi projetado o sistema de arquivos distribuídos *Google File System* (GFS) [Ghemawat et al., 2003], posteriormente implementado no arcabouço Hadoop como *Haddop File System (HDFS)*.Dessa forma, o modelo de programação considera arquivos distribuídos por nós de um cluster e processos alocados a estes nós. A escalabilidade da estratégia para tamanhos crescentes de arquivos é proporcionada pela divisão do arquivo pelos nós de processamento, considerando-se que os problemas sendo tratados possam ser resolvidos de forma paralela e distribuída. O segundo elemento dessa abordagem esta ligado a extensibilidade do arcabouço para diferentes problemas que compartilhem a característica de independência de processos paralelos. Este aspecto do problema é tratado pela adoção do modelo de programação funcional tendo por base as funções de *Map* e *Reduce*. Neste modelo, conforme veremos na seção 5.4, o usuário fornece procedimentos da aplicação modelados para processar cada item do conjunto de entrada (Map), e um conjunto de elementos da entrada (Reduce). A partir dessa simplificação, uma grande gama de problemas pode ser resolvida fornecendo-se o comportamento para as respectivas funções.

Neste contexto, definimos o tema geral deste curso. Dada um ambiente de processamento de grandes volumes de dados, principalmente norteado por soluções com arcabouços MR, especificam-se algoritmos que se adequam ao modelo de programação subjacente e forneçam soluções eficientes para a implementação de aplicações.

O restante deste curso encontra-se dividido da seguinte forma. Na seção 5.2 apresentamos um exemplo de aplicação no domínio da astronomia que servirá de base para nossas discussões. Em seguida, na seção 5.3 discutimos o ambiente para arcabouços MR, incluindo: a arquitetura , o ecosistema MR, e seus principais módulos tomando como base o arcabouço Apache Spark. Na seção 5.4, apresentamos a API de programação com suas funções principais. Uma vez apresentado o ambiente de execução e a API para especifi-

[2]Usaremos o termo *arcabouços MR* para nos referirmos aos arcabouços que implementam o modelo MapReduce

Figure 5.1. Dilúvio de Dados na Midia

cação de aplicações, passamos a discutir algoritmos de apoio e sua implicação no modelo MR. Neste sentido, a seção 5.5 discute estratégias para o particionamento dos dados pelos nós do cluster. Em seguida, na seção 5.6 discutimos as estruturas de indexação QuadTree e PH-Tree e KD-Tree, de apoio para consultas de intervalos e vizinhança. A seção 5.7 discute o tratamento de redução de dimensionalidade. Finalmente, a seção 5.8 trata de algoritmos que otimizam as funções de agregação. A seção 5.9 tesse alguns comentários finais.

5.2. Exemplo de Aplicação

Nesta seção, descrevemos uma aplicação da astronomia, *Constellation Queries* (CQ) [Porto et al., 2017], cujo objetivo é varrer grandes arquivos contendo objetos estelares identificados por telescópios em levantamentos astronômicos, tal como o SDSS [SDSS,], buscando por ocorrência do fenômeno conhecido como lentes gravitacionais [3]. A aplicação busca por padrões geométricos definidos a partir da composição de corpos estelares. A Einstein cross é uma dessas formas representantes do fenômeno de lentes gravitacionais. Seu padrão geométrico é composto por quatro corpos estelares, veja na Figura 5.3. Consideram-se soluções candidatas todas as composições cujas formas se assemelhem à forma geométrica fornecida e que guardem com esta uma relação de escala. Esta última

[3]https://www.cfa.harvard.edu/castles/

é função da razão entre as distâncias entre dois elementos correspondentes, entre a forma fornecida como padrão e a forma candidata. Neste sentido, o *dataflow* CQ implementa o processo de busca por lentes gravitacionais. Sua representação gráfica aparece na Figura 5.2

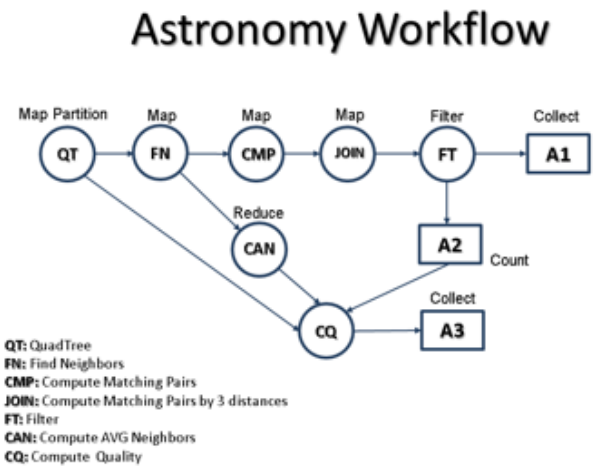

Figure 5.2. Dataflow - Constellation Queries

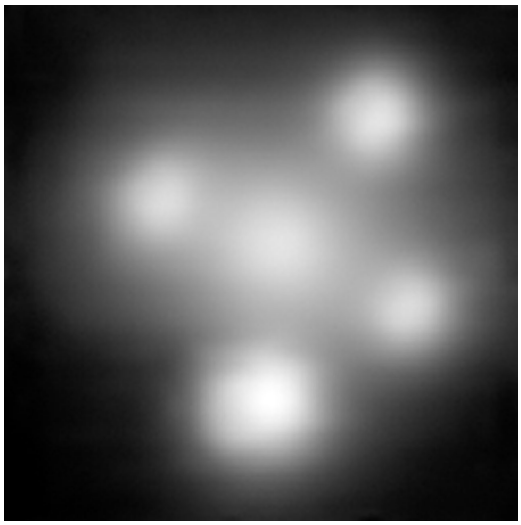

Figure 5.3. Einstein Cross

[Eigenbrod et al., 2008]

Ao longo do texto, detalheremos o dataflow *CQ*. Por hora, é suficiente perceber

que o dataflow define uma estrutura em grafo direcionado. Os nós do grafo correspondem a operações de transformação de dados e as arestas estabelecem uma relação produtor-consumidor. Os elementos quadrangulares são operações de produção de resultados finais.

5.3. Arquitetura

Arcabouços MR foram projetados em conformidade com arquitetura de paralelismo de dados. A Figura 5.4 [Ozsu and Valduriez, 1990] apresenta os principais modelos arquiteturais de clusters para processamento paralelo. Na Figura 5.4 (a), ilustra-se a arquitetura de memória compartilhada. Neste modelo de arquitetura, nós de computação possuem CPU e disco e compartilham uma área de memória obtida pela composição de áreas das máquinas envolvidas. Um exemplo importante deste tipo de arquitetura é o Módulo *fatnode* do sistema de super-computação Santos-Dumont. Esta arquitetura privilegia programas desenvolvidos para perceberem uma grande área de memória global com mecanismos de controle de acesso concorrente entre os diversos processos paralelos em execução. Em seguida, na ilustração (b) apresenta-se a arquitetura de disco compartilhado. Esta é a arquitetura paralela mais comum entre *clusters*. Basicamente, cada nó possui sua própria área de memória e CPU, porém compartilha o acesso aos dados com todos os outros nós através de um sistema de arquivo distribuído, como, por exemplo, o Lustre [4]. Todo acesso a dados requer a transferência pela rede entre o nó em que o arquivo se encontra e o solicitante. Usualmente, instala-se mecanismos de alta-velocidade entre os nós de processamento. Por último, na arquitetura ilustrada em (c), encontramos a arquitetura *sem compartilhamento*. Este é o modelo adotado por arcabouços MR. O sistema de arquivos distribuído oferece uma visão integrada de arquivos particionados e distribuídos pelos nós do *cluster*. Um módulo centralizado pode tomar conta dos fragmentos armazenados pelos vários nós de processamento. Adicionalmente, escalonadores de execução de jobs, como o *YARN*, procuram alocar os programas em nós onde haja partição do arquivo de entrada. Assim arquivos grandes possuem número de partições (veja seção 5.5) bem superior ao número de nós de processamento, fazendo com que os nós realizem trabalho durante todo o tempo de processamento do arquivo. Cada processo ativado, acessará o bloco local do arquivo de entrada e carregá-lo-a em sua área de memória local para processamento pelo diferentes núcleos da máquina.

5.3.1. O Sistema de Arquivos HDFS

O *Hadoop File System* (HDFS) é o sistema de arquivos distribuídos utilizado pela maioria dos arcabouços MR atuais, incluindo Apache Hadoop, Apache Spark e Apache Flink. O sistema foi projetado para aplicações de grandes volumes de dados priorizando: distribuição de dados e com isso, influenciando paralelismo de tarefas; tolerância à falhas através de réplicas de arquivos; e impedimento de atualização, eliminando controle de concorrência.

O sistema adota o modelo (grava-um-lê-muitos) de forma que um arquivo HDFS não sofre atualizações, uma vez que tenha sido criado, podendo ser lido múltiplas vezes. Veremos, quando discutirmos o sistema Spark, que arquivos do HDFS são lidos por trans-

[4] lustre.org

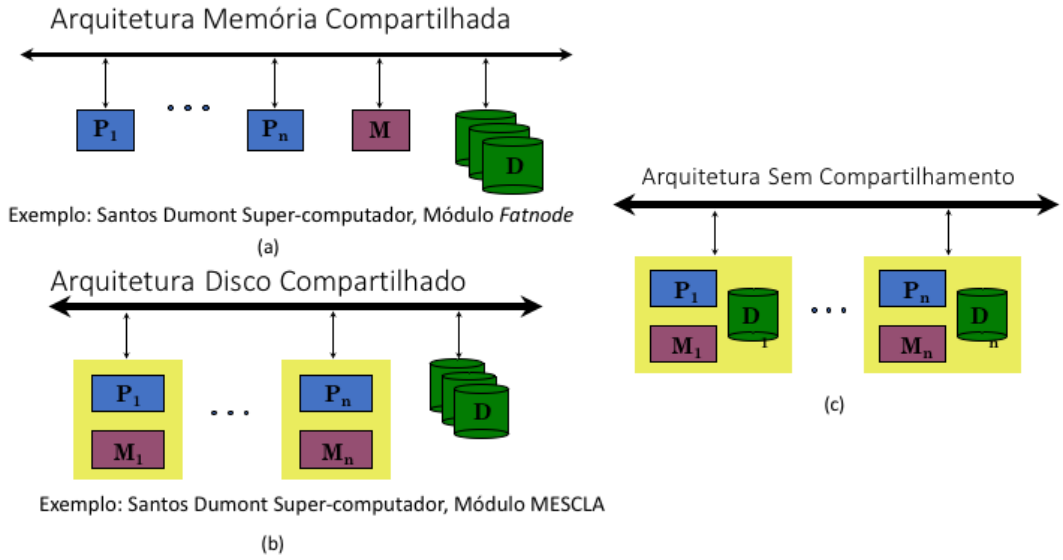

Figure 5.4. Arquitetura para Processamento Paralelo

formações que os processam e produzem novos arquivos, em memória ou em disco. Desta forma, o modelo não considera processos de atualização dos arquivos existentes.

O sistema está estruturado por um módulo servindo de catálogo geral *Nó de Nomes*, executando no nó computacional Mestre, e um módulo presente em cada um dos nós escravos, chamado *Nó de Dados*, ver Figura 5.5. Um arquivo no sistema HDFS é referenciado pelo seu caminho e nome individual. O sistema de caminhos obedece a uma estrutura hierárquica, comum a maioria dos sistemas de arquivos.

Um arquivo no HDFS é particionado em blocos de tamanho fixo típico de 64MB ou 128 MB. Blocos de arquivos são distribuídos pelos nós do cluster. Blocos em um nó são gerenciados pelo respectivo *Nó de nomes*. Novamente referindo-se ao processamento, blocos do HDFS são a unidade de processamento de tarefas no Spark. Sua distribuição pelos nós do cluster dirigem a execução paralela de processos que consomem blocos dos arquivos.

