Chapter 3

Tecnologias para Gerenciamento de Dados na Era do Big Data

Victor Teixeira de Almeida e Vitor Alcântara Batista

Abstract

This chapter is an overview of big data management, and intends to explore and diferentiate several recent technologies. A classic problem of the database community will be used as a background for the examples given throughout this course: triangle counting on graphs. This problem has been chosen because it is being extensively used to identify the importance of individuals on social networks. Also, since it can be described by an algorithm that is simple to understand and yet complex to execute in terms of performance, the differences between technologies in design and performance will be easily demonstrated.

Resumo

Este capítulo pretende explorar e diferenciar de forma introdutória diversas tecnologias recentes para gerenciamento de dados na era do big data. Será utilizado como pano de fundo para os exemplos um problema clássico da comunidade de bancos de dados: a contagem de triângulos em grafos. Ele foi escolhido por ser um problema atual e prático, frequentemente utilizado para identificar a importância de indivíduos em redes sociais. Além disso, ele é de fácil representação e alta complexidade de execução. Através do seu uso, é possível demonstrar as diferenças entre as tecnologias em termos de expressividade e desempenho.

3.1. Introdução

A era da informação, com o advento da Internet (Web 2.0), está terminando. Estamos entrando na era da computação voltada ao consumidor, chamada por Grossman de *Device Era* [Grossman 2012] e por Wirfs-Brock de *Ambient Computing Era* [Wirfs-Brock 2011] (Figura 3.1). Nesta era, focada no indivíduo, as pessoas possuem um ambiente rico em recursos e comunicação, principalmente através de dispositivos móveis. A computação é

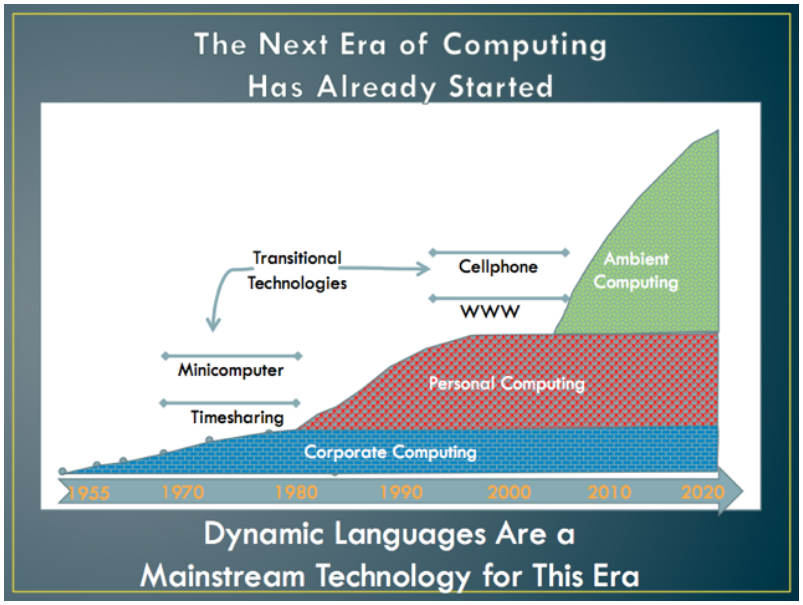

Figura 3.1. Eras da computação. Retirado do blog de Wirfs-Brock [Wirfs-Brock 2011].

somente um alicerce para a entrega e melhoria de vida do indivíduo, que é seu principal objetivo.

O que está acontecendo neste momento é que a quantidade de informação que se está gerando está em crescimento explosivo. Tecnologias consideradas de *big data* tornam-se necessárias para poder eficientemente armazenar, gerenciar, processar ou visualizar essa quantidade massiva de informação. Talvez não na mesma velocidade, mas também com um extraordinário crescimento, novas tecnologias têm surgido para suprir tal necessidade.

Este capítulo pretende explorar e diferenciar de forma introdutória diversas tecnologias recentes para o gerenciamento de dados na era do big data. Será utilizado como pano de fundo para os exemplos, um problema clássico da comunidade de bancos de dados: a contagem de triângulos. Esta problema é perfeito para explicar as diferenças entre as tecnologias em termos de expressividade e desempenho, pois pode ser representado por uma simples consulta SQL com duas junções, contudo extremamente complexa de ser executada eficientemente. A partir deste problema, é possível identificar como cada tecnologia se comporta para representar sua solução, bem como seu desempenho.

O capítulo está subdividido por tecnologia e na Seção 3.9 são apresentadas considerações finais sobre o que ainda há de tecnologia a ser detalhada e que não foi descrito nesse capítulo. A Seção 3.2 detalha o problema da contagem de triângulos, que irá permear as próximas seções de tecnologia. A Seção 3.3 mostra como funcionam bancos de dados relacionais tradicionais. As Seções 3.4 e 3.5 mostram as abordagens de bancos de dados orientados a colunas e em memória, respectivamente. A Seção 3.6 apresenta bancos de dados paralelos e introduz os conceitos de paralelismo de dados e computação. A Seção 3.7 que apresenta o Hadoop e alguns dos seus principais frameworks para análise de dados.

3.2. Contagem de triângulos

Nesta seção é descrito o problema da contagem de triângulos, que será utilizado ao longo deste capítulo. Este problema é extremamente interessante para a avaliação de tecnologias uma vez que é de simples descrição e implementação, logo didádico, e complexo em termos de execução, desempenho.

O problema, como o próprio nome diz, consiste em contar triângulos em um grafo, ou seja, contar os subgrafos t_i de um grafo G contendo 3 diferentes vértices conectados entre si (triângulos).

Uma das grandes aplicações da contagem de triângulos é o cálculo do coeficiente de agrupamento de um nó em um grafo ou do grafo como um todo. Esta métrica possui diversas aplicações práticas em análises de redes sociais. O coeficiente de agrupamento de um nó v expressa a probabilidade de dois nós vizinhos a v serem também vizinhos entre si. Para o grafo G como um todo, o coeficiente de agrupamento é a média dos coeficientes de cada vértice do grafo. Altos valores para este coeficiente significam uma comunidade coesa (*small world community*).

Implementações de algoritmos eficientes para este problema abundam na literatura. O Algoritmo 1, adapdado de [Chu and Cheng 2012] para retornar somente a contagem de triângulos, apresenta uma execução eficiente para o problema, e é a base para as principais implementações de algoritmos que assumem que os dados cabem na memória.

Algoritmo 1 Contagem de triângulos em memória

Input: Grafo $G = (V, E)$
Output: c, a contagem de triângulos em G

```
1: c ← 0
2: para cada v ∈ V faça
3:     para cada u ∈ adj_G(v), dado que u > v faça
4:         para cada w ∈ (adj_G(v) ∩ adj_G(u), dado que w > u faça
5:             c ← c + 1
6:         fim para
7:     fim para
8: fim para
9: return(c)
```

Este algoritmo começa por inicializar a variável c de contagem de triângulos com 0. Então, para todos os vértices do grafo (nomeados v), o algoritmo tenta resgatar um vértice que seja adjacente a v, nomeado de u, e outro vértice que seja ao mesmo tempo adjacente a v e a u, nomeado de w. Cada vez que esses três vértices conectados por arestas são encontrados, o algoritmo incrementa o contador de triângulos c. Há mais um ponto a ser explicado aqui que é um teste para remover duplicatas que garante que $u > v$ e $w > u$, assumindo que os vértices possuam identificadores únicos no domínio dos números naturais, por exemplo. A complexidade deste algoritmo depende da implementação da busca pelos vértices u e w nas listas de adjacências de v e u, respectivamente.

Nos exemplos desse capítulo, serão utilizados dois conjuntos de dados, que foram também adotados por Leskovec et al. [Leskovec and Mcauley 2012] e encontram-se

disponibilizados pelo Projeto SNAP (*Stanford Network Analyst Project*). Eles contém usuários do Facebook e do Twitter, com estatísticas descritas na Tabela 3.1. Deve-se atentar para o fato de que o Twitter e o Facebook possuem políticas diferentes; no Twitter é possível que um usuário siga algum outro sem que este também o siga de volta, enquanto que no Facebook a contrapartida é exigida. Isto torna o grafo do Twitter direcionado e o do Facebook não direcionado.

