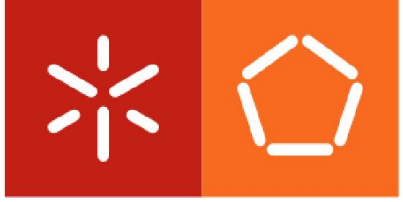
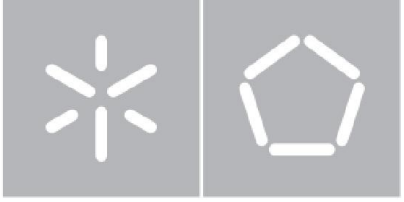


Universidade do Minho
Escola de Engenharia

Dário Almeno Matos da Silva

Inclusão de Funcionalidades MapReduce em Sistemas de Data Warehousing

Novembro de 2013



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Dário Almeno Matos da Silva

Inclusão de Funcionalidades MapReduce em Sistemas de Data Warehousing

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor Doutor Orlando Manuel de Oliveira Belo

Novembro de 2013

À minha família.

Agradecimentos

Este trabalho não estaria concluído sem apresentar os meus mais sinceros agradecimentos a todos aqueles que contribuíram para a elaboração deste trabalho. Agradeço, primeiramente, ao Professor Doutor Orlando Manuel de Oliveira Belo porque, para além de me orientar nesta dissertação, teve toda a sua disponibilidade, interesse e apoio incondicional mas também pela confiança depositada em mim. As suas reflexões e concelhos foram inestimáveis e contribuíram para que esta dissertação tomasse a direção certa. A ele, muito devo. Agradeço ainda ao professor António Sousa por colocar à disposição uma máquina virtual e a sua disponibilidade para a resolução de problemas com esta. Aos membros da família, especialmente os meus pais e irmão, que sempre me apoiaram e me deram a coragem e confiança para continuar, principalmente nos momentos de maior desanimo. A todos, família, amigos e colegar reitero os meus profundos agradecimentos pela motivação e paciência.

Resumo

Inclusão de Funcionalidades MapReduce em Sistemas de Data Warehousing

Em geral, o processo de aquisição de dados nas organizações tornou-se gradualmente mais fácil. Perante a atual proliferação de dados, surgiram novas estratégias de processamento que visam a obtenção de melhores desempenhos dos processos de análise de dados. O MapReduce é um modelo de programação dedicado ao processamento de grandes conjuntos de dados e que coloca em prática muitos dos princípios da computação paralela e distribuída. Este modelo tem em vista facilitar o acesso aos sistemas paralelos e distribuídos a programadores menos experientes, de forma a que estes possam beneficiar das suas características de armazenamento e de processamento de dados. Os *frameworks* baseados neste modelo de programação ocupam hoje já uma posição de destaque no mercado, sobretudo no segmento dedicado à análise de dados não estruturados, tais como documentos de texto ou ficheiros *log*. Na prática, o problema do armazenamento das estruturas multidimensionais de dados e a capacidade de realizar cálculos "*on the fly*", com tempos de execução reduzidos, constituem desafios muito importantes que têm que ser, também, encarados pelos sistemas de *data warehousing* modernos. Com efeito, nas últimas décadas, surgiram técnicas de otimização de desempenho para dar resposta às necessidades mais correntes dos agentes de decisão. O espaço multidimensional é tipicamente sustentado por um sistema de gestão de base de dados relacional através de um esquema em estrela. Igualmente, algumas soluções alternativas a estes sistemas, tal como a Bigtable, e o aparecimento de tecnologias de sistemas de *data warehousing* baseadas em MapReduce, como o Apache Hive e o

Apache Pig, assumem um papel cada vez mais relevante. Nesta dissertação foram analisadas várias técnicas orientadas para a otimização do desempenho de um sistema multidimensional de dados, com base nas características de armazenamento e processamento de *queries* que o MapReduce nos propicia nos dias que correm. Os princípios fundamentais destas técnicas consistem numa estruturação dos dados contidos no *data warehouse*, de forma a facilitar a sua manutenção e usufruir de excelentes desempenhos na satisfação de *queries*, tendo em consideração, contudo, as limitações impostas pelo modelo de programação MapReduce. Adicionalmente, esta dissertação apresenta e descreve um processo de adaptação de uma estrutura convencional de um data warehouse para uma estrutura baseada em MapReduce, analisando os seus aspetos mais pertinentes.

Palavras-chave: Sistemas de suporte à decisão, *Business Intelligence*, *Data Warehousing*, Processamento de estruturas Multidimensionais de dados, *On-line Analytical Processing*, MapReduce.

Abstract

Including MapReduce functionalities in Data Warehousing Systems

In general, the data acquisition process by organizations become gradually easier. Given the current data proliferation, new processing strategies aimed at archiving better performance of data analysis processes. MapReduce is a programming dedicated to processing large data sets and puts into practice many of the principles parallel and distributed computing. This model aims to facilitate access to parallel and distributed systems to less experienced programmers, so that they can benefit from their storage characteristics and data processing. Frameworks based on this programming model today already occupy a prominent position in the market, especially in the segment devoted to the analysis of unstructured data such as text documents or log files. In practice, the problem of storage of multidimensional data structures and the ability to perform on-the fly calculations, with reduced execution time, are very important challenges that must also faced by modern data warehousing systems. Indeed, in recent decades, emerged techniques for performance optimization to meet the most common needs of the decision makers. The multidimensional space is typically supported by a relational database management system through a star schema. Also, some alterative solutions to these systems, such as Bigtable, and the emergence of data warehousing systems technologies based on MapReduce, such as Apache Hive e Apache Pig are playing an increasingly important role. This dissertation analyzed several techniques aimed at optimizing the performance of a system of multidimensional data, based on characteristics of storage and query processing in the MapReduce provide these days. The fundamental principles of these techniques consist of a structure of data in the data warehouse, in order to facilitate their management and boasts excellent performance in satisfying queries, taking

account, however, the limitations imposed by the MapReduce programming model. Additionally, this dissertation introduces and describes an adaptation process of a conventional data warehouse structure for a framework based on MapReduce, analyzing its most relevant aspects.

Keywords: Decision Support Systems, Business Intelligence, Data Warehousing, Multidimensional Data Structures Processing, On-line Analytical Processing, MapReduce.

Índice

Índice de Figuras

Índice de Tabelas

Capítulo 1

Introdução

1.1 Exploração de sistemas de *data warehousing*

Facilmente se verifica que várias organizações e empresas trabalham no seu quotidiano com grandes volumes de dados. Todas elas pretendem, de alguma maneira, extrair, armazenar, pesquisar, partilhar e analisar dados. Porém, hoje, enfrentam um problema de escala. Cada vez mais, os dados estão descentralizados e com tendência a aumentar de forma significativa. Quanto maior for a dinâmica da organização ou a sua necessidade de crescimento, maior é (e mais se incentiva) a coleta de novos dados. Perante esta proliferação de dados são necessárias novas estratégias de processamento que visem a obtenção de melhores resultados, de forma mais efetiva e cujas *queries* sejam satisfeita de forma muito expedita (um alto nível de desempenho).

Nas duas últimas décadas, as tecnologias relacionadas com a *Business Intelligence* (BI) têm crescido muito rapidamente, tanto no número de produtos e serviços oferecidos como na sua adoção por organismos empresariais. Aparentemente, tal pode ser justificado pela queda dos custos relacionados com a aquisição e armazenamento dos dados. Atualmente, as organizações e as empresas recolhem os dados numa granularidade mais fina, o que produz um volume de dados considerável. Estas têm implantado e testado, de forma agressiva, algumas técnicas de análise mais sofisticadas para o processo de tomada de decisões, de forma a orientar os seus serviços de acordo com os gostos ou necessidades dos seus clientes. Hoje, dificilmente se encontram

empresas de sucesso que ainda não tenham tirado partido desta tecnologia para o seu negócio de alguma forma (Chaudhuri et al., 2011).

Uma das necessidades que organizações usualmente partilham é o cruzamento da informação de que dispõem para responder a um conjunto de questões. Ao realizar tais relacionamentos entre os dados, é possível obter observações ou perspetivas importantes para os agentes que realizam análises sobre os negócios de uma empresa. Assim, é necessário um repositório – geralmente designado por *Data Warehouse* (Inmon, 2002) – com a capacidade de centralizar os dados provenientes de várias fontes, de forma a ser possível cruzar os dados de toda a organização, sem perturbar os seus sistemas operacionais. Obter rapidamente estas análises, através de questões colocadas de forma *ad hoc*, permite ao agente de decisão desenvolver os seus processos de decisão de forma sustentada. O acesso em tempo útil à informação é também essencial para que empresas tomem as decisões certas, no momento adequado.