Adicionalmente, o sistema HDFS permite a réplica de blocos por nós do cluster. Inicialmente, o sistema é configurado para que cada bloco tenha duas réplicas, totalizando 3 réplicas por bloco alocadas em diferentes nós. Quando um nó falha durante a execução de um job, o *Nó de Nomes* informa a localização de réplicas dos blocos alocados ao nó apresentando problemas, permitindo que o escalonador re-inicie a tarefa em andamento em um dos nós contendo réplicas.

Finalmente, o HDFS é o componente fundamental no ecosistema de arcabouços MR. O sistema foi desenvolvido em Java, permitindo fácil integração com aplicações nas linguagens Scala, Java e com conectores para Python. As aplicações desenvolvidas utilizando-se de APIs nestas linguagens podem ser integradas aos arcabouços, permitindo a paralelização de suas tarefas, segundo o modelo MR.

[6]https://www.ibm.com/developerworks/br/library/waintrohdfs/

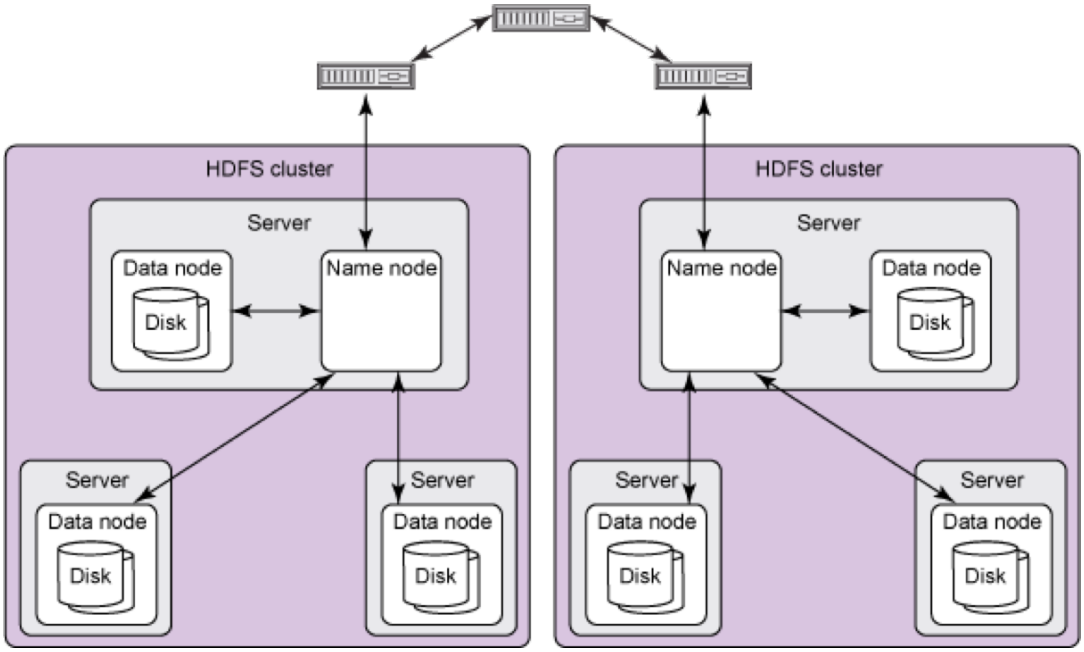

Figure 5.5. Arquitetura do Sistema HDFS - Fonte: IBM [6]

5.3.2. Utilizando o sistema HDFS

O *HDFS* pode ser utilizado como um sistema de arquivo adaptado a partir do protocolo *POSIX*. Todos os comandos são executados a partir do script *Hadoop*. Assim, pode-se executar:

```
hadoop [comando] [opçoes-genericas] [opçoes-comando]

hadoop fs -ls /user/hadoop/file1

hadoop fs -mkdir /user/hadoop/dir1

hadoop fs -cp /user/hadoop/file1 /user/hadoop/file2

hadoop fs -tail /user/hadoop/file1
```

5.4. O modelo de Programação MapReduce

Os arcabouços MR, como o Apache hadoop e o Apache Spark, foram desenvolvidos considerando a arquitetura paralela *sem-compartilhamento* conforme discutido em 5.3. Neste sentido, os sistemas utilizam a distribuição de dados provida por um sistema de arquivos distribuídos, como o HDFS. A distribuição de blocos de dados pelos nós do cluster direciona a alocação dos processos da aplicação para os mesmos nós. A este princípio chamamos de *Localidade de dados*.

Table 5.1. Comandos HDFS

Comando	Descrição	Exemplo
fs	realiza operações sobre os arquivos no HDFS	hadoop fs -cp /user/hadoop/file1 /user/hadoop/file2
put	copia arquivos para o HDFS do sistema de arquivos local	hadoop fs -put localfile1 localfile2 /user/hadoop/hadoopdir
copytolocal	copia arquivos do HDFS para o sistema de arquivos local	hadoop fs -copytolocal /user/hadoop/file localfile

Definition 5.4.1. *Localidade de Dados* é uma estratégia de processamento de dados distribuídos em que os processos são alocados, prioritariamente, nos nós em que os dados se encontram, minimizando a transferência de dados entre nós.

Precisamos, no entanto, entender que tipo de processos estão envolvidos em processamento de grande volumes de dados. Na seção 5.2, apresentamos um dataflow representando a aplicação de busca por lentes gravitacionais. Podemos iniciar a discussão sobre modelos de processamento de grandes volumes de dados comparando: *Workflow Científicos*, *Dataflows* e *Consultas em Banco de dados*. Enquanto todos esses modelos tratam de processamento de grandes volumes de dados, suas diferenças são marcantes.

- Workflows Científicos - são conjuntos de programas escritos em linguagens diversas e de forma autônoma, colocados em um mesmo processo de forma a definirem uma relação de produtor-consumidor de dados. A heterogeneidade de dados e o uso de programas intensivos de CPU, levam a implementações que não favorecem necessariamente a localidade de dados [Ogasawara et al., 2013, Ludäscher et al., 2006].
- Dataflows - são processos que implementam uma aplicação e são desenvolvidos em uma mesma linguagem de programação, de forma funcional, estabelecendo relação de produtor-consumidor entre as funções através de arquivos. O processamento é considerado intensivo de dados.
- Consulta de banco de dados - é a expressão de interesse em um subconjunto do banco de dados cujas operações são conhecidas e compõem a álgebra do modelo de dados. Uma consulta de banco de dados é tradicionalmente traduzida para um plano de consulta composto por operadores e uma relação de produtor-consumidor entre eles. No entanto, diferentemente das anteriores, as operações têm semântica conhecida.

Neste sentido, em arcabouços MR, uma aplicação é especificada como um dataflow, podendo ser representado por um grafo $D = (O, E)$, onde O é um conjunto de operações e E é um subconjunto de $O \times O$ definindo a relação de produtor-consumidor entre os elementos de O. Cada operação $o \in O$ está associada a um regra de transformação de dados, definida na linguagem de especificação do dataflow.

Definition 5.4.2. Transformações são funções $F = \{f_1, f_2, \ldots, f_n\}$ que recebem um ou mais arquivos de entrada e produzem um ou mais arquivos de saída. O comportamento de uma função $f_i \in F$ é conhecido e estabelece uma relação entre cada elemento do arquivo de entrada e elementos do arquivo de saída.

Tendo introduzido esses conceitos, podemos agora discutir sua implementação no arcabouço Apache Spark.

5.4.1. Apache Spark

O Apache Spark é um arcabouço MR escrito na linguagem Scala e descrito na tese de Doutorado de Matei Zaharia e apresentado em [Zaharia et al., 2010]. O arcabouço é integrado a várias fontes de dados, incluindo o *HDFS*. Seu modelo de programação baseado em *Localidade de Dados* é bastante semelhante ao do Hadoop porém otimiza o processamento de um *dataflow* mantendo os arquivos intermediários em memória. A motivação principal é que *dataflows* apresentam sequências de operações que podem ser executadas, uma após a outra, em um mesmo nó de processamento, evitando a transferência de arquivos entre nós. Neste cenário, a manutenção dos arquivos intermediários entre funções na memória principal pode levar a ordens de magnitude de ganho em tempo de processamento. Em *Spark* operações são classificadas em dois tipos: transformações e ações. As transformações recebem um ou mais *RDDs* de entrada e produzem um *RDD* de saída. Já uma *ação* recebe um *RDD* na entrada e produz um valor de tipo primitivo, por exemplo um inteiro para a ação de *count()*.

```
# Map transformando um RDD "data" com textos em linhas,
# separando por tabs e gerando um novo RDD "reviews"

reviews = data.map(lambda x: x.split("\tablename") )
```

Retomando o dataflow da figura 5.2, temos a seguinte divisão das operações do dataflow: Transformações = $< QT, FN, CMP, JOIN, FT >$, Ações: $< A_1, A_2, A_3, CAN, CQ >$

Spark processa *dataflows* através de uma estratégia conhecida como *avaliação tardia* (AT). Uma vez que precisa identificar os trechos do dataflow sujeitos a execução em uma mesmo fluxo, é necessário estabelecer as *fronteiras de localidade de dados*.

Definition 5.4.3. Uma fronteira de localidade de dados é um fragmento do dataflow que pode ser executado sem que haja necessidade de transferência de dados entre nós de um cluster. Em Spark, uma fronteira de localidade de dados define *estágios de processamento*.

Desta forma, o processo de *avaliação tardia* permite que otimizações sejam aplicadas sobre o fragmento do dataflow delimitado pela fronteira de localidade de dados. Um efeito secundário da AT é que transformações apenas são realizadas quando o sistema de avaliação encontra uma ação, fechando a fronteira relativa aquele fragmento.

5.4.2. Resilient Distributed Datasets (RDD)

O arcabouço *Spark* estendeu o conceito de *localidade de dados* proporcionada pelo *HDFS* de forma a otimizar a comunicação entre operações que estabeleçam um fluxo de processamento no modelo produtor-consumidor.

Um RDD é uma estrutura de dados distribuída. Recuperando o modelo de distribuição de dados do *HDFS*, o RDD segue uma estratégia semelhante com distribuição pela memória principal dos nós de um cluster. Assim, um RDD corresponde a um conjunto de objetos distribuídos e alocados em unidades de blocos pela área de memória disponível no cluster. Em Spark, blocos recebem o nome de *partições*, sendo essencialmente o mesmo conceito que blocos em HDFS, apesar de a princípio não serem persistentes. Adicionalmente, RDDs são não atualizáveis. A partir de um RDD podem-se aplicar transformações, gerando um novo RDD, ou ações, veja Figura 5.6. Estas últimas retornam o comando e os dados para o nós mestre.

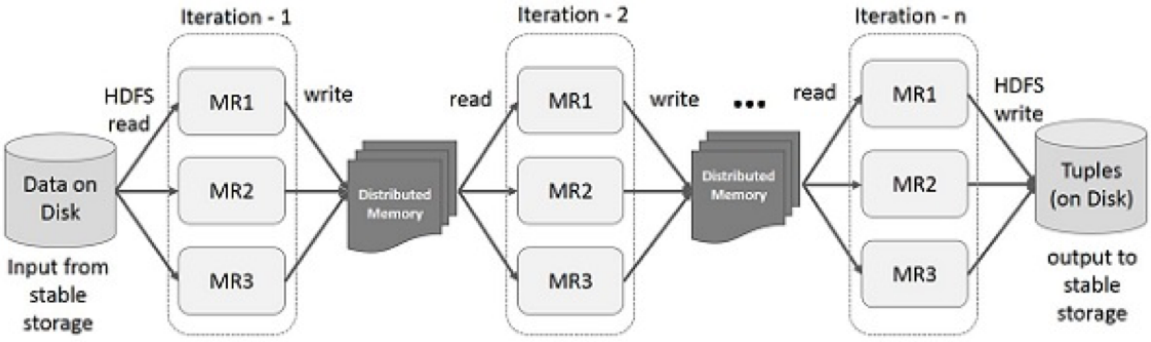

Figure 5.6. RDD em Memória

Transformações em Spark operam sobre RDDs, recebendo-os na entrada e produzindo-os na saída. RDD é um tipo de dado em Spark que além de sua estruturação em blocos contendo objetos, definidos em Java, Scala ou Python, itera sobre o conjunto de forma implícita. Assim uma transformação ou ação em *Spark* agindo sobre um RDD é invocada tantas vezes quantos forem os objetos alocados ao RDD. No código abaixo, um dataflow especifica um processo em que dois RDDs são juntados e, em seguida, o resultado é agrupado por uma chave. O RDD de agrupamento é filtrado por uma condição e o número de objetos no RDD filtrado é retornado. Notem a estrutura de fluxo de processamento explicitamente representados através do operador ".".

```
rdd1.join(rdd2)
     .groupby(..)
     .filter(...)
     .count()
```

Retomando a questão da avaliação tardia, dois efeitos imediatos deste modelo de execução sobre RDDs são (1) re-execução de um fragmento para reconstrução de uma partição de um RDD. Dado que RDDs são, a princípio, mantidos em memória, em um dataflow com mais de uma ação e que tenha uma dependência de dados entre RDDs usados no fragmento de uma ação e funções no fragmento de ações posteriores, Spark pode reconstruir o RDD inicial. Para isso, Spark mantém proveniência de construção associada a cada partição de um RDD (2).