Tabela 3.1. Estatísticas dos conjuntos de redes sociais (Facebook e Twitter) utilizados.

Conjunto de dados	Facebook	Twitter
Vértices	4.039	81.306
Arestas	88.234	1.768.149
Triângulos	1.612.010	13.082.506

O que torna este problema de contagem de triângulos interessante, em termos de complexidade de execução, é a natureza dos dados. Normalmente, esses dados de redes sociais possuem distorção (*skew*), seguindo uma função de distribuição chamada *power-law*. Tentando explicar de uma forma simples, o que acontece é que a grande maioria dos vértices possuem muito poucas ligações, mas uma pequena fração possui muitas ligações. O gráfico abaixo mostra a distribuição dos dados. O eixo das abcissas mostra o grau do vértice e o eixo das ordenadas mostra quantos vértices possuem tal grau.

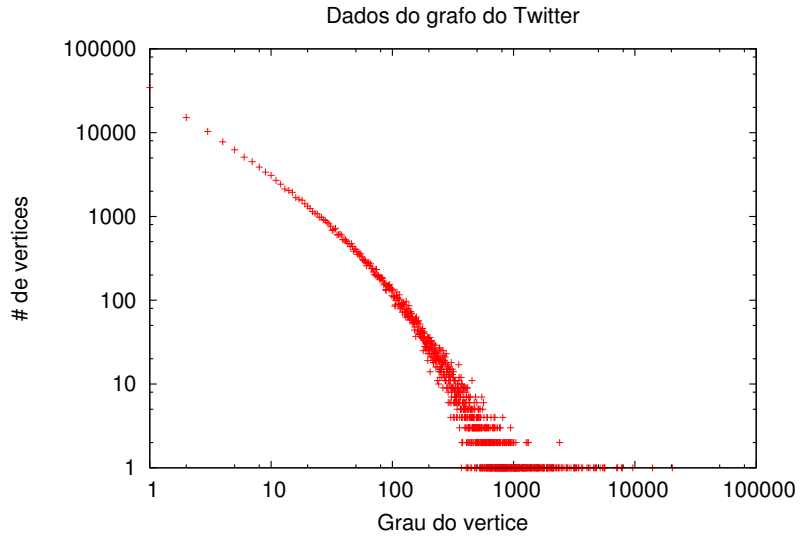

Figura 3.2. Distribuição de dados de redes sociais (*power-law*) com dados do Twitter

3.3. Bancos de dados relacionais

Bancos de dados relacionais são sistemas que implementam o modelo de dados relacional proposto por Codd na década de 70 [Codd 1970], rapidamente implementado por um sistema chamado System R da IBM [Astrahan et al. 1976]. A principal ideia é a de representar os dados em forma de relações (tabelas) e as operações são definidas a partir de uma álgebra: a álgebra relacional. O sistema recebe comandos através de uma linguagem

declarativa (SQL) [Chamberlin and Boyce 1974], os converte em operadores da álgebra relacional para então executá-los nos dados persistentes nas relações.

recID	fname	lname	gender	city	country	birthday
...	...	...	...	...	...	...
39	John	Smith	m	Chicago	USA	12.03.1964
40	Mary	Brown	f	London	UK	12.05.1964
41	Jane	Doe	f	Palo Alto	USA	23.04.1976
42	John	Doe	m	Palo Alto	USA	17.06.1952
43	Peter	Schmidt	m	Potsdam	GER	11.11.1975
...	...	...	...		...	...

Figura 3.3. Representação de uma tabela em um banco de dados relacional.

A Figura 3.3 mostra a representação de uma relação contendo informações sobre indivíduos em um formato de tabela. Esta relação possui os seguintes atributos do indivíduo : (i) *fname*, o nome; (ii) *lname*, o sobrenome; (iii) *gender*, o sexo, *m* para masculino e *f* para feminino; (iv) *city*, a cidade onde nasceu; (v) *country*, o país onde nasceu; e (vi) *birthday*, a data de nascimento.

Os principais operadores da álgebra relacional são: projeção (Π), seleção (σ), junção ($\bowtie$). Sejam duas relações R e S com atributos $r_1, \ldots, r_m$ e $s_1, \ldots, s_n$, representadas por $R(r_1, \ldots, r_m)$ e $S(s_1, \ldots, s_n)$. Uma projeção em R recebe como argumento uma lista de atributos subconjunto dos atributos de R e retorna a relação contendo somente os atributos desta lista. A projeção é representada por $\Pi_{lista\ de\ atributos}(R)$. Uma seleção em R recebe como argumentos uma condição em forma de predicado e retorna todas as tuplas em R que satisfazem o predicado. A seleção é representada por $\sigma_{predicado}(R)$. A junção é um operador binário e é aplicado sobre duas relações R e S. Ele recebe como argumento uma condição em forma de predicado (usualmente a igualdade entre dois atributos das relações R e S), e retorna a relação correspondente ao produto cartesiano das duas relações R e S cujas tuplas satisfazem o predicado da junção. A junção é representada por $R \bowtie_{predicado} S$.

A linguagem SQL expressa de forma declarativa operações sobre os dados armazenados nas relações a partir da álgebra relacional. Uma versão simplificada da linguagem SQL é mostrada a seguir:

```
SELECT <lista de atributos>
FROM <lista de tabelas>
WHERE <predicado>;
```

Após o sistema receber uma consulta na linguagem SQL, esta deve ser expressada segundo operadores da álgebra relacional da forma mais eficiente possível. Um otimizador de consultas é o responsável por esta tradução. A ordem dos operadores influencia no resultado; normalmente seleções (σ) são as primeiras a serem executadas, pois reduzem o tamanho das relações resultantes. A ordem das junções também é de extrema importância; uma junção entre as relações R, S e T pode ser executada nas seguintes ordens: (i) $R \bowtie S \bowtie T$, (ii) $R \bowtie T \bowtie S$ e (iii) $S \bowtie T \bowtie R$. Adicionalmente, existem inúmeros al-

goritmos para a execução eficiente das junções que devem coexistir no sistema e serem utilizados em circunstâncias em que são os mais eficientes [Mishra and Eich 1992].

O problema da contagem de triângulos da Seção 3.2 pode ser expresso na seguinte consulta SQL, assumindo que há uma relação chamada Twitter com os attributos *follower* e *followee* contendo números inteiros de identificadores de usuários representando relacionamentos na rede social, onde *follower* segue *followee*.

```
SELECT COUNT(*)
FROM TWITTER R, TWITTER, S, TWITTER T
WHERE R.followee = S.follower AND
      S.followee = T.follower AND
      T.followee = R.follower AND
      R.follower > S.follower AND
      S.follower > T.follower;
```

Uma ordem de execução desta consulta, expressa em operadores da álgebra relacional é

$$\sigma_{S.follower>T.follower}(\sigma_{R.follower>S.follower}((R \bowtie_{R.followee=S.follower} S) \bowtie_{(S.followee=T.follower)\wedge(T.followee=R.follower)} T)) \quad (1)$$

Este plano de execução irá seguir os seguintes passos:

1. Primeira junção entre as tabelas R e S com predicado $R.followee = S.follower$
2. Seleção com predicado $R.follower > S.follower$ aplicado ao resultado anterior
3. Junção do resultado anterior com a tabela T com predicado $(S.followee = T.follower) \wedge (T.followee = R.follower)$
4. Seleção com predicado $S.follower > T.follower$

3.4. Bancos de dados colunares

A ideia por traz dos bancos de dados colunares não é nova. Uma das primeiras propostas de organizar os dados de um banco de dados por colunas, em vez da tradicional representação por linhas dos bancos de dados relacionais, apareceu em 1969 [Estabrook and Brill 1969]. De forma resumida, enquanto um banco de dados relacional tradicional armazena cada registro ou tupla em um espaço contínuo do disco (ou memória), os bancos de dados colunares armazenam cada coluna em espaços contínuos [Abadi et al. 2009].