Um sistema de *data warehousing* (SDW) consiste num conjunto de técnicas que dá suporte aos utilizadores no processo de tomada de decisões e enriquece o repositório de informação sobre as suas atividades de negócio. Entre os vários sistemas de suporte à decisão (SSD), o SDW é um dos que tem recebido mais atenção por parte das comunidades académica e industrial. Este sistema é composto por processos de extração periódica de dados dos sistemas operacionais e processos de transformação de dados operacionais em informação útil, que descreva o negócio, para dar resposta às necessidades de analistas, diretores e gestores, tais como informação concisa, acessível, e como tal permitir diferentes perspetivas de análise sobre as suas atividades. O *Data Warehouse* (DW) tem um papel central nestes processos.

Cada vez mais, as empresas optam por implementar SDWs, bem como ferramentas de processamento analítico para a exploração de dados (Chaudhuri and Dayal, 1997). A flexibilidade e o desempenho no acesso à informação da organização fazem deste sistema a escolha predileta dos analistas e gestores. Uma estrutura multidimensional permite criar e suportar diversos pontos de vista do negócio. Porém, o cálculo frequente de agregações sobre uma grande quantidade de dados históricos, que vulgarmente se realizam sobre estas estruturas, é dispendioso, tanto em termos de espaço de armazenamento como em termos de processamento de dados. Nos ambientes de DW tradicionais, a redução do tempo de resposta pode ser obtido a partir de várias técnicas de otimização. Por exemplo, a utilização de vistas materializadas baseada nas dimensões

– para o esquema da Figura 11 – permite responder a *queries* mais rapidamente, como as que usualmente se definem para calcular o valor total de vendas de um produto para um determinado ano. Tudo isto acontece assim, uma vez que os resultados a estas consultas são pré-calculados e armazenados em tabelas separadas. No entanto, para realizar a seleção de alguns parâmetros que não foram pré-calculados, tal como o valor total de vendas de um produto para um determinado dia, a resposta não pode ser fornecida com a mesma rapidez, já que o processamento da *query* é executado com base numa tabela de factos base, existindo um limite – o espaço em disco – para o número de dimensões que podem ser materializadas (O'Neil and Quass, 1997). O DW não teria o mesmo desempenho, caso estas otimizações não fossem realizadas. As operações de *full table scaning* devem ser evitadas, uma vez que consumem demasiado tempo e recursos, em situações em que apenas se exija o acesso a um número reduzido de registos.

1.2 Otimização de desempenho

Os sistemas *On-line Analytical Processing* ou OLAP (Codd et al., 1993) – também designados por sistemas de processamento analítico – proporcionam aos decisores a possibilidade de explorar os dados de acordo com as suas diferentes perspetivas do negócio e realizar novas análises a partir de resultados obtidos através das várias *queries* que vão lançando. Estes sistemas organizam os dados de forma multidimensional (Chaudhuri and Dayal, 1997), o que facilita a análise e a visualização destes dados num DW. Por exemplo, um DW de vendas com as dimensões de interesse "tempo" e "produto", organizadas pelas hierarquias: "dia -> mês -> ano" e "produto -> subcategoria -> categoria", respetivamente. Esta organização dos dados faculta a aplicação das operações típicas OLAP, que são: o *roll-up* (aumenta o nível de agregação ou diminui o detalhe); o *drill-down* (diminui o nível de agregação ou aumenta o detalhe) entre as hierarquias das dimensões (Figura 1); o *slice and dice* (seleção e projeção); e o *pivot* (reorganiza os dados multidimensionais de forma a criar uma vista nova).

Tradicionalmente, as bases de dados relacionais têm vindo a suportar os DWs. Os requisitos funcionais e de desempenho de um DWs são bastantes diferentes das aplicações *On-Line Transactional Processing* (OLTP), normalmente também suportadas por bases de dados relacionais. Numa aplicação OLTP, as transações são curtas e isoladas, limitando-se a ler ou atualizar umas dezenas de registos, acedidas tipicamente através das chaves primárias. As bases de dados operacionais são projetadas para maximizar a taxa de transações (a métrica principal do

desempenho), sem perder a consistência e a integridade dos dados. Estas bases de dados – ajustadas para suportar uma carga de trabalhos OLTP – não são apropriadas para a execução de consultas OLAP complexas, uma vez que requereriam desempenhos um pouco intoleráveis para as máquinas que as acolhem. Ademais, usualmente, as bases de dados operacionais não mantêm um histórico, essencial para perceber tendências, armazenam apenas dados atuais. Para se conciliar várias fontes de dados heterogéneas – entre as quais as bases de dados operacionais – e suportar os modelos multidimensionais de dados, bem como as operações típicas OLAP, exige-se uma implementação separada dos sistemas operacionais (Chaudhuri and Dayal, 1997).

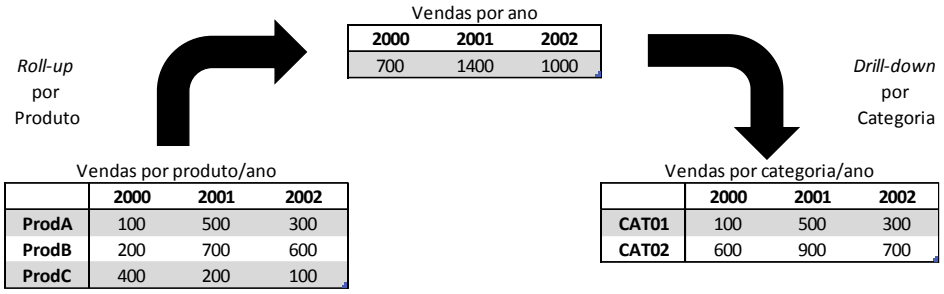


Figura 1: Ilustração das operações de *roll-up* e de *drill-down*

Os DWs implementados em bases de dados relacionais armazenam os dados estruturados segundo o modelo relacional. Estas bases de dados devem ser capazes de executar *queries* complexas em SQL[1] da forma mais eficiente, sobre conjuntos de dados muito grandes. A produtividade dos agentes de decisão depende essencialmente dos tempos de resposta às interrogações lançadas sobre os sistemas OLAP disponíveis. Ao longo das últimas duas décadas, os sistemas de gestão de bases de dados relacionais desenvolveram estruturas de dados, otimizações e técnicas de processamento de *queries* para a execução de *queries* complexas SQL sobre grandes volumes de dados (Chaudhuri et al., 2011). Como as *queries* analíticas consomem muito tempo e recursos, torna-se necessário investir em mecanismos otimizados de maneira a melhorar o seu desempenho e contribuir para a redução dos tempos de resposta às *queries*.

[1] ***Structured Query Language***: Uma linguagem de pesquisa estruturada que permite gerir e realizar consultas sobre os dados armazenados em bases de dados.

1.2.1 O armazenamento de dados

Modelação

O armazenamento dos dados cumpre uma função importante na otimização do desempenho, continuando a ser, ainda hoje, um dos grandes desafios em DW. As *queries* típicas de suporte à decisão requerem operações, como agregações, junções e seleção. A fim de suportar de forma eficiente estas operações, algumas estruturas para armazenar os dados, de forma redundante, foram desenvolvidas em bases de dados relacionais. O esquema em estrela é o mais utilizado para reproduzir o modelo de dados multidimensionais. O desenho deste modelo de dados é constituído por uma tabela de factos e um conjunto de tabelas dimensão (uma tabela por dimensão). Existem variantes mais complexas do esquema em estrela que configuram o modelo de dados multidimensional, como é o caso do esquema em floco de neve. Este esquema normaliza (uma técnica habitualmente utilizada em sistemas OLTP) as tabelas dimensão de forma a que as hierarquias possam ser representadas explicitamente. Deste modo, o custo da manutenção destas hierarquias torna-se menor, uma vez que as atualizações dos dados são efetuadas de forma mais rápida - mais à frente estes esquemas e suas implicações serão discutidos (Secção 3.1.).

Linhas *versus* Colunas

As bases de dados relacionais armazenam os dados em estruturas de dados com duas dimensões (tabelas), em que as linhas representam as instâncias de uma dada entidade e as colunas representam os atributos que descrevem essa entidade. Por exemplo, a entidade "Cliente" – uma entidade bastante comum em bases de dados que armazenam informação de um negócio na área de retalho – pode ser representada por uma tabela, em que cada coluna armazena um determinado atributo relacionado com a entidade cliente ("nome", "idade", "email", etc.) e em que cada linha armazena a informação acerca de um cliente. A estrutura de "duas dimensões" subsiste apenas ao nível lógico. No nível físico, a informação de uma tabela é mapeada para uma dimensão antes do seu armazenamento, já que os dispositivos de armazenamento de dados (ex. discos rígidos) fornecem apenas uma dimensão (Abadi, 2008).