No dataflow exemplo da figura 5.2, temos três ações: A_1, A_2, A_3. Considerando um escalonamento parcial: $< QT, FN, CMP, JOIN, FT, A1, CAN, CQ, A_2 >$, devido a AT,

o RDD produzido por FT será reconstruído para ser consumido por CQ quando da ação A_2, apesar de FT já ter sido construído quando da ação A_1.

5.4.3. Funções Spark

Como discutido, Spark oferece um conjunto de funções de segunda ordem como API para interface com as linguagens de programação Java, Python e Scala. As funções são ditas de segunda ordem pois invocam as implementações do usuário que fornecem o comportamento de primeira ordem a ser aplicado aos objetos dos RDDs. Assim, as funções de segunda ordem definem o modelo de transformação ou ação a ser aplicado enquanto a de primeira ordem fornece a implementação para tal operação.

Neste contexto a API Spark inclui as transformações conforme aparecem resumidamente na tabela 5.2:

Table 5.2. Transformações

Comando	Descrição
map(func)	transforma um RDD_i em um RDD_j
filter(func)	retorna um RDD em que os objetos atendem à condição imposta por *func*
flatmap(func)	produz um RDD em que o processamento de cada objeto do RDD de entrada pode gerar 0 ou mais objetos no RDD de saída
mappartition(func)	processa todos os objetos da partição do RDD na função *func*. Permite que se mantenha um estado de processamento entre o processamento de objetos do RDD de entrada.
union(RDD)	executa um operação binária em que o RDD de entrada é unido ao RDD passado como parâmetro.
groupbykey()	opera sobre um RDD do tipo (chave, valor) agregando pelos valores de uma mesma chave e produzindo um RDD <chave, <listavalores>

Igualmente, podemos listar as seguintes ações típicas, conforme a tabela 5.3.

Exemplo de código com a ação *Reduce* 5.7:

5.5. Algoritmos de Particionamento

Nesta seção discutiremos sobre algoritmos de particionamento de dados.

Definition 5.5.1. Dado um arquivo $D = \{d_1, d_2, ..., d_n\}$, onde d_i é um elemento do conjunto, e um cluster $N = \{n_1, n_2, ..., n_k\}$, onde n_j é um nó de um cluster, *Particionamento* é uma função $f : D \Rightarrow N$ que determina a alocação de um objeto em um nó de um cluster.

Como discutido em 5.4, os arcabouços MR se baseiam no critério de *localidade de dados* 5.4.1 como estratégia para processamento eficiente de grandes volumes de dados. O sistema HDFS 5.5, por exemplo, utiliza uma estratégia balanceada de distribuição de

Table 5.3. Ações em Spark - resumido

Comando	Descrição
reduce(func)	recebe uma função com dois parâmetros. O primeiro é um acumulador e o segundo é a variável cujo valor será acumulado durante a iteração pela partição
collect()	recupera todo o conteúdo do RDD nas diversas partições retornando ao mestre como uma lista
count()	retorna o número de entradas do RDD
take(n)	retorna uma lista com as *n* primeiras entradas do RDD
saveAsTextFile(path)	retorna um arquivo texto com o conteúdo do RDD no sistema de arquivo local ou HDFS

```
# reduce numbers 1 to 10 by adding them up
>>> x = sc.parallelize([1,2,3,4,5,6,7,8,9,10], 2)
>>> cSum = x.reduce(lambda accum, n: accum + n)
>>> print(cSum)
55

# reduce numbers 1 to 10 by multiplying them
>>> cMul = x.reduce(lambda accum, n: accum * n)
>>> print(cMul)
3628800

# by defining a lambda reduce function
>>> def cumulativeSum(accum, n):
...     return accum + n
...
>>> cSum = x.reduce(cumulativeSum)
>>> print(cSum)
55
```

Figure 5.7. Ação Reduce em Spark-Python

dados em que cada nó recebe blocos de mesmo tamanho. Da mesma forma, as partições

em memória utilizadas em *Spark* dividem os arquivos em unidades também de mesmo tamanho físico e as distribui, usando uma função de *hashing* como particionamento. Em [Oliveira et al., 2015], mostramos que quando o critério de particionamento atende à características de processamento dos dados, há um ganho significativo de processamento. Considere por exemplo, a aplicação exemplo 5.2. A formação de soluções se baseia em objetos localizados a uma certa distância uns dos outros, conforme estipulado pelo padrão geométrico fornecido. Neste contexto, não faz sentido buscar por vizinhos que estejam a uma distância além da maior distância no padrão geométrico sendo procurado. Essa restrição, que aparece em muitas aplicações com critérios de vizinhança, permite que os dados sejam particionados pelos nós de um cluster seguindo uma orientação de vizinhança, por exemplo, mantendo objetos com uma distância de até um ε em uma mesma partição, veja por exemplo a Figura 5.8.

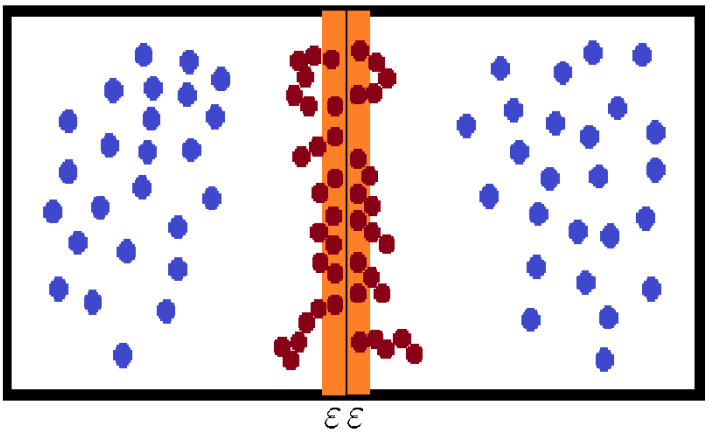

Figure 5.8. Fronteira em Partição

[7][Pires, 2016]

Na Figura 5.8, vê se uma região do espaço 2D, representada pela retângulo da figura com uma partição separando-a em duas regiões, não necessariamente de mesmo tamanho. Os elementos em azul são aqueles fora da zona fronteiriça. Já os elementos em roxo estão na zona de fronteira ou são efetados por ela. Neste caso, utilizarmos a fronteira delineada como critério espacial para o particionamento, precisaríamos incluir em cada região os objetos da região vizinha localizados na área de fronteira, (i.e. objetos roxos na figura 5.8).

5.5.1. O Algoritmo FRANCE

Em [Gaspar and Porto, 2014], propõe-se o *FRANCE*(FRAgmeNtador de Catálogos Espaciais, um algoritmo iterativo para particionar dados em histogramas *equi-depth*. Formalizamos o particionamento, segundo o FRANCE conforme a definição 1.

Definition 5.5.2. (Particionamento) Dado um objeto $o_{ji} < x, y, a_1, a_2, ..., a_t >$ tal que a_z, $1 \leq z \leq t$, é um atributo de o_{ji}, e $x \in X$ e $y \in Y$, tal que X e Y são as dimensões espaciais; um particionamento P é uma lista de valores $< v_1, v_2, ..., v_g >$ tal que $v_h \in x$ e $1 \leq h \leq g$. Desta forma, uma partição P_r, tal que $1 \leq r \leq g-2$ apresenta objetos vizinhos em uma região do espaço delimitada por P_r e P_{r+1}.

Considerando um contexto de dados espaciais 2D, o algoritmo calcula iterativamente os pontos de fragmentação segundo uma das dimensões, se preocupando em manter as partições com uma quantidade equivalente de objetos (com tolerância δ).

O *France* assume que se conhece o tamanho de arquivo e o número de partições desejado. Estas últimas são geradas iterativamente até que a quantidade de elementos seja próxima em δ do balanceamento perfeito. Consideramos perfeito, balanceamentos em que as partições apresentam n/g elementos, onde n é o total de elementos e g a quantidade de partições.

Primeiramente, divide-se, em uma das dimensões, a área espacial coberta pelo *dataset* em duas partições, cada uma cobrindo metade dos objetos. Uma variável *diff* é definida como metade do tamanho de cada partição. Para cada partição, enquanto não atingir um particionamento com n/g objetos (com uma tolerância de δ definida a priori), ela será reduzida por $diff$. Para cada passo, $diff = diff/2$. Quando a partição tiver a quantidade de objetos dentro do limite de δ, ela será fixada. O algoritmo executa recursivamente no espaço disponível entre as partições fixas.

O valor de δ é alterável no código e, tem como valor *default*, $0,5\%$ da quantidade de objetos desejada. Logo, todas as partições geradas terão $(n/g) \pm 0,005 \times (n/g)$ elementos. Com esse δ em 0,5% temos uma variação muito pequena no tamanho das partições.

O FRANCE gera como saída os valores da coordenada x onde ocorreram as divisões do espaço. A partir desses valores, podemos definir formalmente uma fronteira através da Definição 5.5.3.

Definition 5.5.3. Uma fronteira f_i definida segundo um valor de particionamento v_i, pertencente ao conjunto de partições P, na dimensão D, contem um conjunto de objetos O_{f_i} cujos valores da coordenada x estão entre $v_i - \varepsilon \leq x \leq v_i + \varepsilon$.

O pseudocódigo é apresentado no Algoritmo 1 e o seu passo a passo é ilustrado na Figura 5.9, onde os retângulos vermelhos correspondem às partições com uma quantidade de elementos maior que $n/x + \delta$ e que ainda precisam ser divididas para conter o número de objetos ideal. Os retângulos verdes representam as partições que contêm a quantidade de objetos dentro do limite de $n/x + \delta$ e não precisam mais ser subdivididas, ou seja, elas são partições fixadas. Para maiores detalhes, a implementação do FRANCE que fizemos em java está disponível à comunidade na internet[8].