A grande vantagem da representação colunar é a compressão de dados, uma vez que o domínio dos dados é preservado em espaços contíguos de disco ou memória. Uma das principais formas de compressão de dados é por dicionário, onde cada coluna de uma tabela (ou relação) é dividida em um índice com os valores distintos ordenados e um vetor com a codificação dos valores de cada tupla na mesma sequência em que aparecem na tabela. Esta codificação utiliza um numero mínimo de bits do domínio de valores de cada

registro, baseada no dicionário. Com os dados comprimidos, menos bytes trafegam entre o disco e a memória principal, tornando operações de consulta e agregação mais rápidas. A Figura 3.3 mostra a representação tradicional de uma tabela num banco de dados relacional tradicional (orientado por linhas) e a Figura 3.4 mostra como a primeira coluna da relação é organizada em um banco de dados colunar com compressão por dicionário.

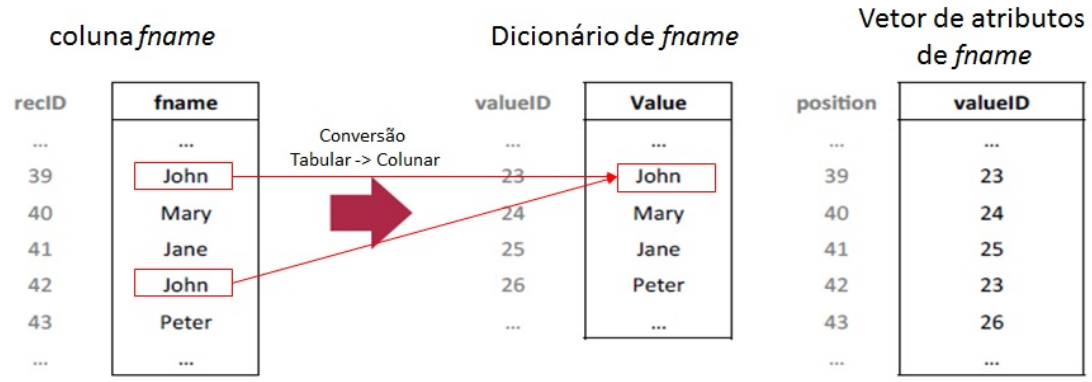

Figura 3.4. Organização da coluna fname em um banco de dados colunar.

Para explicar melhor o mecanismo de compressão, suponha que a coluna *fname* tenha 100 caracteres representados por 1 byte cada. Suponha também que a tabela em questão trata-se da lista de todos os cidadãos brasileiros, ou seja, há aproximadamente 200 milhões de registros. Logo, o espaço de armazenamento no modelo tradicional é de $200 * 10^6 * 100$ $Bytes \cong 18,62$ $GBytes$. No armazenamento em formato de colunas, os dados são armazenados em um dicionário que contém uma entrada para cada valor distinto. Imagine que há 10 mil nomes (primeiro nome) distintos no Brasil. Com isso, para o dicionário, são necessários $10 * 10^3 * 100$ $bytes \cong 1$ $Mbyte$. Já o vetor de valores conterá os 200 milhões de registros, mas codificados pela quantidade mínima de bits necessários para se codificar os 10 mil valores distintos $log_2(\ 10.000) \cong 13,2$, ou seja, 14 bits. Nesse caso, o vetor de valores terá $200 * 10^6 * 14$ $bits \cong 2,6$ $GBytes$. Nesse exemplo simples é possível observar um fator de compressão de $18,62/(0,001+2,6) \cong 7,2$ vezes.

A recuperação de um determinado registro (tupla) passa por um acesso direto à posição dele em cada um dos vetores de valor das colunas da tabela e mais um acesso em cada dicionário para traduzir a codificação para o valor original.

O grande problema nessa representação são operações de remoção e inserção, que usualmente causam uma reorganização do dicionário e consequentemente uma reorganização de todos os dados das colunas cujo dicionário foi reordenado. Por isso, o uso desse tipo de tecnologia deve ser preferível em conjuntos de dados cuja operação predominante seja de consulta. Plattner [Plattner 2009] alega que os bancos de dados corporativos possuem uma carga predominantemente de consulta e com isso, pode-se unificar os repositórios OLTP e OLAP na mesma estrutura, sugerindo para tal, a representação colunar.

Os sistemas modernos que utilizam a representação colunar lançam mão de diversas estratégias para minimizar esses efeitos de reorganização dos dicionários. O SAP HANA, por exemplo, divide os dados em principal e diferencial [Plattner and Zeier 2012].

Novos registros e atualizações de valores são incluídos no conjunto diferencial, que é mantido em tamanho pequeno. As operações de consulta juntam os dados do conjunto principal com o conjunto diferencial. Embora a busca no conjunto diferencial seja ineficiente, ela é realizada sobre um conjunto pequeno de dados. De tempos em tempos, os conjuntos são mesclados resultando em um novo conjunto principal e um conjunto diferencial praticamente vazio, que conterá apenas as operações realizadas durante a operação de mesclagem.

Há também outros mecanismos de compressão para outros domínios de dados, como por exemplo, números inteiros e de ponto flutuante [Abadi et al. 2006, Zukowski et al. 2006]. É importante ressaltar que todos estes mecanismos de compressão de dados tentam balancear a taxa de compressão de dados com o processamento para compressão e descompressão. Em geral, os mecanismos mais eficientes em termos de taxa de compressão não são utilizados, uma vez que o processamento (CPU) para descompressão dos dados durante a execução das consultas pode ser um fator determinante. Adicionalmente, em geral, todos os mecanismos de compressão de dados preservam as propriedades de igualdade e ordenação dos dados, ou seja, valores $=$, $<$ ou $>$ no domínio original dos dados são também $=$, $<$ ou $>$ no domínio de compressão. Esta propriedade é extremamente importante, pois a execução de diversos algoritmos da álgebra relacional que envolvem a comparação e ordenação dos dados, como a junção, podem ser executados no domínio dos dados comprimidos, evitando assim processamento desnecessário de descompressão dos dados. É fácil observar esta propriedade no exemplo acima da compressão da coluna *fname*, desde que o dicionário esteja ordenado por nome.

Atualmente, os principais fornecedores de bancos de dados relacionais tradicionais também fornecem soluções de armazenamento colunar, como Microsoft, Oracle, IBM, SAP e outros. Entretanto, há sistemas gerenciadores de bancos de dados puramente colunares, os comerciais SAP IQ e Actian Vectorwise, e os de código aberto C-Store e MonetDB.

3.5. Bancos de dados em memória

Sistemas de banco de dados em memória são aqueles em que a fonte primária dos dados reside em memória principal (RAM). Esses dados têm, usualmente, cópia em disco, para eventuais falhas no hardware ou simplesmente falta de energia. Embora os sistemas de bancos de dados tradicionais também mantenham dados em memória na forma de *cache*, a principal diferença é que, nos bancos de dados em memória, a fonte de dado primária (ou principal) está armazenada na memória RAM, enquanto nos tradicionais, está armazenada em disco [Garcia-Molina and Salem 1992, DeWitt et al. 1984].

Esse tipo de tecnologia ganhou força nos últimos anos devido aos avanços nas arquiteturas de hardware que, atualmente, permitem sistemas com Terabytes de memória RAM compartilhada entre vários processadores. Além disso, houve um barateamento no custo desses equipamentos, o que tornou viável os sistemas de banco de dados em memória.

Como o acesso à memória principal chega a ser cerca de 1.000 vezes mais rápido que os discos modernos como SSD (disco de estado sólido), esse tipo de tecnologia é bem convidativo. O leitor mais atento pode se perguntar: e se um sistema de banco de dados

tradicional tiver um *cache* grande o suficiente para caber todo o volume de dados, qual seria a diferença? Ainda assim, os sistemas tradicionais são projetados de forma não ótima para uso da memória, pois há ainda a indireção do *cache*. Por exemplo, será necessário consultar um gerenciador do *cache* toda vez que for acessar o dado; outro exemplo são as estruturas dos índices, que estão em estruturas onde o acesso não é imediato Árvores B+.