Existem duas formas claras de mapear a informação de uma tabela para uma dimensão: armazenar a tabela linha a linha ou armazenar a tabela coluna a coluna. A abordagem linha a linha mantém a informação acerca de uma entidade toda junta. Para o exemplo dos clientes, esta abordagem armazena toda a informação relativa ao primeiro cliente, em seguida armazena toda a informação relativa ao segundo cliente, e assim por diante. A abordagem coluna a coluna mantém

toda a informação respeitante a um dado atributos toda junta: todos os nomes dos clientes serão armazenados consecutivamente, em seguida as idades, etc. A Figura 2 representa os dois tipos de armazenamento físico dos dados: o armazenamento linha a linha (Figura 2.a); o armazenamento coluna a coluna (Figura 2.b).

nome	idade	email
CL01	18	CL01@mail.com
CL02	22	CL02@mail.com
CL03	30	CL03@mail.com

(a) linha a linha

nome	CL01	CL02	CL03
idade	18	22	30
email	CL01@mail.com	CL02@mail.com	CL03@mail.com

(b) coluna a coluna

Figura 2: Armazenamento físico dos dados

Estas duas formas de armazenamento apresentam configurações que são aceites usualmente e tipicamente constituem uma escolha baseando-se no desempenho esperado. Se a grande carga de trabalho do sistema de exploração está associada com o acesso a dados ao nível da entidade (ex. pesquisar um cliente, adicionar um cliente, remover um cliente), então o armazenamento linha a linha é preferível, uma vez que toda a informação necessária estará armazenada conjuntamente e como tal de mais rápido acesso. Por outro lado, se a carga de trabalho tende a ler poucos atributos por *query* a partir de muitos registos (ex. a média de vendas de um ano), então o armazenamento coluna a coluna é claramente preferível já que todos os atributos que não entram no processo de consulta não serão acedidos.

A maioria dos SGDB, incluindo os mais populares, apresenta uma configuração de armazenamento linha a linha. O desenho implementado por estes produtos provém da pesquisa realizada desenvolvida nos anos 70. Este desenho está otimizado para as aplicações das bases de dados mais comuns da altura: processamento de dados transacionais (aplicações OLTP). As aplicações mais comuns destas bases de dados (ex. transferir uma quantia monetária de um cliente A para um cliente B, em apenas uma transação) executam um conjunto de operações – a execução vista como atómica – recorrendo a consultas de dados sobre uma entidade (ex. encontrar cliente A ou encontrar conta do cliente A). Este mecanismo de armazenamento apresenta uma eficiência superior nas operações de escrita, pois os valores relativos a um dado tuplo podem ser escritos no disco com um número muito menor de operações de escrita (os atributos são escritos de forma

adjacente). Devido a este tipo de trabalhos, a abordagem para o armazenamento de dados linha a linha foi escolhida para estes sistemas.

Por volta dos anos 90, as organizações começaram a usar as bases de dados para analisar o seu negócio. Os DWs tornaram-se comuns no suporte a processos de tomada de decisão e consequentemente na satisfação de *queries* analíticas. A natureza destas *queries* é diferente das empregadas em bases de dados transacionais. As *queries* analíticas têm tendência a ser menos previsíveis, isto é, as *queries* em DW exploram os dados através de questões *ad hoc* colocadas no momento, ao contrário das *queries* que correm em aplicações OLTP, em que estas normalmente representam operações sobre os dados predefinidas como a inserção de um cliente. Estas *queries* caracterizam-se também por se focarem mais em determinados atributos de diferentes entidades do que propriamente numa entidade. Numa *query* como "Qual o valor total das vendas realizadas ao cliente A?", os atributos são selecionados das entidades "Cliente" e "Venda", centrando-se essencialmente no valor das vendas, que será agregado através de uma operação de somatório (Abadi, 2008).

A estratégia de armazenamento dos dados linha a linha deixou de ser uma decisão óbvia para armazenar os dados de um DW. As características das *queries* destes sistemas, como as leituras intensivas dos dados a alguns atributos em particular, indicam que o armazenamento orientado à coluna é o mais adequado, mesmo sabendo-se do seu desempenho menos adequado no que diz respeito a operações de escrita. Por exemplo, uma inserção de um tuplo numa base de dados orientada à coluna implica "partir" os tuplos de forma a separar os seus atributos, para que cada um delas possa ser escrito independentemente. No entanto, esta ineficiência pode ser moderada, caso se pretenda escrever um conjunto de tuplos numa única ação.

Um DW tem propensão para receber um número muito superior de *queries* de leitura, focadas em atributos, do que receber *queries* de escrita ou mesmo atualizações. O armazenamento orientado à coluna tende a favorecer a execução de consultas analíticas, uma vez que as *queries* acedem apenas aos atributos que necessitam para o cálculo, o que se traduz em operações de I/O mais eficientes. Caso a mesma consulta seja executada em bases de dados orientadas à linha, então todos os tuplos serão carregados para a memória com os atributos desnecessários para o cálculo (que ainda implicaria também uma tarefa de filtragem de atributos). Tal a custos adicionais de I/O que, como sabemos, devem ser evitados sempre que possível (Chaudhuri et al., 2011).

Compressão de dados

Um DW de grande dimensão pode beneficiar significativamente caso se recorra a alguma técnica de compressão de dados, uma vez que os dados comprimidos permitem (Chaudhuri et al., 2011):

- Uma redução de I/O no cálculo de uma *query*, ou seja, os dados comprimidos representam um volume de dados menor a consultar, diminuindo, assim, o número de consultas a disco.
- Fazer uso da memória de forma mais eficiente, ou seja, como a dimensão dos dados é menor, é possível assim manter mais dados em memória.
- Reduzir o espaço de armazenamento solicitado por uma base de dados e, consequentemente, a redução dos custos de armazenamento e de manutenção.
- Diminuir a quantidade de dados a transferir através de uma rede. Esta característica é importante para sistemas com bases de dados paralelas, nos quais os dados se movimentam frequentemente entre os nodos.
- Executar algumas *queries*, sem recorrer à descompressão.

A motivação principal para empregar esta técnica é a melhoria do desempenho do sistema. O custo do espaço em disco é normalmente baixo, verificando-se atualmente uma tendência a descer cada vez mais. Uma taxa de compressão (*compression ratio*) elevada – obtida através de esquemas de compressão, tais como a codificação de Huffman (Huffman, 1952) ou a codificação de Lempel-Ziv (Ziv and Lempel, 1978) – não se adaptam ao desempenho desejado, pois os dados comprimidos de forma excessiva originam descompressões demoradas. O *hardware* tem sido também alvo de melhorias significativas. Todavia este crescimento – do desempenho e da capacidade – não é repartido, de forma equilibrada, pelos principais componentes de um sistema. A velocidade de CPU tem crescido acentuadamente e continua a ultrapassar as velocidades de acesso ao disco e à memória. Este desajuste favorece o uso da compressão. Os esquemas de compressão que renunciam a taxas de compressão elevada – os esquemas de compressão de dados mais "leves" – detêm os tempos de descompressão mais eficientes (um *overhead* menor), de tal modo que reduz os custos de CPU para a compressão e descompressão dos dados (Boncz et al., 1999).

Uma *query* capaz de operar diretamente sobre os dados comprimidos requer modificações no plano de execução. Durante o cálculo de uma *query*, o conhecimento do esquema de compressão aplicado aos dados é necessário para se adaptar ao plano de execução e, assim, ter a possibilidade de processar adequadamente os dados. As primeiras bases de dados descompactavam os dados que se encontravam comprimidos em disco, imediatamente antes de os transferir para a memória. Esta abordagem é ineficiente, uma vez que as leituras dos dados para a memória exigiam uma descompressão, mesmo quando não fossem utilizados. Em (Graefe and Shapiro, 1991) os atributos são comprimidos individualmente – seguindo o mesmo esquema de compressão para todos os atributos relativos a um domínio – e mantidos em memória. Os dados são descompensados, caso seja necessário para fazer o seu manuseamento. As operações, como o *hybrid hash join*, utilizam os processos de compressão de dados para aumentar o seu desempenho, já que mantêm, em memória, mais dados relativos à tabela, reduzindo o número de acessos ao disco (Abadi et al., 2006).