5.5.2. Particionamento de grafos. *HDRF: Stream-Based Partitioning for Power-Law Graphs*.

Nos últimos anos, ao passo que ciência e tecnologia evoluem, nossa sociedade tem se deparado cada vez mais com a geração de grande quantidade de dados. Mais ainda, parte relevante desses dados é disposta estruturalmente como redes de larga escala. Além da grande dimensão que essas redes, representadas por grafos, possuem, sua presença se dá nos mais diversos domínios do conhecimento, tais como biologia, ciência da computação, química e sociologia [Lovász et al., 2009]. Nesse contexto, pode-se citar as redes sociais

[8] http://github.com/vinipires/FRANCE

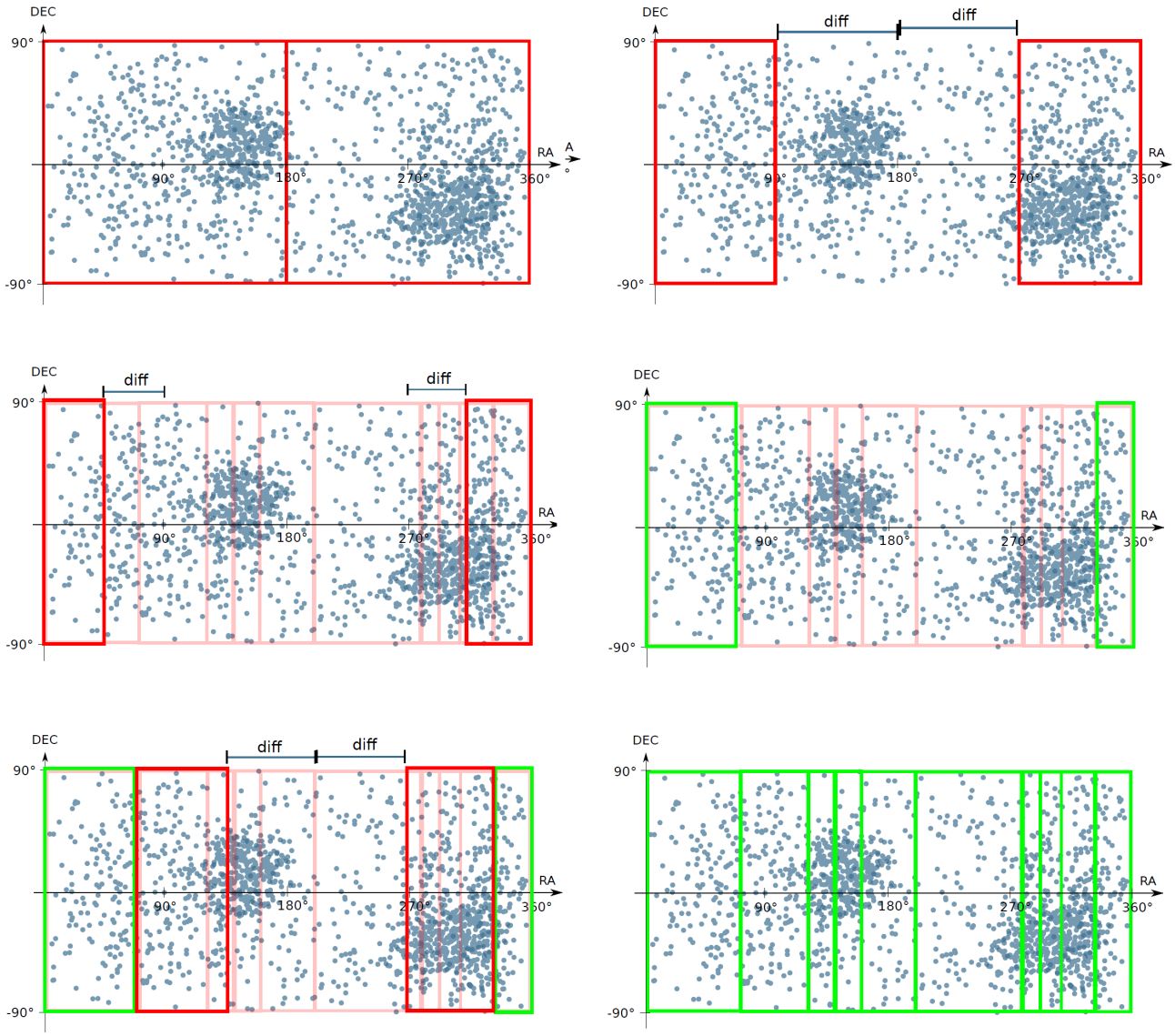

Figure 5.9. Passo a passo do FRANCE

online, como o *Facebook* que conta atualmente com cerca de 2 bilhões de usuários mensalmente ativos, cada qual com, em média, 155 ligações de amizade [Facebook, 2017]. Dada a relevância dessas redes, sua larga escala e usualmente padrões de conexão não triviais, suas análises demandam ferramental computacional compatível com o processamento em ambientes distribuídos e paralelos. Com isso, uma gama de plataformas de *software* projetadas para esse tipo de ambiente tem surgido recentemente [Guo et al., 2014].

Com a necessidade do uso de ambientes distribuídos e paralelos na análise de grafos de larga escala, um dos pré-requisitos para execução eficiente, dos algoritmos envolvidos é o particionamento dos grafos [Verma et al., 2017]. Intuitivamente, se quer particionar o grafo de modo que haja poucas arestas entre as partições, a fim de que o custo de comunicação seja reduzido. Aliás, ao processar um grafo em paralelo em k elementos

Algorithm 1 FRANCE: Algoritmo de Particionamento Espacial

Entrada 1 : *Arquivo de Entrada*
Entrada 2 : Quantidade de partições
function FRANCE(dataset,x) ▷ x é a quantidade partições desejada
    $objetopparticao \leftarrow \frac{|dataset|}{x}$
    startL=endL;startR=endR
    diffL=endL-startL
    diffR=endR=startR
    QTObjeL= computeQtdObj(StartL, endL)
    QTObjeR= computeQtdObj(endR, startR)
    for all $partition \in partitions$ **do**
        while $objetopparticao \notin QTObjeL \pm \delta \vee objetopparticao \notin QTObjeR \pm \delta$ objetos **do**
            Partição é reduzida/aumentada por $diff[L|R]$
            Update $diff[L|R]$
            Update endR, endL
        end while
        $Store\ part[partition] \leftarrow endL$
        $Store\ part[x - partition] \leftarrow endR$
    end for
    return part
end function
Saída: fronteiras;

de processamento ou em n nós computacionais, se deseja que o grafo seja particionado em k ou n partições de relativamente mesmo tamanho, de modo que seja minimizado o desbalanceamento de carga. Desse modo, haja vista a relevância das aplicações envolvendo redes de larga escala, assim como as dificuldades inerentes de seu particionamento pelo recurso computacional, técnicas destinadas à divisão de grafos em larga escala no âmbito de computação distribuída e paralela têm sido tópico de desenvolvimento e pesquisa. Nesse cenário, pode-se situar essas técnicas em duas classes [Gonzalez et al., 2012]: *corte em aresta* e *corte em vértice*, as quais são discutidas a seguir.

De modo geral, nas abordagens de corte em aresta, atribui-se os vértices às partições, enquanto as arestas são possivelmente estendidas pelas divisões. Enquanto isso, nas técnicas de corte em vértices, as arestas são atribuídas às partições, ao passo que os vértices são possivelmente estendidos entre as divisões. Vale atentar que em ambas estratégias, a extensão de arestas ou vértices pelas partições contribui no sobre-custo de armazenamento, comunicação e sincronização, posto que a informação da adjacência (corte em aresta) deve ser mantida em cada partição, e vértices devem ser replicados pelas divisões (corte em vértice). Como resultado de suas características, essas duas estratégias correlacionam-se com o grafo de modo distinto [Verma et al., 2017]. Técnicas de corte em aresta são preferíveis para grafos em que a maioria dos vértices seja de grau baixo, já que todas as arestas adjacentes a determinado vértice são alocadas na mesma partição. Entretanto, para grafos cuja distribuição de grau segue uma lei de potência [Barabási and Albert, 1999], nos quais estão presentes vértices com grau muito acima da média, estratégias baseadas em corte em vértice podem ser mais indicadas, uma vez que permitem melhor balanceamento de carga ao distribuir a carga desses vértices de grau elevado por várias partições.

Apesar da aplicabilidade das técnicas baseadas em corte em aresta, com a pro-

fusão de redes reais (grafos) cuja distribuição de grau segue uma lei de potência em diversas áreas do conhecimento [Barabási and Albert, 1999], abordagens de particionamento baseadas em corte em vértice vem ganhando notoriedade e sendo aplicadas em sistemas de processamento distribuído de grafos [Petroni et al., 2015]. Contudo, como esse tipo de estratégia lança mão à replicação de vértices, a sincronização entre as réplicas pode dificultar o desempenho computacional eficiente, haja vista a necessidade de coordenação e troca de dados entre os vértices replicados várias vezes no decorrer da execução dos algoritmos de análise. Portanto, em vista a essas dificuldades, uma estratégia eficiente no particionamento de grafos que se baseie em corte em vértice precisa resolver o problema de minimização *balanced k-way vertex cut*, isto é, além de manter a carga de trabalho por partição p o mais próximo possível do ótimo teórico $|E|/|P|$, deve-se minimizar o número de partições distintas $|A(v)|$ que cada vértice v foi atribuído:

$$\min \frac{1}{|V|} \sum_{v \in V} |A(v)| \quad \text{s.t} \quad \max_{p \in P} |p| < \sigma \frac{|E|}{|P|}$$

Diante do cenário exposto, [Petroni et al., 2015] propuseram um algoritmo guloso de particionamento de grafos com foco em *streaming - High degree (are) Replicated First* (HDRF) - para resolução do problema apresentado de *balanced k-way vertex cut*. Como o algoritmo HDRF é focado no processamento em *streaming*, sua entrada é um fluxo de arestas que forma o grafo a ser particionado, sendo realizada apenas uma passagem no grafo, isto é, dado que uma aresta já foi avaliada e atribuída a uma partição, o algoritmo posteriormente não altera de forma alguma essa atribuição. Além disso, HDRF, de forma gulosa, prioriza a replicação de vértices de grau relativamente mais alto. Como esses vértices de grau mais alto (*hubs*) servem muitas vezes como pontes entre subgrafos densos compostos por vértices de grau médio, ao priorizar a partição pelos *hubs* intrinsecamente almeja-se que as partições reflitam os agrupamentos observados no grafo sendo particionado. Desse modo, com base nas premissas apresentadas, o algoritmo funciona como a seguir.

Ao processar uma aresta e que conecta dois vértices v_i e v_j, inicialmente o algoritmo HDRF incrementa em um o grau parcial $\delta(v_i)$ e $\delta(v_j)$ desses vértices. Note que o grau parcial de um vértice é a quantidade de arestas a que ele pertence e que já foram processadas até o momento. Posteriormente, os graus de v_i e v_j são normalizados de modo que $\theta(v_i) = \delta(v_i)/[\delta(v_i) + \delta(v_j)] = 1 - \delta(v_j)$. Logo em seguida, o algoritmo HDRF calcula a pontuação C^{HDRF} para todas as partições $p \in P$, atribuindo portanto a aresta e a partição p^* que maximiza C^{HDRF}. Deve-se atentar que a pontuação é a combinação linear entre um fator de replicação, em que prioriza-se a cópia de *hubs* entre as partições, e um fator de balanceamento, no qual partições mais vazias são privilegiadas.

$$C^{HDRF}(v_i, v_j, p) = \overbrace{g(v_i, v_j, p)}^{\text{fator de replicação}} + \overbrace{\lambda \cdot \frac{\text{maxsize} - |p|}{\varepsilon + \text{maxsize} - \text{minsize}}}^{\text{fator de balanceamento}}$$
$$g(v_i, v_j, p) = p \in A(v_i) \cdot (2 - \theta(v_i)) + p \in A(v_j) \cdot (2 - \theta(v_j))$$

Vale notar ainda que alguns parâmetros devem ser informados ao algoritmo. O

tamanho máximo maxsize e mínimo minsize das partições. Um valor constante ε pequeno. O número de partições $|P|$. E por fim, o parâmetro λ, o qual permite o controle da influência do desbalanceamento de carga no tamanho da partição. Esse parâmetro é utilizado para lidar com problemas decorrentes de simetria e ordem na qual as arestas são processadas. O comportamento do algoritmo HDRF em relação a λ pode ser sintetizado nesses casos:

$$\begin{cases} \lambda = 0, & \text{agnóstico ao balanceamento de carga} \\ 0 < \lambda \leq 1, & \text{balanceamento utilizado para quebrar a simetria} \\ \lambda > 1, & \text{importância dada ao balanceamento proporcional a } \lambda \\ \lambda \to \infty & \text{atribuição aleatória das arestas} \end{cases}$$

Enfim, apesar da eficiência computacional demonstrada empiricamente em [Petroni et al., 2015], a necessidade de intervenção do usuário na parametrização de λ, a fim de lidar com particionamentos ineficientes decorrentes da ordem em que as arestas são processadas, pode ser considerada uma desvantagem e um tópico de investigação futura. Além disso, os autores deixam claro que implementações eficientes distribuídas e paralelas do algoritmo são trabalhos em aberto.