Embora existam hardwares especializados com memória não volátil, fontes de energia redundantes e outros mecanismos para minimizar falhas no hardware, ainda é virtualmente impossível garantir que o dado em memória principal esteja seguro. Por isso, uma questão muito importante para bancos de dados em memória é a recuperação de falhas. A estratégia tipicamente utilizada é persistir em disco cada uma das transações em um *log*. Uma transação só é concluída após a escrita da mesma no disco (*log*). De tempos em tempos, são criados *checkpoints*, onde uma cópia da memória principal é realizada em disco, podendo assim, descartar os *logs* de transações anteriores ao *checkpoint*. A frequência da criação desses *checkpoints* depende da confiabilidade do hardware e da volatilidade dos dados.

Outras questões importantes do projeto de sistemas de bancos de dados em memória, como controle de concorrência, processamento de transações, organização dos dados, métodos de acesso, processamento de consultas, desempenho e clusterização são discutidos em mais detalhes por Garcia-Molina e Salem [Garcia-Molina and Salem 1992].

Atualmente, praticamente todos os grandes fornecedores de soluções tradicionais de bancos de dados relacionais também possuem versões em memória de seus produtos, como a Microsoft, Oracle e IBM, mas também há fornecedores especializados nesse tipo de solução, sendo os principais comerciais o SAP HANA e o VoltDB, e o MemSQL de código aberto. Alguns desses produtos também possuem características de bancos de dados colunares apresentados na seção 3.4, principalmente com o objetivo de atingir boa compressão dos dados e melhor utilização da memória.

3.6. Bancos de dados paralelos

Bancos de dados paralelos ou massivamente paralelos (MPP, do inglês *massively parallel processing*) também não são tecnologias recentes, discussões sobre esse assunto e implementações de sistemas datam da década de 80-90 [DeWitt and Gray 1992, Fushimi et al. 1986]. Nestes sistemas, os dados das tabelas são distribuídos em diversos nós de um cluster e o processamento da consulta é paralelizado.

Os dados podem ser distribuídos através de particionamento vertical, onde as tabelas são quebradas por colunas; horizontal, onde as tabelas são quebradas por tuplas; ou ambos. A distribuição dos dados entre os nós pode utilizar o conhecimento prévio das consultas mais utilizas no sistema (*query workload*), e o otimizador de consultas, nestes casos, conhecendo a forma como os dados estão distribuídos, pode repassar a parte das consultas para os nós específicos que devem processá-la. O problema geral desta abordagem é que existe a possibilidade de desbalanceamento de nós (*data skew*). O particionamento dos dados por funções de hash é bastante utilizado, pois através de boas funções de hash é possível evitar este desbalanceamento.

A arquitetura dos bancos de dados paralelos pode ser de compartilhamento de

memória (*shared-memory*, onde a memória é compartilhada entre os nós do cluster; compartilhamento de disco (*shared-disk*), onde cada nó do cluster possui sua própria memória, mas o disco é compartilhado entre todos os nós; ou sem compartilhamento (*shared-nothing*), onde cada nó possui sua própria memória e disco e o compartilhamento de dados entre nós do cluster se faz através de transferência por mensagens. As arquiteturas com compartilhamento (*shared-memory* e *shared-disk*) mostraram-se ineficientes em escalabilidade para grandes implantações [DeWitt and Gray 1992, Stonebraker 1986]. A Figura 3.5 mostra um banco de dados paralelo com um nó mestre e arquitetura sem compartilhamento.

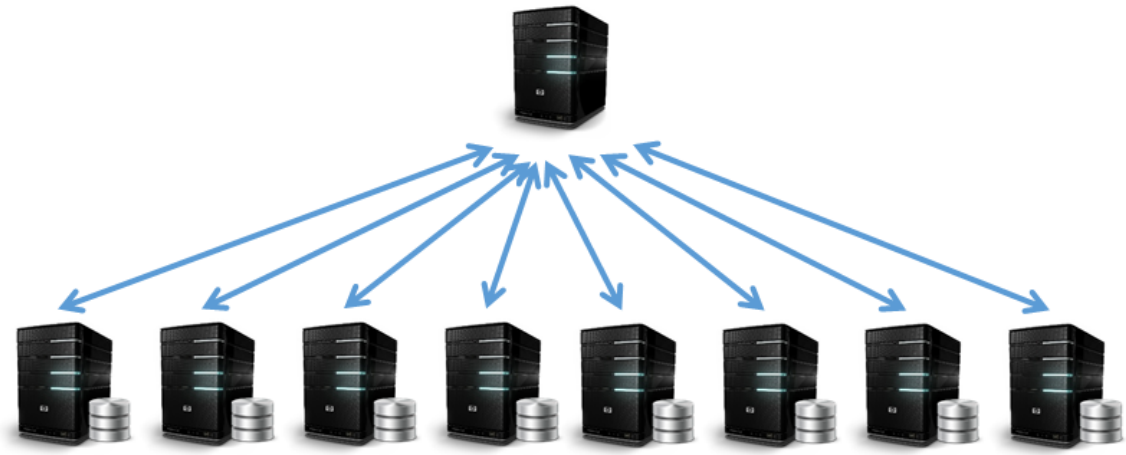

Figura 3.5. Banco de dados paralelo e a arquitetura sem compartilhamento.

Os operadores da álgebra relacional são todos paralelizáveis, principalmente a junção, com algoritmos de distribuição dos dados por função hash [Schneider and DeWitt 1989]. A ideia principal destes algoritmos é a de que cada nó lê sua parcela dos dados, aplica uma função hash nos atributos da junção para determinar o nó de destino e envia os dados para os nós específicos. Desta forma, quando os nós recebem os dados, há a garantia (pela função de hash) de que os nós que irão participar do resultado da junção estão fisicamente localizados juntos, neste passo. Cada nó então executa um algoritmo linear de junção, tipicamente uma junção por hash em memória, e reporta o resultado para o nó coletor. Este processo é demonstrado na Figura 3.6. Nesta figura, é mostrada uma arquitetura com quatro nós sem compartilhamento, e estes quatro nós estão replicados em duas camadas para melhor mostrar a fase de espalhamento dos dados por função de hash (*shuffle*).

A solução do problema da contagem dos triângulos apresentada na Seção 3.2, utilizando banco de dados paralelos é a mesma utilizada em bancos de dados relacionais, uma vez que tal tecnologia implementa a álgebra relacional. Na Seção 3.3 é mostrada a consulta SQL que resolve o problema. Esta consulta envolve duas junções, que serão executadas em paralelo, e as seleções de remoção de duplicatas que serão aplicadas após as junções no fluxo de execução das consultas (*pipeline*).