A forma como os dados são armazenados é um fator relevante para a compressão. Os dados armazenados coluna a coluna permitem compressões mais eficientes do que quando armazenados linha a linha. Tipicamente, o armazenamento coluna a coluna concentra, em disco, os valores de um atributo com o mesmo tipo de dados. Portanto, esta abordagem possibilita comprimir vários valores de uma só vez. Já o armazenamento linha a linha "mistura" vários atributos num único tuplo, ou seja um atributo está intrinsecamente ligado a um tuplo e rodeado por mais atributos, possivelmente de domínio diferente. Assim, o conjunto de valores de um atributo não se encontram armazenados de forma contígua. Isso dificulta a implementação de um esquema de compressão de dados.

A técnica *dictionary compression* é aplicada aos dados armazenados linha a linha. Esta técnica identifica valores repetidos nos dados e constrói um dicionário de forma a mapear estes valores por representações mais condensadas (ex. mapear o valor do atributo nome da tabela dimensão cliente "Maria", um nome bastante comum, por um inteiro). Já para o armazenamento coluna a coluna, a técnica RLE (*run-length encoding*) é mais atrativa, uma vez que permite comprimir vários valores repetidos através de um único par. Numa compressão RLE, uma sequência de valores *v*, com o tamanho *n*, pode ser expressa pelo par (*v*, *n*). Esta técnica é bastante eficiente, caso ocorram sequências grande de um mesmo valor para um dado atributo (com poucos valores

distintos) ou o atributo se encontrar ordenado. Todavia, estabelecer uma ordem para os valores não é uma tarefa trivial (Chaudhuri et al., 2011).

Índices, Vistas Materializadas e Particionamento de dados

As estruturas de acesso aos dados, como os índices ou vistas materializadas, são partes importantes para um DW. A definição e seleção de índices, juntamente com as vistas materializadas, são temas de interesse de pesquisa em DW, já que constituem as técnicas mais eficazes para melhorar o desempenho do sistema (Golfarelli and Rizzi, 2009). Mais à frente serão discutidas estas técnicas com mais detalhe, bem como implementações de índices e de particionamento de dados. (Secção 3.2.).

A consulta de uma estrutura como um índice pode tornar o processamento das *queries data-selective* mais eficientes, uma vez que um índice permite o acesso associativo com base nos valores de um atributo. A seleção de tuplos, através de uma *query* com uma ou várias condições de seleção (ex. idade <25), tem a possibilidade de explorar os índices para melhorar o seu desempenho. As *queries* com várias condições podem beneficiar do *index intersection*. Já as *queries* com uma condição de seleção beneficiam de um *index scan* ou uma consulta a um índice. Estas operações reduzem significativamente ou mesmo eliminar a necessidade de aceder à tabela base (Chaudhuri and Dayal, 1997).

A pré-computação e armazenamento dos dados – as vistas materializadas – também aceleram o tempo de resposta às *queries* de um DW. O cálculo de agregações, por exemplo as vendas semanais de um produto, é muito frequente em DWs. Esta técnica tem a capacidade de devolver o resultado de determinadas consultas sem realizar cálculos adicionais, apenas se limita a "varrer" os tuplos de uma tabela com o resultado pré-calculado. No entanto, uma *query* imprevista não beneficiará de uma vista materializada e, assim, a aplicação desta técnica é limitada. Os índices refletem o oposto, uma vez que são estruturas mais abrangentes. Todavia o desempenho de um índice não é tão dramático como o desempenho de uma vista materializada. O plano físico permite combinar estas duas técnicas (Chaudhuri et al., 2011).

1.2.2 O Processamento de *queries*

Normalmente, os DWs são extremamente grandes e requerem bastantes recursos para dar resposta às análises que sobre eles são realizadas, em tempos aceitáveis. O paralelismo (DeWitt and Gray, 1992) desempenha um papel importante no processamento de *queries* sobre bases de dados volumosas (juntamente, obviamente, com os índices e as vistas materializadas), já que responde satisfatoriamente à necessidade de manipular conjuntos de dados grandes, de forma eficiente. A evolução do conceito de DW, de um repositório local e centralizado para um contexto mais amplo de partilha e de análise de dados num mundo ligado à Internet, deu origem a novas abordagens e novas formas de distribuição. A implementação de um sistema paralelo e distribuído pode fornecer um melhor desempenho e escalabilidade (Furtado, 2011).

Frequentemente, os DWs grandes são implementados por sistemas paralelos, devido à sua grande necessidade de altos desempenhos e de armazenamento de dados. Em 1986, Michael Stonebraker identificou três arquiteturas básicas para o paralelismo (Stonebraker, 1986): *shared-memory*, *shared-disk* e *shared-nothing*. Os três elementos básicos dos sistemas paralelos e que ditam o desempenho – o processador (P), a memória (M) e o disco (D) – ajudam a descrever estas arquiteturas apresentadas (Figura 3): a arquitetura *shared-memory* ou *shared everything* (a), em que todos os processadores partilham o acesso a uma memória em comum e a todos os discos; a arquitetura *shared-disk* (b), na qual cada processador tem uma memória dedicada, mas partilha o disco com todos os outros processadores; e a arquitetura *shared-nothing* (c), em que cada processador tem uma memória e um disco particular (tipicamente uma *commodity machine*).

Num sistema *shared-memory* ou SMP (*Symmetric Multi-Processing*), todos os processadores partilham apenas uma memória e uma coleção de discos. Tipicamente, estas máquinas têm um número pequeno de processadores. A principal vantagem deste sistema consiste em utilizar o mesmo paradigma de programação empregue nas máquinas com um processador. Assim, o sistema operativo enfrenta alguns problemas de concorrência, como por exemplo a distribuição dinâmica de tarefas pelos processadores (partilha de memória e balanceamento de carga), que resultam das várias execuções em paralelo. Infelizmente, os sistemas *shared-memory* têm limitações de escalabilidade fundamentais. Todos os pedidos de I/O e de memória têm de ser transferidos sobre o mesmo barramento que todos os processos partilham, o que conduz a um "congestionamento" rápido da largura de banda deste barramento.

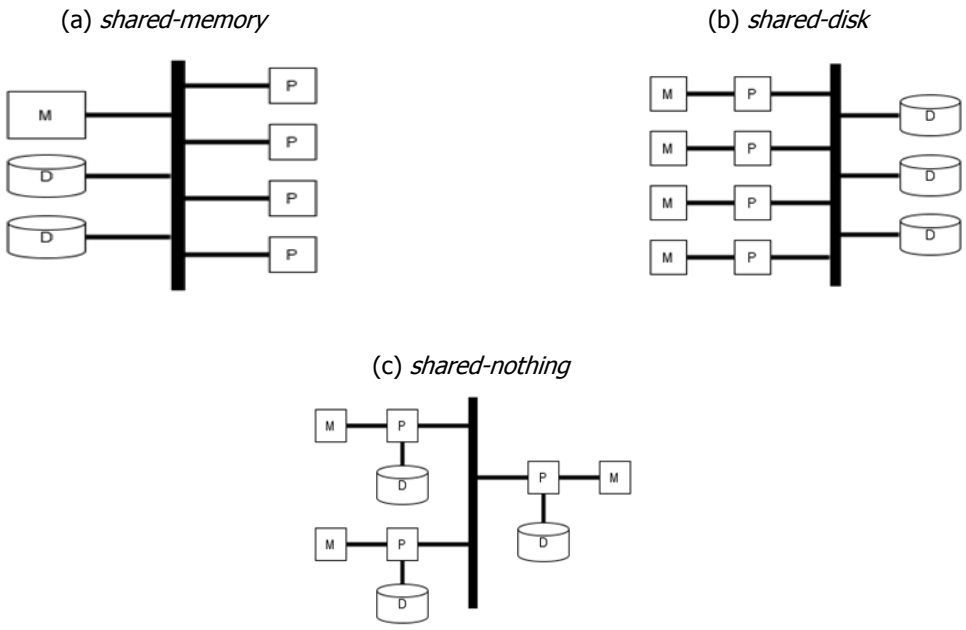


Figura 3: Arquiteturas básicas para o paralelismo

As arquiteturas *shared-nothing* são constituídas por um conjunto de nodos independentes, cada um com memória e armazenamento persistente. Cada nodo pode ser constituído por um ou mais processadores e dispositivos de armazenamento. A comunicação entre os nodos é realizada através de mensagens através da rede. Esta arquitetura permite interligar um grande número de nodos, já que estes não partilham recursos (exceto a rede). A principal desvantagem das arquiteturas *shared-nothing* é o modelo de programação. As trocas de dados constantes de dados entre os nodos (ex. aceder a discos não locais) podem provocar um "congestionamento" no sistema de comunicação. As questões de alocação de dados, otimização do processamento de consultas e balanceamento de carga são as mais evidentes em *shared-nothing*. Estes sistemas podem variar entre *clusters* pequenos, construídos a partir de *commodity machines*, até clusters complexos de alto desempenho, construídos a partir de algumas ou todas as máquinas *shared-memory* (Furtado, 2011).