5.5.3. Uso do Algoritmo de Particionamento em Spark

Spark oferece dois métodos básicos de particionamento: *Hash* e *Range*.

A classe raiz de particionamento em Spark é a *Partitioner* que aparece especializada nas classes *HashPartitioner* e *RangePartitioner*. O *France* pode ser implementado sobre-escrevendo-se o método *getPartition(Object key)*.

5.6. Estruturas de Indexação

Nesta seção, trataremos do apoio ao processamento de grandes volumes através de estruturas de indexação. Em arcabouços MR, tal como o Spark, arquivos são distribuídos pelos nós de processamento, como vimos em 5.3.2. Apesar da aplicação típica envolver transformações aplicadas sobre RDDs, as funções de transformação podem se valer de uma estrutura de *lookup* para complementar a informação principal sendo extraída do RDD, ou ainda realizar a busca por vizinhos. Em nossa aplicação exemplo 5.2, a primeira tarefa constrói uma *Quadtree* [Samet, 2011] a ser usada para separação de estrelas em regiões e usar estas últimas como agregadores, veja seção 5.8. Cada região possui um centro geométrico se comportando como representante do conjunto de estrelas cobertos pelo seu quadrante espacial. Além disso, a estrutura pode responder a consultas buscando por vizinhos a uma distância indicada. De uma maneira geral, no processamento de grandes volumes de dados, estruturas de indexação podem ser utilizadas:

- como mecanismos de particionamento dos dados, permitindo paralelismo de processos. Este é o caso de índices baseados em *hashing* da chave de acesso.

5.6.1. Patricia-Hypercube Tree

Dados multidimensionais são comuns em várias aplicações, tais como simulações numéricas, sistemas de informação geográficas e na astronomia. Comumente, os dados multidimensionais se apresentam com um conjunto de dimensões espaço-temporais, embora possam existir outras dimensões dependendo do contexto da aplicação.

Existem diversas estruturas para indexação deste tipo de dados. Podemos citar como as mais predominantes, as R-Trees, KD-Trees e Quad-Trees. Porém, outra alternativa interessante, a PH-Tree (Patricia Hypercube Tree) foi apresentada mais recentemente [Zäschke et al., 2014]. A PH-Tree se baseia no conceito de árvores de prefixo binárias, que têm por objetivo reduzir o custo de armazenamento de dados. Neste tipo de árvore, fazemos uso do compartilhamento de prefixos comuns, diminuindo o tamanho total de armazenamento de dados multidimensionais. Além disso, a PH-Tree se baseia em cubos multidimensionais, permitindo assim uma navegação mais rápida pelos dados em comparação com outros tipos de árvores binárias.

A PH-Tree foi projetada com intuito inicial de servir como um método de indexação multidimensional para dados em memória. Embora, de acordo com os seus autores, no caso de conjuntos de dados com várias dimensões, a PH-Tree também seria adequada para indexação em disco, porque, neste caso os nós da árvore iriam conter uma quantidade grande o suficiente de registros para serem divididos de maneira eficiente em páginas do disco.

Uma PH-Tree é similar a uma Quadtree, embora utilize cubos multidimensionais e se baseie no conceito de compartilhamento de prefixos. A PH-Tree segue algumas filosofias caraterísticas. Ao contrário da abordagem da KD-Tree, em que apenas uma dimensão é particionada em cada nó, a PH-Tree segue o conceito implementado na Quatree, em que todas as dimensões são divididas em cada nó. Isto faz com que o acesso aos dados seja independente da ordem na qual as dimensões são armazenadas.

Outra característica da PH-Tree, é sua altura limitada e seu número de nós reduzido em comparação a outros tipos de árvores para dados multidimensionais. Isto ocorre porque cada nó da PH-Tree pode ter até 2^k filhos (sendo k o número de dimensões), e a altura máxima da árvore é igual ao número de bits do valor mais longo armazenado. Além disso, a PH-Tree é uma árvore inerentemente desbalanceada, e não requer algoritmos de rebalaceamento para operações de inserção e remoção de nós. Como a altura máxima da árvore é limitada, o problema da degeneração também fica limitado.

A estrutura interna da PH-Tree é determinada apenas pelo conjunto de dados presente nela. A ordem em que os dados são inseridos não é relevante na configuração final, isto é, se um mesmo conjunto de valores for inserido em uma ordem diferente, a estrutura final será a mesma. Operações de atualização (inserções e remoções) envolvem apenas a manipulação de no máximo dois nós, haja visto que a PH-Tree, pois nenhuma operação de balanceamento posterior é necessária.

A PH-Tree armazena entradas de dados, que são conjuntos de valores. Por exemplo, um ponto em 2D é armazenado como uma entrada com dois valores, cada um representando uma dimensão de um ponto. Os valores são armazenados em sua forma binária como uma cadeia de bits. Para árvores contendo dados com mais de uma dimensão, as

cadeias de bits são armazenadas em paralelo, como mostra a Fig. 5.10. No topo da árvore, temos um nó raiz com apenas uma única referência na segunda posição. A posição é calculada a partir do primeiro bit de cada um dos dois valores da entrada bi-dimensional. Usando esta abordagem, o vetor de posições com as referências se transforma em um cubo multidimensional. Abaixo do nó raiz, existe um prefixo para o sub-nó, consistindo de 0 para ambos os valores.

Cada nó da PH-Tree então contém um cubo multidimensional de referências. O tamanho desse vetor de referências é igual a 2^k. Cada posição desse vetor está relacionada ao endereço no cubo multidimensional, isto é, uma combinação de bits para cada dimensão. Associado a cada posição do vetor de referências, podemos ter um posfixo, indicando um nó folha. O posfixo é a parte final de um valor armazenado. Para nós intermediários, ao invés de um posfixo, temos um infixo e uma referência a outro cubo multidimensional/nó da PH-Tree.

Os bits entre um nó e seu nó pai são chamados de infixo. Chamamos de prefixo, todos os bits acima de um nó, formado pela concatenação dos infixos e endereços de cubos multidimensionais. Se concatenarmos o prefixo de um nó, com os bits do endereço do cubo multidimensional e o posfixo nós temos um conjunto de valores multidimensionais armazenados na arvóre.

Os cubos multidimensionais associados aos nós da PH-Tree podem ser esparsos, isso ocorre principalmente quando a árvore não tem entradas suficientes, com prefixos suficientemente variados. Neste caso podem-se adotar representações variadas para o vetor de referências baseado em um esquema de chave valor (chamado de linearized hypercube - LHC) ao invés do cubo multidimensional baseado em um vetor de referências (hypercube - HC). A Figura 5.10 mostra um exemplo de um PH-Tree de Exemplo contendo quatro entradas bi-dimensionais. São elas: (0001, 1000), (0011, 1000), (0011, 1010).

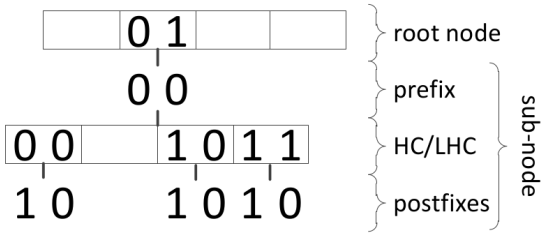

Figure 5.10. Exemplo de PH-Tree [1]

As PH-Trees suportam basicamente dois tipos de operações de consulta. As consultas pontuais em que buscamos pela existência de uma entrada específica. Esta busca é trivial, começando a partir do nó raiz, e através de comparações e buscas nos nós intermediários, é possível determinar a existência de uma entrada, avaliando apenas um subconjunto pequeno do número total de nós. De forma análoga é possível também executar as chamadas window queries, em que todos os valores dentro de uma janela multidimensional que determina um intervalo em cada dimensão devem ser retornados.

Testes comparativos mostraram que PH-Trees ocupam significativamente menos espaço que KD-Trees, e em alguns casos, até mesmo menos espaço que armazenamento

simples não indexado. Além disso, os autores demonstraram que a PH-Tree oferece um desempenho competitivo para atualizações, consultas pontuais e window queries. Sendo uma alternativa interessante para uma representação indexada dos dados com baixo custo de armazenamento.

5.6.2. Randomized KD-tree

O algoritmo KD-tree é uma popular estrutura de organização de dados multidimensionais, que tem por principal característica ser um algoritmo de simples implementação e com custo computacional pouco elevado, sendo utilizada em uma vasta gama de aplicações como processamento de imagens e localização espacial.

O procedimento de construção desta estrutura se inicia com a listagem das dimensões em uma dada sequência, em geral ordenada pelas dimensões de maior variância. Seguindo esta ordenação tem-se um processo de divisão dos dados, onde estes são fragmentados na mediana de uma dada dimensão dando origem a dois subconjuntos. A rotina então se repete de maneira recursiva para a próxima dimensão da lista ordenada, sendo particionado neste momento os subconjuntos gerados pelas divisões anteriores. Este processo se repete até que cada conjunto se remeta a um único elemento, tendo por fim uma árvore indicando as sucessivas divisões do espaço em dimensões alternadas. O Algoritmo 2 resume o processo.

Algorithm 2 Construção KD-tree(dataset $\in R^k$)

```
p=lista das dimensões ordenada pela variância
π(ROOT)=dataset
h ← 0
while árvore não construída do
    for i = 1...k do
        for cada nó x no nível h da árvore do
            medianNode=median(π(x),p(i))
            dividir π(x) em π(x_l) e π(x_r), com x_l(i) < medianNode e x_r ≥ medianNode, x_l ∈ π(x_l), x_r ∈ π(x_r)
        end for
        h++
    end for
end while
return KD-tree
```

Nesta estrutura a busca consiste em percorrer a árvore verificando as faixas de valores relativas a cada nó até se atingir um nó folha, conferindo se este registro atende a consulta especificada. Para caso de buscas por vizinhança é ainda necessário se verificar um subconjunto de nós adjacentes a folha encontrada, dado que este nó não é necessariamente o ponto mais próximo à coordenada da busca realizada.

Em se tratando destas buscas por vizinhança, o algoritmo pode apresentar algumas dificuldades em termos de desempenho, dado que a análise do subconjunto de pontos se trata de um procedimento recursivo e a quantidade de pontos verificada é proporcional ao número de dimensões do dado armazenado. Desta forma, a busca pode se tornar custosa quando se tratam de bases de elevada dimensão e um grande número de dados.

Para estas bases de dimensão elevada, uma possibilidade frequentemente utilizada para melhorar o desempenho das consultas é a utilização de buscas aproximadas. Em buscas aproximadas abre-se mão da garantia de fornecimento da resposta correta, buscando um ganho de desempenho ao se verificar um subconjunto dos pontos necessários que ainda permita certo grau de certeza da qualidade da resposta. Utilizando-se deste artifício há um *trade off* entre a qualidade da resposta fornecida e o custo computacional ao se realizar a busca, sendo uma decisão de projeto definir este melhor ajuste.

Neste contexto, a implementação de Randomized KD-trees [SilpaAnan and C., 2008] tem como objetivo aprimorar a precisão de buscas aproximadas para uma mesma quantidade de pontos verificados. Isto é feito com a construção de múltiplas KD-trees apresentando subdivisões distintas dos dados. Para criação destas árvores distintas são utilizadas diferentes ordens no processo de divisão dos dados pelas dimensões, sendo a ordem determinada por sucessivos sorteios em um subconjunto contendo as dimensões de maior variância ainda não selecionadas. O Algoritmo 3 destaca a mudança no processo de construção entre a KD-tree tradicional e uma RKD-tree.

Algorithm 3 Construção RKD-tree(dataset $\in R^k$)

```
1: vecVar(dataset)
2: for i = 1...k do
3:     p(i)=sorteio entre as D dimensões de maior variância não selecionadas
4: end for
5: RKD-tree ← Construir KD-tree com a sequência das divisões entre dimensões dada por p
6: return RKD-tree
```

A combinação de um conjunto de árvores RKD-tree permite a existência de múltiplas visões da distribuição dos dados, podendo a busca ser realizada em paralelo nestas árvores e de forma a ocorrer a verificação de subconjuntos distintos de pontos próximos à coordenada da consulta sem necessidade de um processo recursivo intensivo em cada uma destas individualmente.