Os principais sistemas gerenciadores paralelos de bancos de dados comerciais são Pivotal Greenplum Database, HP Vertica, Teradata, Microsoft SQL Server Parallel Data

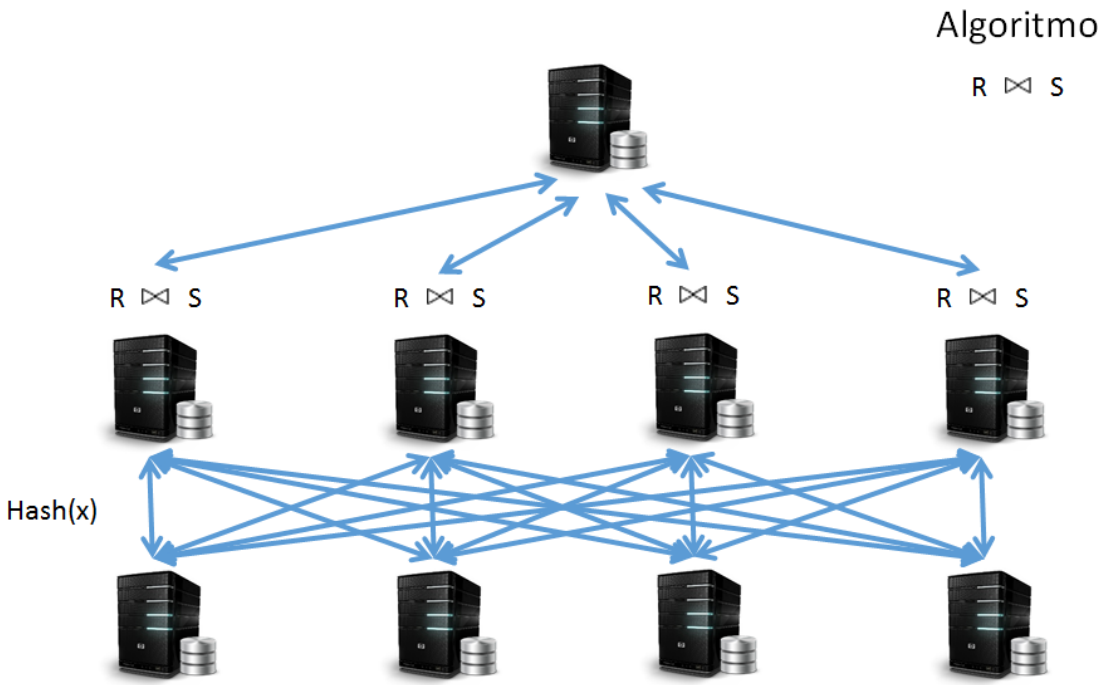

Figura 3.6. Junção por hash em um banco de dados paralelo sem compartilhamento.

Warehouse e IBM Netezza. Dentre os de código aberto pode-se citar: AsterixDB, Stratosphere (atualmente Apache Flink), Myria e Presto.

3.7. Hadoop

3.7.1. Breve histórico

O problema era simples: como criar um índice para uma máquina de busca de toda a Internet? Foi com esse desafio que Mike Cafarella e Doug Cutting resolveram desenvolver o Apache Nutch. Rapidamente o *crawler* e a máquina de busca ficaram prontos, mas eles perceberam que a arquitetura não escalaria para criar um índice de mais de um bilhão de páginas da Internet. Na mesma época, a equipe do Google publicou um artigo que explicava a arquitetura do GFS (Google FileSystem) [Ghemawat et al. 2003], que era um sistema de arquivos distribuído usado em sua máquina de busca. Doug e Mike decidiram criar uma implementação *open source* dessa arquitetura e a chamaram de NDFS (Nutch Distributed FileSystem).

Em 2004, a equipe do Google publicou um novo artigo detalhando como era possível criar um índice de toda a Internet usando o mecanismo de processamento paralelo denominado MapReduce [Dean and Ghemawat 2008]. Com base nesse trabalho, os desenvolvedores do Nutch migraram a maior parte de seus algoritmos para executar sobre o MapReduce e o NDFS. Mais tarde, Doug Cutting foi trabalhar no Yahoo! liderando uma equipe que construiu a nova geração de máquina de busca deles. Depois, o NDFS (posteriormente chamado de HDFS ou *Hadoop Distributed Filesystem*) e o MapReduce tornaram-se um projeto da Apache Software Foundation sob o nome de Apache Hadoop.

Desde então, o Hadoop tem sido usado mundialmente para processar enormes quantidades de dados. Vários frameworks foram construídos usando a sua infraestrutura, como será visto a seguir. Diversos fornecedores criaram suas próprias distribuições do Hadoop, como Microsoft, IBM, EMC, Oracle e outras empresas especializadas como Cloudera e Hortonworks.

3.7.2. Funcionamento do HDFS

O HDFS, como mencionado acima, é um sistema de arquivos distribuídos, projetado para armazenar arquivos muito grandes[1] executando em computadores padrão de baixo custo.

Assim como em qualquer sistema de arquivos, um arquivo é dividido em **blocos** de dados. Enquanto tipicamente um sistema de arquivos tradicional armazena dados em pequenos blocos (e.g. 512 bytes ou 1 kbyte), o HDFS usa, por padrão, blocos de 64MB. Isso torna o HDFS ineficiente para uso em arquivos muito pequenos e numerosos. Para garantir disponibilidade e leitura em paralelo, cada um dos blocos é replicado em um dos nós de um *cluster* HDFS. Quando um disco ou um dos nós do *cluster* falha, além do bloco poder ser lido de outro nó, o sistema de arquivos automaticamente recria os blocos presentes naquele disco em outros nós do *cluster*.

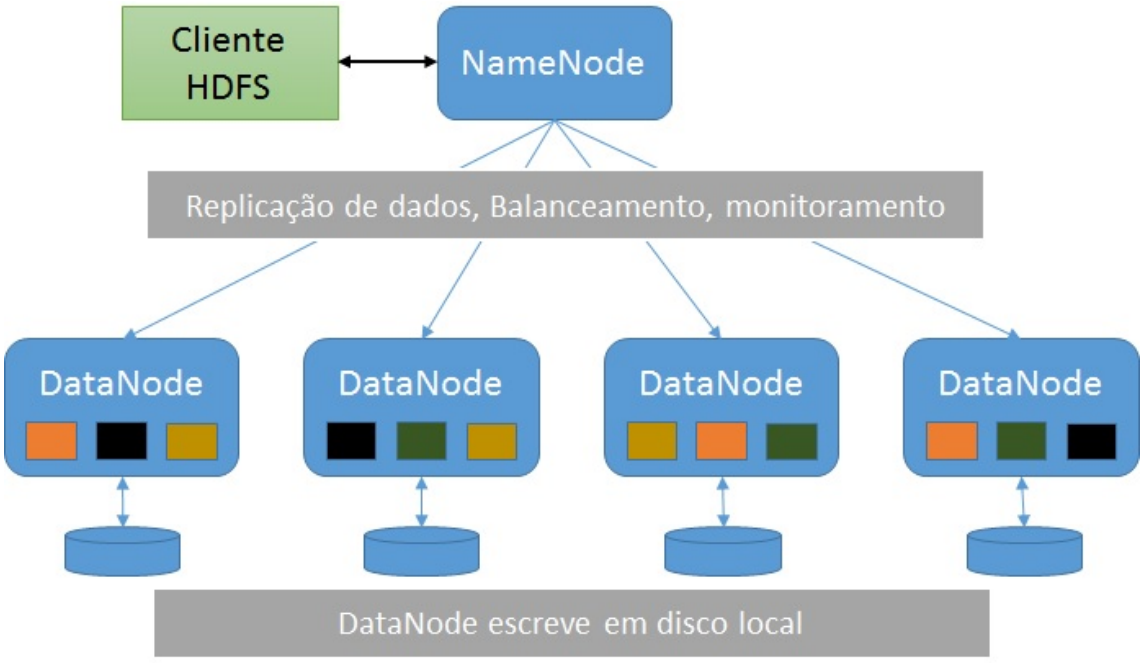

Figura 3.7. Visão geral da arquitetura do HDFS

A arquitetura de um *cluster* HDFS divide-se em *NameNode* e *DataNode*. O primeiro armazena um índice de arquivos e de seus blocos e o segundo armazena os dados (blocos). Um cliente que queira ler arquivos no HDFS, primeiro consulta o *NameNode*, que então diz de quais nós do cluster os blocos serão lidos, garantindo um balanceamento de carga na leitura. Arquivos criados no HDFS só podem ser modificados anexando conteúdo no final. Não é possível modificar blocos já escritos. A falha do *DataNode* implica

[1] Atualmente há instâncias do HDFS armazenando PetaBytes de dados.

na indisponibilidade de todo o HDFS e, por esse motivo, é importante mantê-lo resiliente a falha com mecanismos de redundância. Uma visão geral dessa arquitetura pode ser vista na figura 3.7.

3.7.3. O ecossistema Hadoop

Atualmente o Hadoop conta com um ecossistema com diversos *frameworks*. Uma lista, não exaustiva, de alguns dos principais projetos que compõem o ecossistema pode ser vista abaixo.