Nos sistemas *shared-disk*, cada processador dispõe de uma memória dedicada e tem acesso direto a qualquer disco, através de uma rede. Ao contrário dos sistemas *shared-nothing*, em que cada nodo armazena parte do conjunto de dados que lhe foi atribuída, os sistemas *shared-disk* não

exigem uma partição dos dados prévia, já que os dados armazenados são partilhados (*shared storage*). Durante o processamento de uma *query*, os dados não necessitam de se deslocar através dos nodos. Ademais, o balanceamento de carga é relativamente simples, desde que todos os nodos se encontrem disponíveis para servir qualquer pedido. No entanto, existem fatores que limitam a escalabilidade, como: os nodos precisam de comunicar com o objetivo de assegurar a consistência dos dados (tipicamente implementado por um *distributed lock manager*, que pode incorrer em *overhead* não-triviais), ou a rede suportar todas as leituras e escritas dos processadores (Chaudhuri et al., 2011).

Um sistema paralelo ideal apresenta duas características: o dobro do *hardware* pode executar uma tarefa em metade do tempo (*linear speedup*) e o dobro do hardware pode executar tarefas com o dobro da dimensão, no mesmo intervalo de tempo (*linear scaleup*). Os sistemas *shared-nothing* (ex. Teradata[2]) têm o potencial de escalar até volumes de dados muito superiores do que os sistemas *shared-disk* e *shared-memory*, uma vez que os sistemas *shared-nothing* alcançam *speedups* e *scaleups* próximos do linear em *queries* complexas (DeWitt and Gray, 1992). Todavia, a decisão de como efetivamente espalhar os dados através dos nodos é decisivo para o desempenho e escalabilidade (Chaudhuri et al., 2011). Um sistema escalável denota que pode adaptar-se ou escalar para acomodar mais *hardware*, sem realizar grandes alterações. Esta característica é essencial, dado que os dados contidos num DW nunca (ou raramente) são removidos.

1.3 A Inclusão de MapReduce

As bases de dados relacionais mantêm uma posição de destaque em SDW no armazenamento e nas consultas dos dados. Desde há muito que estas possuem estruturas de dados e técnicas de processamento de *queries* para a execução de consultas complexas em SQL sobre grandes volumes de dados. Algumas das arquiteturas atuais de suporte ao BI já incluem o MapReduce, que foi inicialmente desenvolvido para fazer análises de documentos web e pesquisas sobre ficheiros de log, e para suportar as consultas semelhantes ao SQL em ambientes de DW tradicionais (Chaudhuri et al., 2011). Hoje, já existem plataformas baseadas em sistemas de ficheiros

[2] http://www.teradata.com

distribuídos, utilizando o MapReduce para o processamento de dados, implementadas em *clusters* com um número de nodos superior aos encontrados tradicionalmente nos SGDB paralelos. Estas plataformas são arquiteturas *shared-nothing*, capazes de enfrentar problemas complexos, como a tolerância a falhas e o particionamento de dados.

As plataformas MapReduce têm a capacidade de processar dados não estruturados (ex. ficheiros de texto ou imagens) através da execução de duas funções (*map* e *reduce*), de forma escalável. Apesar de se encontrar numa fase inicial em relação aos SGDB paralelos mais "maduros", as ferramentas baseadas em MapReduce têm já recursos para dar resposta às empresas (ex. Cloudera[3]). A plataforma MapReduce tem crescido rapidamente devido à disponibilidade do código da ferramenta Hadoop[4]. O Hive (Thusoo et al., 2009), uma implementação *open source* de um DW sobre o Hadoop, desenvolveu uma linguagem semelhante ao SQL, denominada HiveQL, que compila automaticamente numa sequência de trabalhos MapReduce. Portanto, a produtividade dos utilizadores aumenta enquanto exploram as vantagens desta plataforma. Alguns Sistemas de gestão de bases de dados paralelos, como o Aster Data[5], permitem invocar funções MapReduce sobre os dados armazenados na base de dados como parte de uma *query* SQL. Isto significa que a função MapReduce surge na *query* como uma tabela que permite os seus resultados serem compostos com os restantes operadores SQL da *query* (Chaudhuri et al., 2011). Habitualmente, as cargas de trabalho provocadas por processo de análise de dados resultam muito da realização de *star joins*, que usualmente envolvem muitas operações de *scan* e agregações multidimensionais, de grandes dimensões. Estas operações podem ser executadas em paralelo, por arquiteturas *shared-nothing* (ex. Teradata). Assim, como a quantidade de dados estruturados para análise continua a crescer, o MapReduce apresenta-se como adequado para o processamento deste tipo de análises, apesar de, inicialmente, ter sido desenhado para processar dados não estruturados (Abouzeid et al., 2009).

Para além da escalabilidade, o MapReduce oferece mais características desejáveis para um SDW. Um sistema de gestão de bases de dados (SGBD) tolerante a falhas, no contexto da análise de dados, significa que não existe a necessidade de reiniciar uma *query*, caso um dos nodos do

[3] http://www.cloudera.com/
[4] http://hadoop.apache.org/
[5] http://www.asterdata.com/

cluster envolvidos no processamento da *query* porventura falhe. Já para as cargas de trabalho transacional, um SGBD tolerante a falhas denota a capacidade de recuperar de uma falha, sem perder dados ou atualizações das transações recentemente terminadas (*commit*). Para as *queries* somente de leitura nas cargas de trabalho analíticas, não existe transações de escrita a perder, em caso de falha (Abouzeid et al., 2009).

Atualmente, dado que o número de nodos que compõem os *clusters* tem tendência para aumentar, a probabilidade de um nodo falhar durante o processamento de uma *query* está, assim, também a aumentar rapidamente. Esta probabilidade tende a aumentar com a escala, já que com o acréscimo de dados necessários para a satisfação das *queries* analíticas, mais nodos são necessários para intervir no processamento da *query*. Por exemplo, a Google regista uma média de 1,2 falhas por cada trabalho de análise (Dean and Ghemawat, 2004).

Tradicionalmente, as bases de dados paralelas assumem que as falhas dos nodos é um evento raro e que os *clusters* de maior dimensão se traduzem em dezenas de nodos. Estas assunções resultaram em decisões que dificultam a tolerância a falhas. Já o MapReduce deteta e atribui, em *runtime*, as tarefas dos nodos que falharam a outros nodos do *cluster* (preferivelmente a nodos com réplicas dos dados de input dos nodos que falharam). Com a introdução de nodos redundantes, o MapReduce tem a possibilidade de atribuir mais tarefas *"on the fly"* aos nodos mais rápidos. As tarefas, que demoram mais tempo a concluir, executam em redundância por nodos que já tenham terminado as suas tarefas. O mesmo conjunto de dados é processado em paralelo. O MapReduce cria também *"checkpoints"* (o *output* da função *map*) nos discos locais dos nodos com o objetivo de minimiar a quantidade de trabalho que tem de ser se repetir, em caso de falha (Abouzeid et al., 2009).

1.4 Motivação e objetivos

Atualmente, qualquer organização recolhe enormes volumes de dados para suportar as suas decisões. Tal atividade provoca grandes aumentos na dimensão de qualquer DW e requer processos de satisfação de *queries* bastante eficientes. Um DW de dimensão elevada dificulta o processamento analítico em tempo útil. Com efeito, a distribuição e o processamento destes conjuntos de dados por várias máquinas tem sido adotado por estas organizações e empresas para dar resposta aos desafios atuais dos SDW. O tamanho do conjunto de dados dita que estes sejam

armazenados e processados em sistemas altamente paralelos, como os *clusters shared-nothing*. As bases de dados paralelas oferecem uma solução para o armazenamento e uma linguagem de pesquisa simples, ocultando os detalhes de um *cluster* físico. No entanto, estas soluções têm um preço muito elevado em escalas web (Olston et al., 2008).

A inclusão de um *framework* baseado em MapReduce – o Hadoop – deve ser considerada em SDW para minimizar os problemas decorrentes de um processamento analítico complexo. Este trabalho de dissertação teve a intenção de estudar o modelo de programação MapReduce e construir uma estrutura de dados adequada para suportar um DW típico através de MapReduce. Um programa MapReduce executa, essencialmente, um agrupamento seguido de agregações em paralelo por um *cluster* de máquinas. O utilizador tem apenas de definir duas funções (através de uma linguagem de programação à sua escolha): a função *map* que determina como o agrupamento é executado e a função *reduce* que efetua as agregações.