Além de ser capaz de reduzir o número de nós analisados necessários para se obter determinada precisão da consulta, as RKD-trees ainda apresentam estrutura de processamento adequada a implementação de paralelismo e utilização em grandes bases de dados. Em termos de escalabilidade do processo, é possível distribuir as árvores em múltiplas unidades de processamento e realizar as buscas em paralelo, de forma a diminuir o tempo de processamento total de consulta as múltiplas árvores. Para as grandes bases de dados, outra possibilidade é a divisão dos dados entre múltiplas máquinas de forma que estes dados possam ser mantidos em memória nestas unidades de processamento. Desta forma, pode ser construída uma arquitetura com os dados particionados em múltiplos conjuntos de máquinas com cada máquina deste conjunto possuindo uma árvore de indexação para estes dados. Este processo permite que as consultas ocorram de maneira altamente paralelizável e que mesmo bases de dados com um número muito grande de elementos sejam mantidas constantemente em memória [Muja and Lowe, 2014].

Em termos de desempenho, esta abordagem se mostra competitiva tanto em tempo de processamento quanto em capacidade de paralelismo do processo de busca, apresentando desempenho superior ao da KD-tree tradicional e competindo com outras estruturas

utilizadas neste tipo de aplicação como o *Locality Sensitive Hashing* e a *Priority Search K-means Tree*.

5.7. Redução de Dimensionalidade

Desenvolvida em 1901, o PCA (Principal Component Analysis) é uma tradicional ferramenta de transformação para bases de dados, sendo utilizada em diversas áreas incluindo processamento de imagem, visualização de dados, recuperação de informação e redução de dimensionalidade.

O procedimento de execução busca uma transformação linear ortonormal dos dados que permita sua representação por meio de direções mais representativas deste conjunto. Em termos o PCA pode ser posto como a busca por uma transformação C, tendo:

$$X = YC$$

onde X é uma melhor representação da base de dados original Y. Neste sentido é pressuposto que as relações entre os atributos são lineares, guiando-se por aqueles de maior variância na análise. Outra característica desejada pela matriz X é que seus atributos possuam reduzida covariância, tendo seus valores o mais desacoplados possível [Shlens, 2003].

Dado o processo de construção é comum a utilização do PCA como técnica de redução da dimensionalidade dos dados, selecionando um subconjunto das componentes principais de maior variância e representando a base de dados apenas por este conjunto reduzido, com erro de representação relativo ao número de elementos selecionados.

Tendo em vista a definição generalista do procedimento PCA, foram desenvolvidas técnicas diversas para obtenção da matriz de transformação dos dados, podendo ser citadas dentre estas a decomposição da matriz de covariância e a decomposição por vetores singulares.

No contexto de *Big Data*, uma dificuldade que se impõe é a dimensão da base de dados, dado que os processos de construção da base modificada, envolvem, *a priori*, o processamento sobre todos os dados disponíveis, sendo relevante a escolha por uma técnica com maior capacidade de paralelização.

Neste sentido, a técnica sPCA [Elgamal et al., 2015] se apresenta como uma implementação do procedimento *Probabilistc PCA* voltado ao processamento em grandes bases de dados. O *Probabilistc PCA* é outro método de obtenção de aplicação do PCA que visualiza a obtenção da matriz de transformação como um processo inverso, onde dadas as múltiplas observações da base de dados, assume-se a existência de uma distribuição de probabilidade destes dados e busca-se, a partir da função de probabilidade advinda do processo, a estimativa de probabilidade máxima (MLE) da distribuição. A busca pela estimativa de probabilidade máxima é um procedimento recorrente em estatística, sendo possível se inspirar de algoritmos já existentes para resolução deste problema.

Em particular o sPCA, se utiliza de um procedimento iterativo que busca aproximações sucessivas para a matriz de transformação C, obtida pelo PCA originalmente. A técnica possui versões utilizando os principais frameworks para *Big Data*, *MapReduce*

e *Spark*, onde é buscada uma implementação com melhor desempenho tendo em vista a utilização em ambientes de arquitetura distribuída. O Algoritmo 4 mostra um esquema da implementação para o sPCA destacando em negrito as rotinas realizadas de maneira distribuída.

Algorithm 4 sPCA(dataset $\in R^k$)

1: $C = normrnd(D,d)$
2: $ss = normrnd(1,1)$
3: Y_m=**meanJob**(Y)
4: ss1=**FnormJob**(Y)
5: **while** condição de parada não satisfeita **do**
6: $M = C^T * C + ss * I$
7: $CM = C * M^{-1}$
8: $X_m = Y_m * CM$
9: $\{XtX, YtY\} =$ **YtXJob**(Y, Y_m, X_m, CM)
10: $XtX += ss * M^{-1}$
11: $C = YtY/XtX$
12: $ss2 =$ **trace**$(XtX * C^T * C)$
13: $ss3 =$ **ss3Job**(Y, Y_m, X_m, CM, C)
14: $ss = (ss1 + ss2 - 2 * ss3)/N/D$
15: **end while**
16: **return** KD-tree

Trata-se de um procedimento envolvendo diversas operações de produtos entre matrizes, mas havendo apenas alguns passos onde é necessário a paralelização das rotinas dada a dimensão das matrizes. Dentre as otimizações propostas no sPCA podem ser citados os seguintes passos: manutenção da esparsidade da base de dados através da propagação da media dos atributos da base de dados original, minimização da transferência de dados intermediários através do recômputo da matriz X, cálculo de operações de produto de matrizes de forma mais eficiente tomando proveito de uma possível esparsidade da base de dados original e das dimensões das matrizes envolvidas e o cômputo da norma de Froebenius de maneira paralelizável.

Verificando os testes de desempenho, as implementações em ambas as plataformas apresentaram desempenho competitivo, superando inclusive ao de ferramentas frequentemente utilizadas em projetos de *Big Data*, a implementação do método Stochastic SVD no Mahout-PCA (*MapReduce*) e a implementação da decomposição da matriz de covariância na biblioteca MLlib-PCA (*Spark*).

5.8. Clusterização

Uma das técnicas importantes na interpretação de grandes volumes de dados é o agrupamento de itens de dados semelhantes. Segundo [Berkhin, 2006], clusterização, ou agrupamento, é uma divisão de dados em grupos de objetos semelhantes. Cada grupo, chamado de cluster, é composto por objetos que são semelhantes entre si e muito diferente de objetos de outros grupos. A clusterização modela dados através de clusters. A modelagem de dados coloca a clusterização em uma perspectiva histórica enraizada na matemática, estatística e análise numérica. Do ponto de vista prático, o agrupamento desempenha um

papel de destaque em aplicações de mineração de dados, tais como a exploração científica de dados, recuperação de informação e de mineração de texto, aplicações de banco de dados espaciais, análise de Web, CRM, marketing, diagnósticos médicos, biologia computacional, e muitos outros.

Um exemplo simples de divisão de dados em grupos está disposto na Figura 5.11 e foi apresentado em [Matteucci, 2013]. Neste caso, identificam-se facilmente os 4 grupos em que os dados podem ser divididos; o critério de similaridade é a distância: dois ou mais objetos pertencem ao mesmo conjunto se eles são "próximos" de acordo com uma dada métrica de distância (neste caso, a distância geométrica). Esse tipo de agrupamento é chamado de clusterização baseada em distância.

Outro tipo de agrupamento é o agrupamento conceitual: dois ou mais objetos pertencem ao mesmo cluster se este define um conceito comum a todos os objetos. Em outras palavras, os objetos são agrupados de acordo com sua adequação aos conceitos descritivos, e não de acordo com as medidas de similaridade simples.

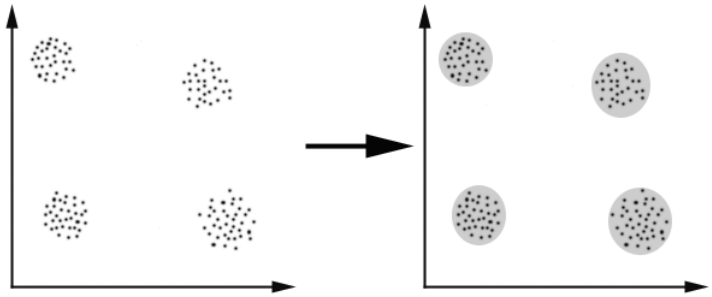

Figure 5.11. Exemplo simples de clusterização [Matteucci, 2013]

Segundo [Matteucci, 2013], não existe um "melhor" critério absoluto para decidir o que constitui um bom agrupamento que seja independente do objetivo final do agrupamento. Portanto, é o usuário que deve fornecer este critério, de tal maneira que o resultado do agrupamento atenderá às suas necessidades. Por exemplo, um usuário poderia estar interessado em encontrar representantes de grupos homogêneos (redução de dados), ou em buscar "agrupamentos naturais" e descrever suas propriedades desconhecidas (tipos de dados "naturais"), ou por buscar agrupamentos úteis e apropriados (classes de dados "úteis") ou ainda em buscar objetos de dados incomuns (detecção de outliers).

Quanto à classificação dos algoritmos de clusterização, [Berkhin, 2006] não a considera simples, pois as classificações se sobrepõem. São elas:

- métodos hieráquicos;
- métodos de particionamento;
- métodos baseados em grid;
- métodos baseados em co-ocorrência de dados categóricos;
- clusterização baseada em restrição;
- algoritmos de agrupamento usados em aprendizado de máquina;
- algoritmos de agrupamento escaláveis;

- algoritmos para dados de alta dimensionalidade.

No entanto, conforme [Berkhin, 2006], as técnicas de clusterização são tradicionalmente, amplamente divididas em hierárquicas e de particionamento. Cada uma deles se subdivide em várias diferentes técnicas. Enquanto algoritmos hierárquicos constroem clusters gradualmente, algoritmos de particionamento *aprendem* clusters diretamente. Ao fazer isso, eles tentam descobrir clusters iterativamente realocando pontos entre os subgrupos, ou tentam identificar os clusters como áreas altamente povoadas com dados. De acordo [Ochi et al., 2004], os algoritmos do primeiro tipo funcionam da seguinte forma: o conjunto de elementos é dividido em k subconjuntos, podendo k ser conhecido ou não, e cada configuração obtida é avaliada através de uma função-objetivo. Caso a avaliação da clusterização indique que a configuração não atende ao problema em questão, uma nova configuração é obtida realocando pontos entre os clusters, e o processo continua de forma iterativa até que algum critério de parada seja alcançado. Nesse esquema de realocação dos elementos entre os clusters, também conhecido como otimização iterativa [Ochi et al., 2004], os clusters podem ser melhorados gradativamente, o que não ocorre nos métodos hierárquicos. O k-médias, ou *k-means* [MacQueen, 1967] é um exemplo deste tipo de algoritmo.

[Berkhin, 2006] ainda comenta que algoritmos de particionamento do segundo tipo tentam descobrir componentes conexas densas de dados, estas são flexíveis quanto à sua forma. Um exemplo desse tipo de algoritmo bastante difundido é o DBSCAN [Ester et al., 1996a]. Esses algoritmos são menos sensíveis à *outliers* e podem descobrir conjuntos de formas irregulares. Eles geralmente trabalham com dados de baixa dimensionalidade de atributos numéricos, conhecidos como dados espaciais.

Já na clusterização hierárquica, conforme [Ochi et al., 2004], os clusters vão sendo formados gradativamente através de aglomerações ou divisões de elementos clusters, gerando uma hierarquia de clusters, normalmente representada através de uma estrutura em árvore, conforme exemplificado na Figura 5.12. Nessa classe de algoritmos, cada cluster com tamanho maior que 1 pode ser considerado como sendo composto por clusters menores.