- Ambari: Uma ferramenta web para aprovisionamento e gerenciamento de um cluster Hadoop e de diversos de seus componentes.
- HBase: Um banco de dados não relacional (NoSQL) e colunar que utiliza a infraestrutura do Hadoop como mecanismo de armazenamento.
- Hive: Uma infraestrutura de armazém de dados com suporte a sumarização de dados e consultas.
- Pig: Uma linguagem de alto nível para fluxo de dados e um *framework* de execução de computação distribuída.
- Spark: Uma *engine* rápida e de propósito geral para processamento de dados em memória baseados nos dados do HDFS. O Spark oferece um modelo de programação simples e poderoso para executar uma enorme gama de atividades como ETL, aprendizagem de máquina, processamento contínuo de dados, processamento de grafos, etc.
- Sqoop: uma ferramenta para transferência massiva de dados entre bancos de dados relacionais e o HDFS.
- Mahout: Um conjunto de bibliotecas para executar algoritmos de aprendizagem de máquina e mineração de dados. Os coordenadores do projeto decidiram mover a implementação dos algoritmos de MapReduce para o Spark.

3.7.4. MapReduce

O Hadoop MapReduce é um *framework* para facilitar a escrita de programas de computador para processar uma enorme quantidade de dados de forma paralela, distribuída e resiliente a falhas. Os dados de entrada para um *job* MapReduce, por estarem armazenados no HDFS, são também processados de forma distribuída, aproveitando dos dados disponíveis localmente em um nó do *cluster*. A Figura 3.8 apresenta um desenho esquemático de um *job* MapReduce, que é, de forma resumida, composto por duas fases:

1. *Map*: quando os dados são processados e produzem saídas como tuplas no formato (*chave*, *valor*);
2. *Reduce*: quando as tuplas com mesma *chave* são agrupadas para alguma atividade de agregação.

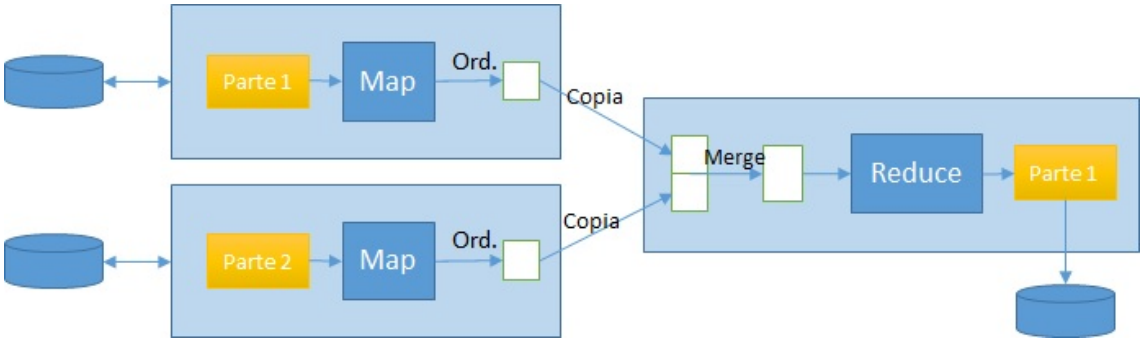

Figura 3.8. Um *job* MapReduce

Um exemplo bem simples para entender o MapReduce é um job para contar a ocorrência de cada palavra em um texto. Na fase de *Map*, cada linha lida do arquivo é dividida em suas palavras que produzem uma saída {PalavraA, 1}. Note que se uma palavra aparece duas vezes na mesma linha duas tuplas idênticas serão produzidas. Na faze de *Reduce*, as tuplas com mesma chave serão agrupadas e os valores 1 serão somados.

Em geral, o programador não precisa se preocupar com comunicação de dados, tratamento de concorrência e eventuais falhas em algum nó que está processando um determinado *job*. Esse é um dos grandes diferenciais do Hadoop MapReduce.

Uma possível solução para contagem de triângulos utilizando MapReduce, baseada na proposta feita por Suri e Sergei [Suri and Vassilvitskii 2011], envolve o encadeamento de 3 *jobs* MapReduce. O primeiro constrói um conjunto de todas as tríades (par de arestas que compartilham um vértice) de um grafo. Essas tríades, juntamente com as arestas originais são gerados como linhas, mas com um atributo identificador dizendo se é uma aresta original do arquivo ou se foi gerada pelo primeiro passo. Essa saída serve de entrada para o segundo *job*, que conecta as tríades com as arestas originais, formando os triângulos. Por fim, o terceiro *job* conta o número de triângulos gerados pelo passo anterior. Vamos olhar um exemplo para melhor entendimento.

Dado o grafo não direcionado de entrada descrito na Tabela 3.2, onde os nomes identificam os vértices e cada linha uma aresta que liga esses vértices, podemos identificar 3 triângulos: (Bernardo, Chico, Renato), (Renato, Chico, Roberto) e (André, Tamara, Marcelo).

Após a etapa de *Map* do primeiro *job*, são gerados um conjunto de pares (*chave*, *valor*) conforme a Tabela 3.3. A chave é escolhida entre os vértices da aresta, mas isso pode ocorrer de diversas maneiras. Para simplificar, usaremos o primeiro nome em ordem alfabética.

A etapa de *Reduce* do primeiro *job* é a mais importante. Nessa etapa, para cada chave da Tabela 3.3, são gerados as arestas originais (valor na Tabela 3.3) bem como as tríades baseadas nessas arestas. O resultado, já ordenado, pode ser visto na Tabela 3.4.

O segundo *job* tem apenas a etapa de *Reduce* e toma de entrada o resultado produzido na Tabela 3.4. Para cada grupo de chaves (arestas), é verificado se há uma aresta original (Gerado = 0). Se sim, é verificada em quantas tríades essa aresta fecha um tri-

Tabela 3.2. Exemplo de grafo de entrada.

Origem	Destino
André	Bernardo
André	Marcelo
André	Tamara
Bernardo	Chico
Bernardo	Renato
Chico	Renato
Chico	Roberto
Chico	Livia
Marcelo	Tamara
Roberto	Renato

Tabela 3.3. Resultado da etapa de Map do primeiro *Job*.

Chave	Valor (arest)
André	André, Marcelo
André	André, Tamara
André	André, Bernardo
Marcelo	Marcelo, Tamara
Bernardo	Bernardo, Chico
Bernardo	Bernardo, Renato
Chico	Chico, Roberto
Chico	Chico, Livia
Chico	Chico, Renato
Roberto	Roberto, Renato

Tabela 3.4. Resultado da etapa de Reduce do primeiro *Job*.

Chave	Gerado	Tríades
André, Bernardo	0	
André, Marcelo	0	
André, Tamara	0	
Bernardo, Chico	0	
Bernardo, Tamara	1	{André,Bernardo}, {André,Tamara}
Bernardo, Renato	0	
Bernardo, Marcelo	1	{André,Bernardo}, {André,Marcelo}
Chico, Livia	0	
Chico, Roberto	0	
Chico, Renato	1	{Bernardo,Chico}, {Bernardo,Renato}
Chico, Renato	0	
Livia, Roberto	1	{Chico, Livia}, {Chico, Roberto}
Livia, Renato	1	{Chico, Livia}, {Chico, Renato}
Marcelo, Tamara	1	{André,Marcelo}, {André,Tamara}
Marcelo, Tamara	0	
Roberto, Renato	1	{Chico,Roberto}, {Chico,Renato}
Roberto, Renato	0	

ângulo, emitindo uma contagem parcial para cada chave. Pelo nosso exemplo, podemos ver que isso ocorre nas chaves (Chico, Renato), (Roberto, Renato) e (Marcelo, Tamara), totalizando nossos 3 triângulos.

A última etapa (terceiro *Job*) é uma tarefa trivial de somar as contagens parciais emitidas pelo passo anterior.