De facto, o acesso em tempo útil à informação é essencial para que organizações tomem as decisões certas. A motivação para o uso das plataformas baseadas em MapReduce é a capacidade de acomodar um número de nodos mais elevado num *cluster* do que os tradicionais SGBDs relacionais e realizar análises diretamente sobre os dados não estruturados, sem os carregar para tabelas com esquemas previamente definidos (Chaudhuri et al., 2011). As técnicas de armazenamento de dados e processamento de *queries* influenciam diretamente os tempos de resposta às questões colocadas pelos decisores. Neste contexto, as vantagens que o modelo de programação MapReduce podem trazer aos SDW devem ser exploradas para minimizar estes tempos de resposta.

Nesta investigação, não será efetuada, contudo, uma análise do impacto das estruturas resultantes do estudo num *cluster* de várias máquinas, apesar de o modelo de programação MapReduce ser projetado para processar conjuntos de dados volumosos em paralelo, distribuído por um *cluster*. Assim, e de forma resumida, para este trabalho de dissertação foram definidos os seguintes objetivos:

- Estudar a estrutura típica de um DW e algumas soluções para a melhoria do desempenho das *queries* que sobre ele são usualmente lançadas.

- Explorar o modelo de programação MapReduce e desenvolver estruturas de dados adequadas para suportar um DW típico utilizando esse modelo.
- Elaborar um estudo comparativo entre um SDW baseado em MapReduce e um SDW convencional baseado num modelo multidimensional de dados assente numa plataforma relacional.

1.5 Estrutura da Dissertação

O objetivo deste trabalho consiste essencialmente no estudo de uma adaptação de uma estrutura multidimensional de dados – habitualmente um esquema em estrela – para uma baseada no modelo de programação MapReduce. Desta forma, interessa destacar algumas das vertentes relacionadas com o problema examinado, com especial destaque nas técnicas de otimização de desempenho de *queries* em ambientes de processamento analítico. Nessa perspetiva e, de acordo com âmbito proposto na Secção 1.3, além do presente capítulo, este documento encontra-se estruturado em mais 4 capítulos, divididos de forma a enquadrar o leitor dos assuntos abordados, nomeadamente:

- **Capítulo 2:** Apresentação do modelo de programação MapReduce, compreendendo pequenos exemplos. Caracterização de um cenário típico para a aplicação deste modelo de programação. Algumas aplicações baseadas em MapReduce e alguns casos de sucesso.
- **Capítulo 3:** Exploração de uma estrutura típica de um DW, incluindo possíveis técnicas utilizadas na satisfação das *queries*. Análise da integração de MapReduce em ambientes de DW (condições para o fazer e implementações de SDW existentes baseadas em MapReduce).
- **Capítulo 4:** Caracterização de um caso de estudo (uma estrutura multidimensional de dados típica). Demostração do processo de integração de MapReduce para o caso de estudo descrito. Elaboração de um estudo de avaliação do antes, em termos de tempos de satisfação das *queries*, espaço em disco, custos de manutenção de estruturas e planos de execução das *queries*. Paralelamente, são apresentados alguns aspetos importantes das *queries* processadas em MapReduce.

- **Capítulo 5:** Apreciação crítica do trabalho desenvolvido. Realização do levantamento dos contributos que este trabalho pode conceder no desenvolvimento de estruturas de dados em MapReduce. Por último, é indicado uma possível orientação para trabalho futuro.

Capítulo 2

MapReduce

2.1 O Modelo de programação

O MapReduce (Dean and Ghemawat, 2004) é um modelo de programação dedicado ao processamento de grandes conjuntos de dados. Este foi introduzido em 2004 pela Google[6], tendo sido desenvolvido por Jeffrey Dean e Sanjay Ghemawat. O MapReduce (MR) coloca em prática os princípios da computação paralela e distribuída. As inúmeras computações simples – com uma quantidade elevada de dados de entrada – que a Google realizava diariamente, como a construção da estrutura de indexação para o seu serviço de pesquisa, requeriam um processamento distribuído efetuado em *clusters* de milhares de máquinas de forma a que fosse possível obter os resultados num intervalo de tempo satisfatório.

A computação paralela e distribuída são caracterizadas por possuírem configurações complexas de forma a puderem dar resposta a falhas, à distribuição dos dados e à sincronização. O MR cria uma abstração que sustenta e oculta estas configurações, por exemplo como particionar os dados de entrada, planear a execução paralela ou coordenar a comunicação entre máquinas para permitir realizar tarefas simples sem grande esforço. Assim, os programadores inexperientes em sistemas

[6] http://www.google.com/

paralelos e distribuídos beneficiam das suas características em atividades de processamento de dados. Outra propriedade atrativa do MR consiste na sua alta escalabilidade, que permite processar *petabytes* de dados por milhares de *commodity machines*. O MR tem também uma implementação *open source*, denominada Hadoop[7] da Apache Software Foundation, que aquando do seu lançamento proporcionou uma versão aberta para a investigação do modelo pelas comunidades académica e industrial.

Inspirado na programação funcional, o MR deriva de duas funções de ordem superior: *map* e *reduce*, usualmente chamada *fold*. No contexto da programação funcional, a *map* aplica uma dada função, definida pelo utilizador, a cada elemento de uma lista; a *fold* combina uma função, também definida pelo utilizador, e uma lista para produzir um resultado. O modelo de programação subjacente ao MR resume-se na definição destas funções para processar uma estrutura de dados, pares chave/valor, onde a função *map* (programação funcional) corresponde à fase *map* (MR) e a função *fold* (programação funcional) corresponde à fase *reduce* (MR). Desta forma e já no contexto do MR, a função *map* (Figura 4.1) aceita um par de entrada e produz um conjunto de instâncias com o formato chave/valor de domínio diferente ao recebido. Antes da fase *reduce* receber o conjunto produzido por *map*, este é organizado de forma a associar todos os valores, de cada chave, e agrupa-os por conjuntos, respetivamente. Esta operação é realizada pelo próprio MR. A função *reduce* (Figura 4.2) recebe o conjunto de instâncias organizado e calcula um subconjunto, tipicamente uma agregação, para cada chave.

$$map\ (k1, v1)\ \rightarrow lista(k2, v2) \quad (1)$$
$$reduce\ (k2, lista(v2)) \rightarrow lista(k3, v3) \quad (2)$$

Figura 4: Funções do MapReduce

Os pares de entrada da função *map* e os pares que a função *reduce* produz estão armazenados num sistema de ficheiros distribuídos; os pares intermédios encontram-se no disco local do nodo que o produziu.

[7] http://hadoop.apache.org/

2.1.1 Exemplos de aplicação

O exemplo mais comum da aplicação de MR consiste no cálculo do número de ocorrências para cada palavra de um conjunto de documentos (Dean and Ghemawat, 2004). As funções *map* e *reduce* são representadas pelo *pseudo-código*:

```
map (String chave, String valor):
     Para cada palavra p em valor
          emitirParIntermedio(p, ‘1’)

reduce (String chave, Coleção valores):
     Resultado = 0;
     Para cada valor v em valores
          Resultado += ParseInt(v);
          Emitir(Resultado);
```

Figura 5: Funções *map* e *reduce*

A *map* recebe um par chave/valor, na qual a chave equivale ao nome do documento e o valor corresponde ao seu conteúdo, que é percorrido de forma a associar a cada palavra o valor '1'. Desta forma, um conjunto intermédio é produzido, composto por pares palavra/1, como a Figura 6 ilustra.

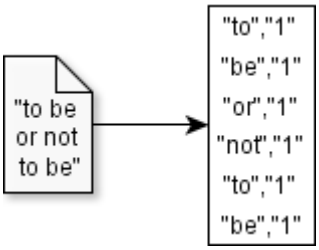


Figura 6: Função *map*

O MR reúne todos os valores de uma palavra e junta-os de forma a aplicar a função *reduce*, que soma todos os valores correspondestes a cada palavra (Figura 7).

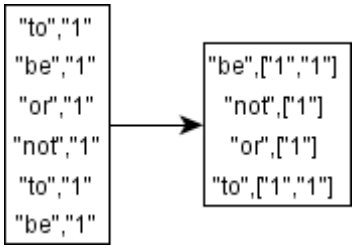


Figura 7: Agrupar valores pelas chaves ordenadas

No final são emitidos os pares palavra/ (resultado da soma), realizada pela função *reduce*. O output é escrito em ficheiro, um por cada *reduce*, e dentro destes, as palavras encontram-se ordenadas alfabeticamente (
Figura 8).