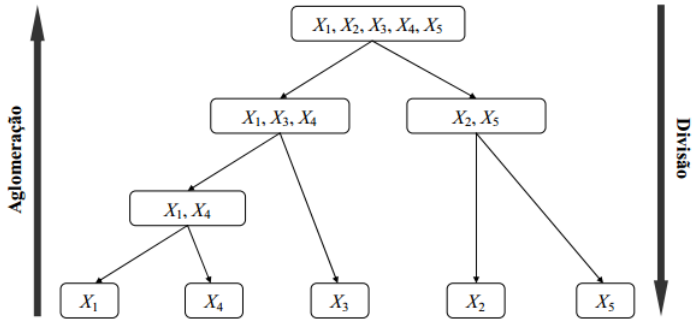

Figure 5.12. Exemplo de árvore de clusters na clusterização hierárquca [Ochi et al., 2004]

Nesses algoritmos existem dois tipos de abordagem: *bottom-up* e *top-down*. De acordo com [Ochi et al., 2004], nos algoritmos de aglomeração, que utilizam uma abor-

dagem *bottom-up*, cada elemento do conjunto é, inicialmente, associado a um cluster distinto, e novos clusters vão sendo formados pela união dos clusters existentes. Esta união ocorre de acordo com alguma medida que forneça a informação sobre quais deles estão mais próximos uns dos outros. Nos algoritmos de divisão, com uma abordagem top-down, inicialmente tem-se um único cluster contendo todos os elementos do conjunto e, a cada passo, são efetuadas divisões, formando novos clusters de tamanhos menores, conforme critérios pré-estabelecidos.

5.8.1. K-Means

K-means, ou k-médias, [MacQueen, 1967] é um algoritmo bastante difundido na literatura e um dos mais simples algoritmos de aprendizagem não supervisionada que resolvem o problema de clusterização. O objetivo é classificar um conjunto de elementos em um número k de clusters, dado como entrada. A idéia principal é definir *k centróides*, um para cada cluster, geralmente aleatórios, mas a melhor escolha é colocá-los longe um do outro. O próximo passo é levar cada ponto pertencente a um determinado conjunto de dados e associá-lo ao centróide mais próximo. Quando nenhum ponto está pendente, a primeira etapa é concluída e uma clusterização parcial é feita. Neste ponto é preciso voltar a calcular os k novos centróides dos clusters resultantes da etapa anterior. Logo após, uma re-alocação tem de ser feita entre os mesmos pontos e o novo centróide mais próximo. Esses passos são repetidos fazendo com que os k centróides mudem sua localização passo a passo até que não haja mais mudanças. Em outras palavras, os centróides não se movem mais [Matteucci, 2013].

O algoritmo visa minimizar uma função objetivo, neste caso uma função do erro quadrado. [Matteucci, 2013] descreve a função objetivo como sendo $J = \sum_{j=1}^{k} \sum_{i=1}^{n} ||x_i^j - c_j||^2$, onde $||x_i^j - c_j||^2$ é a medida de distância escolhida entre um ponto x_i^j e o centróide do cluster c_j. Essa função é um indicador da distância dos n pontos e seus respectivos centróides.

Os passos do algoritmo são [Fontana and Naldi, 2009]:

1. Atribuem-se valores iniciais de centróides seguindo algum critério, por exemplo, sorteio aleatório desses valores dentro dos limites de domínio de cada atributo.
2. Atribui-se cada objeto ao cluster cujo centróide esteja mais próximo ao objeto.
3. Recalcula-se o valor do centróide de cada cluster, como sendo a média dos objetos atuais do cluster.
4. Repete-se os passos 2 e 3 até que os clusters se estabilizem.

Em [Fontana and Naldi, 2009], apresenta-se a Figura 5.13 ilustrando a execução do K-means. Nas Figuras 5.13 (a), 5.13 (b), 5.13 (c), mostra-se a execução dos passos 1, 2 e 3 respectivamente. Na Figura 5.13 (d) apresenta-se a repetição dos passos 2 e 3; e nas figuras 5.13 (e) e 5.13 (f) ilustra-se a repetição dos passos 2 e 3, respectivamente.

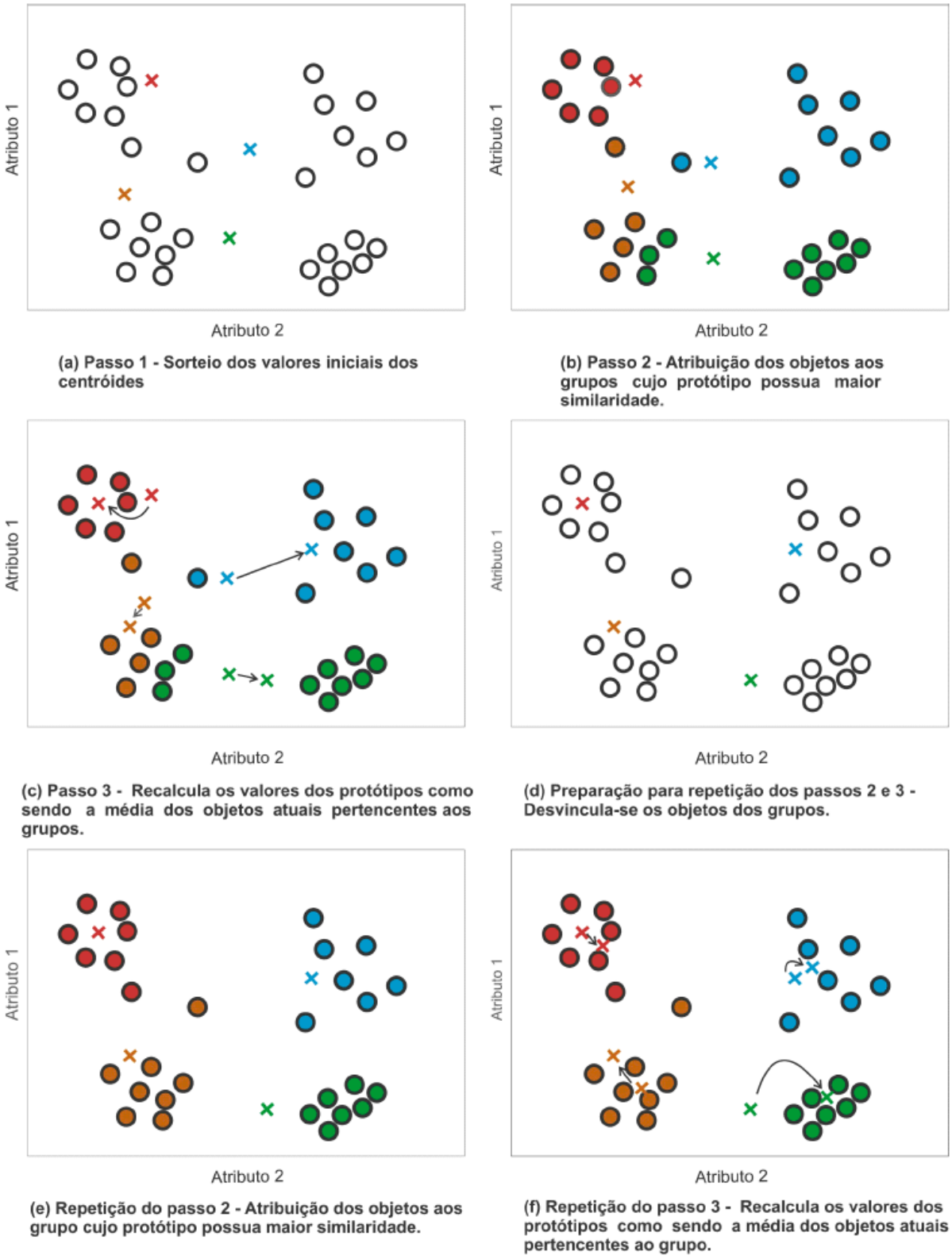

Figure 5.13. Exemplo de execução do k-means. Fonte: [Fontana and Naldi, 2009]

O K-means possui complexidade computacional equivalente a $O(n \times k \times i)$, onde

n é o número de objetos, k é o número de clusters e i o número de iterações. A distância entre os n objetos até cada um dos k centróides é calculada a cada iteração i. O número de dimensões também influência na complexidade do algoritmo, pois se o objeto tem d dimensões, a comparação entre dois desses objetos está em $O(d)$. Logo, para objetos que tenham 2 ou mais dimensões, a complexidade computacional do K-means passa a ser $O(n \times k \times i \times d)$.

5.8.2. Agrupamento de séries temporais. *YADING: Fast Clustering of Large-Scale Time Series Data*

O conceito de série temporal — uma sequência discreta de observações de algum fenômeno feita ao longo tempo — é utilizado em vários domínios do conhecimento, tais como, comunicação, finanças e economia [Shumway and Stoffer, 2010]. Ao passo que os intervalos de tempo entre as observações diminuem, assim como as características observadas aumentam, a escala e dimensionalidade das séries temporais compostas por essas observações demandam que novos algoritmos para tarefas de análise sejam desenvolvidos a fim de se obter eficiência e escalabilidade computacional. Aliás, em muitos cenários, o grau de eficiência computacional deve habilitar análises interativas em tempo real. Nesse contexto, o agrupamento de séries temporais é uma tarefa de análise relevante, o qual consiste em identificar, com base em algum critério de similaridade, grupos de instâncias (observações) homogêneas [Liao, 2005]. Por exemplo, em determinado *data center*, a cada intervalo de tempo predeterminado, podem ser medidas métricas de desempenho como percentual de uso de CPU e memória principal. Ao realizar o agrupamento das séries temporais formadas por essas medições ao longo do tempo, pode ser possível diagnosticar problemas de desempenho e compreender o status dos serviços disponíveis no *data center*.

Em face ao exposto, [Ding et al., 2015] propõem uma técnica de agrupamento que automaticamente congrega séries temporais de larga escala e dimensionalidade com altos resultados de desempenho computacional e qualidade: YADING. A técnica proposta consiste em três passos no agrupamento do conjunto de dados formado por séries temporais: (i) redução, (ii) agrupamento e (iii) atribuição do agrupamento. Em termos gerais, na etapa de redução, o conjunto de dados de entrada é amostrado e tem sua dimensionalidade reduzida. Já na etapa de agrupamento, o conjunto reduzido é agrupado. Na etapa final, o agrupamento previamente realizado é refletido no conjunto total dos dados. Os três passos da técnica YADING são discutidos a seguir, mas antes, para melhor compreensão, define-se formalmente o conceito de série temporal. Uma série temporal é um conjunto de dados $\mathbf{T}_{NxD} = \{T_1, T_2, ..., T_N\}$, onde N é o número de instâncias presentes no conjunto de dados, e D é a dimensão da série temporal, sendo cada instância $T_i = (t_{i1}, t_{i2}, ..., t_{iD})$.

Na primeira fase da técnica YADING, é realizada a redução do conjunto de entrada $\mathbf{T}_{NxD}$, isto é, identifica-se dois parâmetros $s, d \in \mathbf{N}$, onde $s \leq N$ é a quantidade de instâncias amostrada, enquanto $d \leq D$ é a quantidade reduzida de dimensões. Fundamentalmente a metodologia adotada na obtenção de s e d visa preservar a distribuição adjacente a $\mathbf{T}_{NxD}$ no conjunto de dados reduzido $\mathbf{T}_{sxd}$. Assim, para se obter s, assume-se que todo $T_i \in \mathbf{T}_{NxD}$ pertença a k grupos conhecidos. Partindo desse princípio, demonstra-se analiticamente que para $s = 2000$, isto é, ao serem escolhidos aleatoriamente duas mil instâncias de séries temporais, com 95% de confiança, consegue-se acurácia no agrupa-

mento próximo ao que seria realizado no conjunto total. É válido atentar para a relevância desse resultado: s independe de N. Além do processo de amostragem aleatória de T_i lançando mão a s, o método *Piecewise Aggregate Approximation of time series* (PAA) é utilizado na redução computacionalmente eficiente de dimensionalidade de $\mathbf{T}_{NxD}$. Apesar de computacionalmente eficiente, o método PAA demanda o parâmetro d, isto é, para quantas dimensões será reduzido $\mathbf{T}_{NxD}$. Nesse sentido, a técnica YADING utiliza uma abordagem em que, primeiramente são encontradas as frequências típicas de cada instância, sendo elas posteriormente ordenadas em forma de distribuição. A partir dessa distribuição seleciona-se um valor de percentil acima de 80%, o qual é empregado como d. Em conclusão, o tempo total para realizar o processo de redução de dados é de $\mathcal{O}(sD\log(D)+ND)$. Vale observar que o maior custo computacional é referente a aplicação do método PAA ($\mathcal{O}(ND)$) haja vista a necessidade de reduzir a dimensionalidade de cada instância do conjunto de dados.