3.7.5. Hive

O Hadoop MapReduce, como descrito anteriormente na Seção 3.7.4, é um framework para ser utilizado e desenvolvido em uma linguagem de programação, e.g. Java, com as complexidades inerentes de um desenvolvimento de software em linguagens de programação. Assim surgiu rapidamente a necessidade de especificação de programas MapReduce em linguagens declarativas de alto nível.

O Hive [Thusoo et al. 2009] é uma das iniciativas neste sentido, no qual a especificação de *jobs* MapReduce é feita em uma linguagem similar ao SQL: o HiveQL. O Hive foi inicialmente desenvolvido pelo Facebook e atualmente seus principais contribuidores são o próprio Facebook e o Netflix.

O Hive interpreta arquivos do HDFS como tabelas com estrutura, de forma similar a um SGBD. Entretanto, nenhuma estrutura adicional de SGBD é criada pelo Hive, exceto os metadados sobre o formato dos arquivos em tabelas. No HiveQL há uma linguagem de definição destes formatos.

Como um exemplo, imagine que há um arquivo contendo os dados do Twitter no HDFS contendo os dados da tabela do twitter descrita na Seção 3.3 com atributos

follower e *followee*, ou seja, um arquivo texto em formato CSV contendo dois números inteiros (delimitados por vírgula). O comando HiveQL abaixo cria os metadados para a tabela Twitter a partir deste arquivo.

```
CREATE TABLE twitter (follower INT, followee INT);
LOAD DATA INPATH '/input/twitter' OVERWRITE INTO TABLE twitter;
```

Embora o HiveQL não implemente a linguagem SQL em todas as suas funcionalidades, a consulta da Seção 3.3 que resolve o problema da contagem dos triângulos pode ser aplicada diretamente no console do Hive.

É importante notar que o Hive interpreta o SQL e gera *jobs* MapReduce em Java, que são compilados e executados sob demanda.

3.7.6. Pig

De forma similar ao Hive, o Yahoo! criou o Pig [Olston et al. 2008] em 2006 e o moveu como um projeto para o Hadoop em 2007, com o intuito de simplificar o desenvolvimento de *jobs* MapReduce através de uma linguagem procedural de alto nível (Pig Latin).

O Pig Latin, diferentemente do HiveQL, é uma linguagem procedural e não declarativa. É possível efetuar atribuições e manipulação de variáveis intermediárias. Em geral, cada comando PigLatin corresponde a um operador da álgebra relacional.

O exemplo abaixo mostra a declaração da tabela do Twitter na linguagem Pig Latin.

```
Twitter = LOAD '/input/twitter' USING PigStorage(',')
  AS (follower: int, followee: int);
```

Note que, diferentemente do Hive, o Pig Lating cria a tabela Twitter e ao mesmo tempo instancia a mesma lendo do arquivo de input com o formato especificado.

O código abaixo executa a contagem dos triângulos, uma vez mapeada a tabela Twitter.

```
Twitter_1 = LOAD '/input/twitter' USING PigStorage(' ')
  AS (follower:int, followee:int);
Twitter_2 = LOAD '/input/twitter' USING PigStorage(' ')
  AS (follower:int, followee:int);
Twitter_3 = LOAD '/input/twitter' USING PigStorage(' ')
  AS (follower:int, followee:int);

Twitter_SelfJoin = JOIN Twitter_1 by $1,
                        Twitter_2 by $0;

Twitter_FullJoin = JOIN Twitter_3 by ($1, $0),
                        Twitter_SelfJoin by ($0, $3);

Triangles = FILTER Twitter_FullJoin
  BY ($0 < $1 AND $1 < $3) OR ($0 > $1 AND $1 > $3);

Triangles_Grp = GROUP Triangles ALL;
Triangles_Cnt = FOREACH Triangles_Grp GENERATE COUNT(Triangles);

DUMP Triangles_Cnt;
```

As três primeiras linhas atribuem as tabelas Twitter_1, Twitter_2 e Twitter_3. Esta atribuição em três tabelas não deveria ser necessária, mas na versão do Pig (antiga) em que foi criado e executado este código não era possível fazer uma junção da tabela com ela mesma (*self join*).

As próximas duas linhas executam as duas junções; a próxima, um filtro que elimina as duplicatas; e as duas penúltimas executam um agrupamento para contagem. A última linha imprime o valor final. Somente a partir de um comando DUMP que o Pig começa a construir os *jobs* MapReduce, compilar e executá-los.

3.8. Apache Spark

O Apache Spark é uma plataforma para computação distribuída que foi projetada para ser de propósito geral e muito eficiente [Zaharia et al. 2012, Karau et al. 2015]. A principal diferença em relação ao MapReduce é que toda a computação é feita e armazenada em memória, sem necessidade de salvar em disco resultados intermediários.

A unidade básica de dados do Spark é o *Resilient Distributed Dataset* (RDD). O conceito é semelhante ao bloco de dados do HDFS, mas trata-se de coleções de dados que estão na memória RAM dos nós do *cluster*. Na prática, o Spark carrega os dados de um bloco do HDFS na memória RAM do nó em que o bloco está. Os programas Spark fazem dois tipos de operação com um RDD:

- **Transformações**: Transformam um RDD em outro RDD. Entre as transformações mais comuns podemos encontrar as operações de *Map* e *ReduceByKey* (mesmo conceito do MapReduce), ordenação e operações de conjuntos, como união, interseção e diferença.

- **Ações**: Produzem algum resultado a partir de um RDD. Ações típicas consistem em enumerar alguma quantidade de itens de um RDD, contar e somar.

O Spark utiliza o conceito de *execução preguiçosa*, para que só quando realmente um resultado tenha de ser produzido (uma **ação** é executada), e toda a sequência de passos necessária é conhecida, o Spark realmente lê os dados e faz cálculos na memória. Com isso, o motor de execução do Spark consegue otimizar tudo que será executado, escolhendo os melhores nós, a melhor sequência, etc.

O Apache Spark também vem com um conjunto de bibliotecas com algoritmos para aprendizado de máquina, grafos, fluxo contínuo de dados e SQL. Algumas dessas são vistas a seguir.

3.8.1. Utilizando o console python

O Apache Spark possui consoles interativos nas linguagens Python e Scala e seus *jobs* podem ser submetidos em batch também em Java. A API do Spark é acessada a partir de um objeto central denominado `SparkContext`. Esse objeto contém a conexão com uma instância do cluster e a partir dele todos os outros objetos e métodos são acessados. Ao se iniciar um console do Spark, o objeto já está automaticamente disponível através da variável `sc`. Para compreender a simplicidade do modelo de programação do Spark, o trecho de código abaixo lê um arquivo txt e faz a contagem de ocorrência das palavras.

```
#produz um RDD onde cada item e uma linha do arquivo texto
arquivo = sc.textFile('hdfs://servidor:10001/arquivo.txt')

#para cada linha produz N itens no novo RDD, uma para cada
   palavra.
palavras = arquivo.flatMap(lambda linha : linha.split(' '))

#cria novo RDD com tuplas do formato (palavra, 1)
palavrasCV = palavras.map(lambda palavra : (palavra, 1))

#executa o Reduce usando a funcao add para os valores das
   tuplas.
contagemPalafras = palavrasCV.reduceByKey(add)

#somente nesse comando toda a computacao e feita de forma
   otimizada.
contagemPalavras.collect()
```

Para resolver o problema de contagem de triângulos utilizando o Spark, a ideia é semelhante à usada com o modelo do MapReduce.

3.8.2. Spark SQL

O Spark SQL [Xin et al. 2013b, Armbrust et al. 2015], anteriormente chamado de Shark, é um módulo do Apache Spark para processamento de dados estruturados (relacionais).

Ele utiliza uma abstração denominada *DataFrame*, e também serve como uma máquina de execução de consultas distribuídas baseada em SQL.

Um *DataFrame* do Spark SQL tem as mesmas características de um *DataFrame* em R ou em Pandas (Python), e pode ser criado a partir de diversas fontes de dados, como um arquivo JSON, um arquivo texto, um RDD do Spark, uma tabela do Hive, ou qualquer fonte que possua um driver JDBC. O esquema de dados de um *DataFrame* pode ser inferido através de reflexão ou programaticamente.