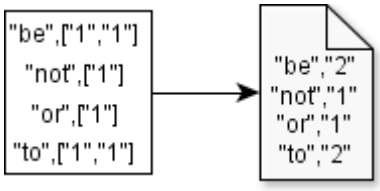


Figura 8: Função *reduce*

Outro exemplo corrente da aplicação do modelo baseia-se na construção de um índice invertido. Um índice invertido é uma estrutura de dados que associa a um conjunto de termos, uma lista de documentos que os menciona. A função *map* lê os documentos e produz uma sucessão de pares, no qual o termo corresponde à chave e o nome do documento ao valor, respetivamente. A função *reduce* recebe o termo e a lista de documentos associada e organiza-a de forma a remover repetidos, para no final devolver o índice invertido.

2.2 Execução de trabalhos em MR

Um programa em MR, também conhecido por trabalho ou job, consiste na implementação das funções *map* e *reduce*, juntamente com as configurações de alguns parâmetros, como os caminhos das pastas de *input* e output (Lin and Dyer, 2010). Antes de analisar a execução de um trabalho em MR, é apresentada a terminologia que será utilizada:

- **Trabalho** - programa em MR.
- **Máquina** - computador que pertence a um *cluster*.
- **Tarefa** ou ***worker*** - um processo que executa numa máquina.
- ***Mapper*** - uma tarefa que aplica a função *map*.
- ***Reducer*** - uma tarefa que aplica a função *reduce*.
- **Nodo** - processo que se encarrega de gerir a execução das tarefas. Estes são executados nas máquinas do *cluster*, idealmente um nodo corresponde a uma máquina. O JobTracker[8], do Hadoop MR, é um exemplo de um nodo.

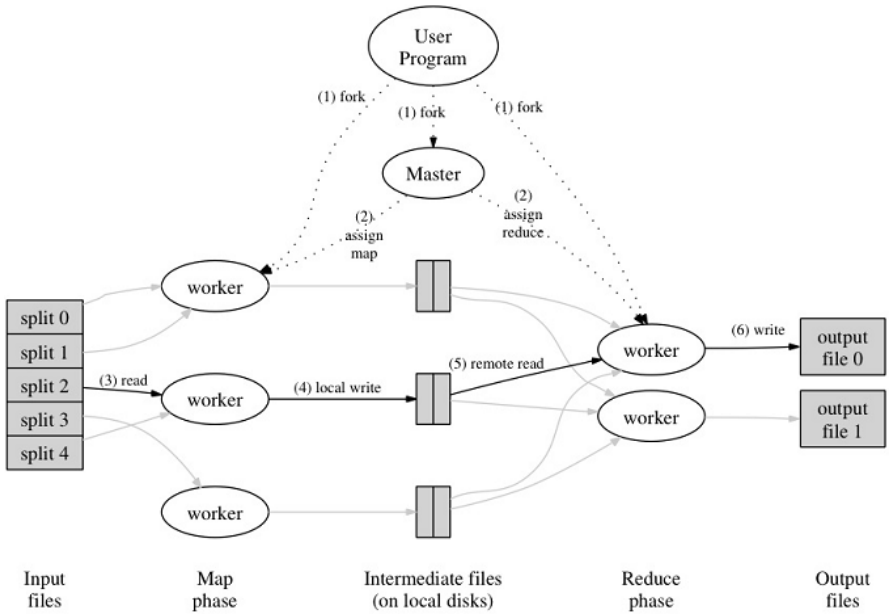


Figura 9: Vista geral da execução em MR (retirado de (Dean and Ghemawat, 2004))

A execução de um trabalho é realizada através de vários passos. No primeiro passo, o *input* é dividido em M partes de tamanho fixo, tipicamente de 64 MB cada uma delas, e inicia o mesmo número de cópias do programa no cluster. Um dos nodos do *cluster* é o master e os restantes são os *workers*. O master escolhe os *workers* que não se encontram em atividade e, para cada um, atribui-lhes a tarefa *map* ou *reduce* (segundo passo). O número de *mappers* é M. No entanto o

[8] http://wiki.apache.org/hadoop/JobTracker/

número de *reducers* pode ser configurado pelo utilizador. O terceiro passo resume-se na execução do *mapper*. Esta operação gera os pares chave/valor intermédios que serão escritos num *buffer* em memória. Periodicamente, este *buffer* é armazenado no disco local (quarto passo). No final da execução dos *mappers*, os pares intermédios que se encontram em disco são divididos em R (número de *reducers*) partições de tamanho igual. A localização destas partições é enviada ao master, que as encaminha para os *reducers*. Quando um *reducer* é notificado pelo master, este usa *remote procedure calls* (RPC) para ler os dados intermédios que se encontram no disco do *mapper* (quinto passo). Por último, cada *reducer* ordena os dados de *input*, ordena e agrega-os por chave e aplica a função *reduce*. O output é escrito num ficheiro separado, um para cada *reducer*, com o formato chave/valor. Quando todos os *mappers* e *reducers* terminarem, o master anuncia ao utilizador que o trabalho foi concluído (Dean and Ghemawat, 2004).

2.3 Cenário típico

O método utilizado pelo MR para a resolução de problemas envolve, tipicamente, o seguinte conjunto de ações:

1. Ler um grande conjunto de dados.
2. Aplicar a função *map* para extrair parte da informação pretendida.
3. Organizar e agregar a informação extraída em pares chave/valor.
4. Aplicar a função *reduce* para agregar, filtrar ou transformar.
5. Escrever o conjunto produzido.

O utilizador define as funções *map* e *reduce*; a execução do programa, de forma paralela, fica encarregue ao MR. As operações *map* não têm dependência entre si, mas as operações *reduce* estão dependentes dos pares produzidos por *map*, já entre si, as operações *reduce* executam de forma independente. Num programa MR, a sincronização ocorre durante a fase de organização, quando os pares intermédios são copiados dos *mappers* para os *reducers*. Este método pode mostrar-se restrito, já que MR impõe o uso de uma estrutura chave/valor, mas é capaz de implementar algoritmos sofisticados, mesmo que implique decompor o algoritmo em vários trabalhos MR (Lin and Dyer, 2010). O MR é caracterizado por se basear numa arquitetura *shared-nothing*, isto é, os nodos utilizam os recursos locais (disco e memória) para as computações. Porém, o *input* e o output dos trabalhos MR são partilhados por todos os nodos. A arquitetura

shared-nothing torna o MR muito mais escalável do que aquelas que partilham disco ou memória (Yang et al., 2007).

Qualquer problema com as capacidades de um trabalho ser partido em problemas menores e executado em paralelo, e de forma independente, tem o perfil adequado para a aplicação de MR. Os problemas denominados de *embarrassingly parallel problems* são os mais adequados, pelo facto dos processos não (ou raramente) comunicarem entre si. Assim, os dados são distribuídos, inicialmente, e no final recolhem-se os resultados (a sincronização é reduzida). Os *embarrassingly parallel problems* são adaptados para executar em apenas um trabalho. O MR também suporta algoritmos iterativos, apesar destes serem difíceis de implementar. Normalmente, estes algoritmos requerem mais do que um trabalho para executar cada iteração e, por vezes, um número elevado de iterações para satisfazer os critérios de paragem. Executar um número elevado de trabalhos resulta em perda de eficiência, devido ao tempo gasto pelo *framework* em tarefas background (latência), por cada trabalho (Srirama et al., 2012).

2.4 O Apache Hadoop

O Apache Hadoop (White, 2012) é um *framework* que permite processar grandes quantidades de dados dispersos por *clusters* de computadores, através de bibliotecas *open source*, em Java. Foi criado por Doug Cutting, o criador do Apache Lucene[9], e teve a sua origem no Apache Nutch[10], um motor de busca *open source* que faz parte do projeto Apache Lucene. O Apache Nucth arrancou em 2002. A sua arquitetura não suportava escalabilidade, uma característica desejada para o processamento (Bondi, 2000). Em (Ghemawat et al., 2003) descreveu-se o sistema de ficheiros utilizado pela Google (GFS), que resolve o armazenamento de ficheiros de grande dimensão. Esta publicação deu origem ao sistema de ficheiros distribuídos do Nutch (NDFS), em 2004, que mais tarde adotou o nome HDFS. Nesse mesmo ano, a Google introduziu o modelo de programação MR, já apresentado anteriormente. Uma implementação deste modelo, para o Nutch, começou a ser desenvolvida em 2005 e, a meio do ano, os principais algoritmos deste motor de busca estavam prontos a executar em MR e NDFS. Em 2006, o NDFS e o MR, separaram-se do Apache Nutch para

[9] http://lucene.apache.org/
[10] http://nutch.apache.org/

criar um subprojecto independente – o Hadoop – uma vez que estes eram aplicáveis a mais domínios, para além dos motores de busca (White, 2012).