Na segunda fase da técnica YADING, o conjunto amostrado de séries temporais é agrupado. Nesse cenário, deve-se atentar que os dados de séries temporais podem apresentar formas e densidades variadas, assim como perturbação de fase e ruído aleatório. Essas características possivelmente impactam na acurácia do agrupamento. Com isso, no intuito de garantir robustez no agrupamento dos dados assim como eficiência computacional, a técnica YADING emprega uma metodologia em que combina-se a métrica computacionalmente eficiente de distância L_1 e um método de agrupamento multi-densidade, sendo as estimativas de densidades ponto chave na técnica. Para realizar essas estimativas, primeiramente, para cada instância de série temporal em $T_i \in \mathbf{T}_{sxd}$ computa-se sua distância L_1 em relação a seus k vizinhos mais próximos, sendo esses valores listados em ordem decrescente em uma curva chamada de $kdis$. Os raios de densidade são definidos como os pontos de inflexão na curva $kdis$, que encontrados, são então empregados juntamente com o parâmetro $minPoints = 4$ pelo algoritmo DBSCAN [Ester et al., 1996b] no agrupamento. A propósito, é importante notar que se uma instância T_i foi agrupada com um raio de densidade, ela não é considerada para os demais raios. Enfim, o tempo total para realizar o processo de agrupamento é de $\mathcal{O}(s^2\log(s))$. Perceba que seria muito custoso realizar o agrupamento no conjunto de dados total.

Na última fase da técnica YADING o agrupamento realizado é refletido ao conjunto de dados total, isto é, os rótulos de grupo definidos para $\mathbf{T}_{sxd}$ são estendidos para $\mathbf{T}_{NxD}$. Com esse objetivo, a técnica YADING, para cada instância não rotulada T_i encontra a rotulada mais próxima T_j. Caso a distância entre T_i e T_j seja inferior ao raio de densidade utilizado no agrupamento da instância rotulada T_j, T_i recebe o rótulo de T_j. Caso contrário, T_j é considerada como ruído. A fim de evitar que seja computada a distância entre cada instância não rotulada e cada instância rotulada adota-se uma estratégia de poda que se baseia no fato de que se a distância entre uma instância não rotulada T_i e uma rotulada T_j é superior ao raio de densidade r utilizado no agrupamento de T_j, pela desigualdade triangular, tem-se que T_i está também distante mais do que r dos vizinhos rotulados de T_j. Assim, pode-se economizar a computação de distância entre T_i e os vizinhos de T_j. Nesse cenário, é construída uma estrutura chamada de *Sorted Neighbor Graph* (SNG), onde para cada ponto de núcleo definido na fase de agrupamento é armazenada suas distâncias em ordem ascendente para os outros elementos do conjunto de dados amostrado. Empiricamente, percebeu-se que o uso de SNG apresenta ganhos de

desempenho entre 2-4 vezes na prática. Por fim, tem-se que o tempo computacional da fase de atribuição é da ordem de $\mathcal{O}(Nsd)$.

Enfim, a metodologia de ponta a ponta no agrupamento de séries temporais empregada por YADING demonstra-se computacionalmente eficiente e provê resultados de qualidade. Mais especificamente, a independência do tamanho de amostragem, e a metodologia proposta de agrupamento multi-densidade são contribuições relevantes da técnica. Tem-se ainda como ponto positivo complexidade aproximadamente linear da técnica em relação à dimensão e ao tamanho de $\mathbf{T}_{NxD}$. Por fim, os trabalhos futuros vão da adaptação da técnica para conjuntos de dados compostos por séries temporais de escala e dimensão diversa, assim como, o uso da metodologia de YADING no congregação de séries temporais em *streaming*. Além desses pontos, [Mirzanurov et al., 2016] propõem melhorias concernentes à estrutura SNG e à acurácia da técnica.

5.9. Comentários Finais

A análise e interpretação de grandes volumes de dados é uma das atividades mais importantes na nova economia digital em que o bem de maior valor é a informação. Neste sentido, academia e as empresas redirecionam suas atividades de forma a prover técnicas, ferramentas e ambientes computacionais para fazer face aos desafios dessa nova ciência. Em particular, rotulou-se as atividade nessa área numa nova área denominada Ciência de Dados que integra: ciência da computação, estatística e matemática. Neste contexto, os arcabouços MR tomam importância significativa por aliarem propriedades importantes, tais como: escalabilidade, tolerância à falhas, extensibilidade para diferentes aplicações e flexibilidade com relação ao ambiente computacional. Apesar dessas qualidade, o desenvolvimento de aplicações neste novo contexto requer a reavaliação de algoritmos e sua contextualização sob os desafios de grandes volumes de dados. Neste curso, apresentamos algumas classes de algoritmos importantes para o tratamento de aplicações em grandes volumes. Algoritmos de particionamento de dados adicionam critérios semânticos para a distribuição dos dados pelas nós de processamento. Aspectos, por exemplo como o de vizinhança são beneficiados por particionamentos que os levem em consideração. Adicionalmente, técnicas de indexação eficiente permitem ações de busca por chave ou mesmo de forma espacial adaptadas ao contexto dos grandes volumes. Finalmente, agregar dados segundo critérios de semelhança permite reduzir a dimensionalidade e usar representantes em análise sobre a massa de dados. Obviamente, a lista de algoritmos não para por ai. Podemos investigar técnicas de dimensão de dimensionalidade, amostragem de dados, além de predição. Esse grande conjunto de técnicas forma essa nova disciplina de processamento de grandes volumes de dados.

References

[Barabási and Albert, 1999] Barabási, A. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286(5439):509–512.

[Berkhin, 2006] Berkhin, P. (2006). A Survey of Clustering Data Mining Techniques. *Grouping Multidimensional Data*, pages 25–71.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified

data processing on large clusters. *Commun. ACM*, 51(1):107–113.

[Ding et al., 2015] Ding, R., Wang, Q., Dang, Y., Fu, Q., Zhang, H., and Zhang, D. (2015). Yading: fast clustering of large-scale time series data. *Proceedings of the VLDB Endowment*, 8(5):473–484.

[Economist, 2010] Economist, T. (2010). The data deluge. *The Economist*.

[Eigenbrod et al., 2008] Eigenbrod, A., Courbin, F., Sluse, D., Meylan, G., and Agol, E. (2008). Microlensing variability in the gravitationally lensed quasar qso 2237+0305 ≡ the einstein cross. i. spectrophotometric monitoring with the vlt. *Astronomy & Astrophysics*, 480(3):647–661.

[Elgamal et al., 2015] Elgamal, T., Yabandeh, M., Aboulnaga, A., Mustafa, W., and Hefeeda, M. (2015). spca: Scalable principal component analysis for big data on distributed platforms. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 79–91, New York, NY, USA. ACM.

[Ester et al., 1996a] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. (1996a). A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press.

[Ester et al., 1996b] Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996b). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231.

[Facebook, 2017] Facebook (2017). Company info.

[Fontana and Naldi, 2009] Fontana, A. and Naldi, M. C. (2009). Estudo e comparação de métodos para estimação de números de grupos em problemas de agrupamento de dados. ICMC.

[Gaspar and Porto, 2014] Gaspar, D. and Porto, F. (2014). A multi-dimensional equi-depth partitioning strategy for astronomy catalog data.

[Ghemawat et al., 2003] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA. ACM.

[Gonzalez et al., 2012] Gonzalez, J., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 17–30.

[Guo et al., 2014] Guo, Y., Biczak, M., Varbanescu, A., Iosup, A., Martella, C., and Willke, T. (2014). How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In *Proc. IEEE Int. Parallel and Distributed Processing Symp. (IPDS)*, pages 395–404.

[Liao, 2005] Liao, T. W. (2005). Clustering of time series data—a survey. *Pattern recognition*, 38(11):1857–1874.

[Lovász et al., 2009] Lovász, L. et al. (2009). Very large graphs. *Current Developments in Math.*, 2008:67–128.

[Ludäscher et al., 2006] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., and Zhao, Y. (2006). Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065.

[MacQueen, 1967] MacQueen, J. B. (1967). Some Methods for Classification and Analysis of MultiVariate Observations. In Cam, L. M. L. and Neyman, J., editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press.

[Matteucci, 2013] Matteucci, M. (2013). A Tutorial on Clustering Algorithms.

[Mirzanurov et al., 2016] Mirzanurov, D., Nawaz, W., Lee, J., and Qu, Q. (2016). An effective cluster assignment strategy for large time series data. In *International Conference on Web-Age Information Management*, pages 330–341. Springer.

[Moore, 2000] Moore, G. E. (2000). Readings in computer architecture. chapter Cramming More Components Onto Integrated Circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Muja and Lowe, 2014] Muja, M. and Lowe, D. (2014). Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):107–113.

[Nature, 2008] Nature (2008). Big data welcome to the petacentre. *Nature*, 455.

[Ochi et al., 2004] Ochi, L. S., Dias, C. R., and Soares, S. S. F. (2004). *Clusterização em Mineração de Dados*.

[Ogasawara et al., 2013] Ogasawara, E., Dias, J., Silva, V., Chirigati, F., de Oliveira, D., Porto, F., Valduriez, P., and Mattoso, M. (2013). Chiron: a parallel engine for algebraic scientific workflows. *Concurrency and Computation: Practice and Experience*, 25(16):2327–2341.

[Oliveira et al., 2015] Oliveira, D., Boeres, C., Fausti, A., and Porto, F. (2015). Avaliação da localidade de dados intermediários na execução paralela de workflows bigdata. In *XXX Simpósio Brasileiro de Banco de Dados, SBBD 2015, Petrópolis, Rio de Janeiro, Brasil, October 13-16, 2015.*, pages 29–40.

[Ozsu and Valduriez, 1990] Ozsu, T. and Valduriez, P. (1990). *Principles of Distributed Databases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Petroni et al., 2015] Petroni, F., Querzoni, L., Daudjee, K., Kamali, S., and Iacoboni, G. (2015). Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 243–252, New York, NY, USA. ACM.

[Pires, 2016] Pires, V. (2016). *NACLUSTER: Resolvendo Entidades em Larga Escala a partir de Multiplos Catalogos da Astronomia*. PhD thesis, Universidade Federal do Ceará.

[Porto et al., 2017] Porto, F., Khatibi, A., Nobre, J. R., Ogasawara, E., Valduriez, P., and Shasha, D. (2017). Constellation Queries over Big Data. *ArXiv e-prints*.

[Samet, 2011] Samet, H. (2011). *The Design and Analysis of Spatial Data Structures*. Springer New York.

[SDSS,] SDSS. Sloan digital sky survey.

[Shlens, 2003] Shlens, J. (2003). A tutorial on principal component analysis: derivation, discussion and singular value decomposition.

[Shumway and Stoffer, 2010] Shumway, R. H. and Stoffer, D. S. (2010). *Time series analysis and its applications: with R examples*. Springer Science & Business Media.

[SilpaAnan and C., 2008] SilpaAnan and C., Hartley, R. (2008). Optimised kd-trees for fast image descriptor matching. In *26th IEEE Conference on Computer Vision and Pattern Recognition*.

[Verma et al., 2017] Verma, S., Leslie, L. M., Shin, Y., and Gupta, I. (2017). An experimental comparison of partitioning strategies in distributed graph processing. *Proc. VLDB Endow.*, 10(5):493–504.

[Zaharia et al., 2010] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA. USENIX Association.

[Zäschke et al., 2014] Zäschke, T., Zimmerli, C., and Norrie, M. C. (2014). The ph-tree: A space-efficient storage structure and multi-dimensional index. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 397–408, New York, NY, USA. ACM.