O trecho de código abaixo mostra como executar a contagem de triângulos utilizando o SQL proposto na seção 3.2.

```
# Importa os modulos e cria o contexto
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)

# Carrega o arquivo de arestas para um RDD
linhas =
   sc.textFile("hdfs://servidor:10001/data/triangles/twitter_combined.txt")
partes = linhas.map(lambda l: l.split())
arestas = partes.map(lambda p: Row(follower=int(p[0]),
   followee=int(p[1])))

# Infere o schema e registra o DataFrame como tabela
schemaWiki = sqlContext.createDataFrame(arestas)
schemaWiki.registerTempTable("arestas")

# Executa o SQL
triangulos = sqlContext.sql("SELECT count(*) FROM arestas R,
   arestas S, arestas T WHERE R.follower = S.followee AND
   S.follower = T.followee AND T.follower = R.followee AND
   R.follower > S.follower AND
     S.follower > T.follower")
print triangulos.collect()
```

3.8.3. Spark GraphX

O Spark GraphX é um novo módulo do Apache Spark que fornece um conjunto de abstrações e ferramentas para processamento paralelo de algoritmos em grafos [Xin et al. 2013a]. Nas abstrações de vértices e arestas é possível incluir propriedades, como pesos, capacidades máximas e mínimas; ou qualquer outra propriedade que seja relevante para modelar um problema.

O Spark GraphX possui um conjunto de operações essenciais para diversos algoritmos de análise de grafos, como operações em paralelo sobre os vértices/arestas dos grafos, obtenção de subgrafos, inversão de arestas e agregação de vizinhos. Esse último, por exemplo, pode ser usado para calcular o grau de cada vértice de um grafo. Mais

detalhes desse módulo podem ser obtidos na documentação oficial do produto [2].

Como é recente seu desenvolvimento, ainda são poucos os algoritmos implementados e a única linguagem suportada é o Scala. Atualmente ele conta com algoritmos de PageRank [Brin and Page 2012], identificação de componentes conectados e contagem de triângulos, que é demonstrada com trecho de código abaixo. Comparado com as estratégias de MapReduce, a contagem de triângulos utilizando Spark GraphX é de várias ordens de grandeza mais rápido e eficiente em consumo de memória.

```
# Carrega o grafo a partir de um arquivo cujas linhas sao pares
   de identificadores dos nos, definindo uma aresta
val graph = GraphLoader.edgeListFile(sc,
   "graphx/data/followers.txt",
   true).partitionBy(PartitionStrategy.RandomVertexCut)

# conta o numero de triangulos
val triCounts = graph.triangleCount().vertices

# imprime o resultado
println(triCounts.mkString("\n"))
```

3.9. Considerações finais

Neste capítulo foram apresentadas algumas tecnologias recentes da era do big data. Entretanto, a lista de tecnologias com certeza não foi exaustiva. Não foram abordadas muitas outras tecnolgias existentes. Como exemplo, não foram citadas as tecnologias de bancos de dados não relacionais (NoSQL) [Han et al. 2011], e a comparação entre estas e o modelo relacional [Cattell 2011, Stonebraker 2010, Stonebraker 2012]. Não é objetivo deste capítulo prover um *survey* sobre todas as possíveis tecnologias de gerenciamento de big data, mas de mostrar de forma introdutória algumas das principais tecnologias já maduras de mercado.

O que também está fora do escopo deste capítulo é a comparação técnica mais profunda e de desempenho entre cada uma das tecnologias [Pavlo et al. 2009], como por exemplo, comparação entre bancos de dados orientados a colunas ou a linhas [Abadi et al. 2008] e a comparação entre modelos de paralelismo Hadoop ou bancos de dados [Stonebraker et al. 2010].

Referências

[Abadi et al. 2006] Abadi, D., Madden, S., and Ferreira, M. (2006). Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM.

[Abadi et al. 2009] Abadi, D. J., Boncz, P. A., and Harizopoulos, S. (2009). Column-

[2] http://spark.apache.org/docs/latest/graphx-programming-guide.html

oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665.

[Abadi et al. 2008] Abadi, D. J., Madden, S. R., and Hachem, N. (2008). Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM.

[Armbrust et al. 2015] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. (2015). Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM.

[Astrahan et al. 1976] Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., Griffiths, P. P., King, W. F., Lorie, R. A., McJones, P. R., Mehl, J. W., et al. (1976). System R: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137.

[Brin and Page 2012] Brin, S. and Page, L. (2012). Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer networks*, 56(18):3825–3833.

[Cattell 2011] Cattell, R. (2011). Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27.

[Chamberlin and Boyce 1974] Chamberlin, D. D. and Boyce, R. F. (1974). Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM.

[Chu and Cheng 2012] Chu, S. and Cheng, J. (2012). Triangle listing in massive networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):17.

[Codd 1970] Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

[Dean and Ghemawat 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[DeWitt and Gray 1992] DeWitt, D. and Gray, J. (1992). Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98.

[DeWitt et al. 1984] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. A. (1984). *Implementation techniques for main memory database systems*, volume 14. ACM.

[Estabrook and Brill 1969] Estabrook, G. F. and Brill, R. C. (1969). The theory of the TAXIR accessioner. *Mathematical Biosciences*, 5(3):327–340.

[Fushimi et al. 1986] Fushimi, S., Kitsuregawa, M., and Tanaka, H. (1986). An overview of the system software of a parallel relational database machine grace. In *VLDB*, volume 86, pages 209–219.

[Garcia-Molina and Salem 1992] Garcia-Molina, H. and Salem, K. (1992). Main memory database systems: An overview. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):509–516.

[Ghemawat et al. 2003] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM.

[Grossman 2012] Grossman, R. (2012). *The structure of digital computing: from mainframes to big data*. Open Data Press, LLC.

[Han et al. 2011] Han, J., Haihong, E., Le, G., and Du, J. (2011). Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE.

[Karau et al. 2015] Karau, H., Konwinski, A., Wendell, P., and Zaharia, M. (2015). *Learning Spark: Lightning-Fast Big Data Analysis*. "O'Reilly Media, Inc.".

[Leskovec and Mcauley 2012] Leskovec, J. and Mcauley, J. J. (2012). Learning to discover social circles in ego networks. In *Advances in neural information processing systems*, pages 539–547.

[Mishra and Eich 1992] Mishra, P. and Eich, M. H. (1992). Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113.

[Olston et al. 2008] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. (2008). Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM.

[Pavlo et al. 2009] Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., and Stonebraker, M. (2009). A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM.

[Plattner 2009] Plattner, H. (2009). A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1–2. ACM.

[Plattner and Zeier 2012] Plattner, H. and Zeier, A. (2012). *In-Memory data management: Technology and applications*. Springer Science & Business Media.

[Schneider and DeWitt 1989] Schneider, D. A. and DeWitt, D. J. (1989). *A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment*, volume 18. ACM.

[Stonebraker 1986] Stonebraker, M. (1986). The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9.

[Stonebraker 2010] Stonebraker, M. (2010). Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11.

[Stonebraker 2012] Stonebraker, M. (2012). Newsql: An alternative to nosql and old sql for new oltp apps. *Communications of the ACM. Retrieved*, pages 07–06.

[Stonebraker et al. 2010] Stonebraker, M., Abadi, D., DeWitt, D. J., Madden, S., Paulson, E., Pavlo, A., and Rasin, A. (2010). Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1):64–71.

[Suri and Vassilvitskii 2011] Suri, S. and Vassilvitskii, S. (2011). Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM.

[Thusoo et al. 2009] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629.

[Wirfs-Brock 2011] Wirfs-Brock, A. (2011). The third era of computing.

[Xin et al. 2013a] Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013a). GraphX: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM.

[Xin et al. 2013b] Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., and Stoica, I. (2013b). Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pages 13–24. ACM.

[Zaharia et al. 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association.

[Zukowski et al. 2006] Zukowski, M., Heman, S., Nes, N., and Boncz, P. (2006). Super-scalar ram-cpu cache compression. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 59–59. IEEE.