2.4.1 Um sistema de ficheiros distribuídos

Ocasionalmente, um qualquer conjunto de dados ultrapassa a capacidade física de armazenamento de uma máquina. Neste caso, uma das soluções consiste em dividir este conjunto de dados por partes e distribui-los por várias máquinas separadas, o que exige um sistema de ficheiros adequado, caracteristicamente um sistema de ficheiros distribuídos, para gerir o acesso e armazenamento de ficheiros numa rede de computadores. O Apache Hadoop é acompanhado por um sistema de ficheiros distribuídos, chamado *Hadoop Distributed Filesystem* (HDFS), mas é capaz de integrar outros sistemas de armazenamento, como o *Amazon S3* ou o *local filesystem*, devido às suas capacidades de abstração (White, 2012).

Tal como já foi referido, o HDFS nasceu após a Google divulgar o GFS. Este partilha muitas características com outros sistemas de ficheiros distribuídos, mas, obviamente, apresenta algumas diferenças, como a alta tolerância a falhas e a capacidade de utilizar *hardware* barato (Borthakur, 2007). Outra característica relevante do HDFS é o *streaming access*. Este sistema de ficheiros foi construído com o intuito que o padrão mais eficiente para processar dados é: uma escrita, muitas leituras. Após o conjunto de dados ser copiado da fonte ou gerado, este é submetido a várias análises, em que cada uma delas envolve uma grande porção (ou todos) os dados - o tempo de leitura integral dos dados é, assim, mais importante do que a latência de ler o primeiro registo (White, 2012).

O HDFS adota a arquitetura cliente-servidor, tal como o Hadoop MR. Um nodo é o master, também designado por *namenode*, enquanto que os restantes nodos são denominados por clientes ou *datanodes*. O *namenode* mantém os *metadados* associados ao sistema de ficheiros, isto é as permissões ou a localização de ficheiros, e coordena as operações de abrir ou mudar o nome a ficheiros. Quanto aos *datanodes*, encontram-se dispersos pelo *cluster*, com a responsabilidade do armazenamento dos dados (Borthakur, 2007).

2.4.2 O HBase

O MR e o GFS formam a base para processar grandes quantidades de dados. Porém este processamento revela-se ineficiente em acessos aleatórios aos dados, principalmente nas consultas em tempo real. Isto deve-se à ineficiência do GFS quando trabalha com um número elevado de ficheiros de tamanho reduzido, apesar do bom desempenho para um número reduzido de ficheiros de tamanho elevado. Quanto maior o número de ficheiros, maior a pressão que a memória do nodo master tem de suportar. Assim, a Google desenhou uma solução capaz de armazenar blocos pequenos, em GFS, designada de Bigtable (Lars, 2011).

O Apache Hadoop está relacionado com diversos projetos da Apache e, entre eles, encontra-se o HBase (Lars, 2011). Apenas decorrido um ano após a publicação de (Chang et al., 2006), que apresentou a arquitetura da Bigtable da Google, surge o HBase, uma implementação *open source* em *Java*, com base no HDFS e criada pela Powerset - mais tarde, este projeto tornou-se parte da Apache Software Foundation.

O modelo de dados da Bigtable, assim como do HBase, consiste num conjunto de associações chave/valor multidimensional, esparso, ordenado, persistente e distribuído. Este conjunto é indexado através da chave linha (*row key*), chave coluna (*column family* e *qualifier*) e um *timestamp*; o valor corresponde a um conjunto de bytes. O acesso a um valor pode ser expresso da seguinte forma:

$$(chave\ linha, chave\ coluna, timestamp) \rightarrow valor$$

As linhas são, arbitrariamente, cadeias de caracteres (*Strings*), ordenadas lexicograficamente. As colunas são formadas por famílias (criadas juntamente com a tabela e armazenadas em ficheiros separados) e *qualifiers*. A chave para a coluna segue o formato *família:qualifier*. O *timestamp* permite armazenar várias versões para a mesma chave (chave linha, chave coluna). Este é registado em microssegundos para a Bigtable, já para o HBase é em milissegundos. As versões encontram-se ordenadas decrescentemente pelo seu *timestamp*, de forma a aceder primeiro à versão mais recente (Chang et al., 2006). As tabelas são automaticamente particionadas, de forma horizontal pelo HBase em regiões e, cada uma abrange um subconjunto de linhas de uma tabela (Lars, 2011).

2.5 Aplicações

A comunidade científica apresentou várias implementações para o MR em diferentes áreas, como computações *machine learning*, simulação e processamento de imagem (Chen and Schlosser, 2008), indexação e processamento de grafos (Lin and Dyer, 2010), entre outros. O domínio de problemas estudado com MR é, assim, bastante amplo. O artigo que apresenta o MR (Dean and Ghemawat, 2004) expõe um conjunto de aplicações para este modelo. Dentro do artigo, os autores identificam várias aplicações, como: calcular do número de ocorrências para cada palavra de um conjunto de documentos ou fazer a construção dos já referidos índices invertidos.

A pesquisa e a ordenação são duas aplicações bastante comuns deste modelo. O *Distributed Grep* é a versão distribuída da ferramenta *grep* que emite as linhas de um conjunto de ficheiros se corresponder a um padrão fornecido. Em MR, esta operação é realizada pelos *mappers*. O *output* gerado, o conjunto intermédio, corresponde ao *output* final. Desta forma, não é necessário definir a função *reduce* para o cálculo do *grep*. Na ordenação distribuída, o *map* extrai a chave de cada registo e emite um par <chave, registo>. O *reduce* emite todos os pares, sem os alterar, do mesmo modo que o *distributed grep*.

Outro dos usos mais comuns deste modelo é a análise de dados, principalmente quando a fonte de dados para análise é grande. A análise de *logs* (Blanas et al., 2010) emergiu como uma forma de análise importante, vulgarmente realizada pelo MR. O processamento de *logs* (ex. *click-stream*, chamadas de telefone ou uma sucessão de transações) são recolhidos continuamente e armazenados em ficheiros. O MR é usado para o cálculo de várias estatísticas, de forma a obter uma perceção mais correta do negócio a partir dos dados, uma vez que os ficheiros que armazenam os registos ou os eventos não necessitam de uma formatação específica para serem processados. Não é exigida para isso uma transformação prévia dos dados para os processar. A contagem de acessos a um URL, semelhante ao do número de ocorrências para cada palavra, com a diferença que a função *map* recebe o registo das solicitações de uma página web e emite o par <URL, 1>, é um exemplo de uma análise log.

O MR é também aplicado a problemas mais complexos, como é o caso dos algoritmos iterativos (Ekanayake et al., 2010). A arquitetura do MR, apresentado pela Google, concentra-se num trabalho MR para processar os dados com uma melhor tolerância a falhas e armazenar grande parte dos outputs em ficheiros, durante a computação. Numa sequência de vários trabalhos MR, os

dados são carregados repetidamente, o que se traduz num custo elevado no desempenho para muitas destas aplicações. No entanto, este modelo tem a capacidade de implementar vários tipos de algoritmos iterativos, como: grafos, nos quais são exemplos o *PageRank* e a pesquisa paralela *beadth-first*, ou data *mining*, como sucede com o algoritmo *k-means*.

2.6 Alguns casos de sucesso

Desde a sua introdução à comunidade tecnológica, em 2004, o MR foi adotado por várias entidades para os diversos desafios que labutam diariamente. O modelo tem vindo a cimentar, cada vez mais, uma posição na indústria da computação e da comunidade de investigação. Em 2008, a Google, pioneira deste modelo, processava mais de 20 *petabytes*, diariamente, grande parte dedicado à reconstrução do sistema de indexação que produz a estrutura de dados usada no seu serviço de pesquisas (Dean and Ghemawat, 2008). Ainda na Google, o MR é aplicado a outros domínios, como o processamento de imagens de satélite, computações de grafos, produzir relatórios de *queries* populares, tradução (processamento de linguagens) e notícias (*clustering*). Em 2010, o Facebook[11], um dos maiores serviços de redes sociais, apresentou uma implementação de um DW e de uma infraestrutura analítica (Thusoo et al., 2010). O DW armazena mais de 15 *petabytes* de dados (2.5 *petabytes* após a compressão) e carrega, diariamente, mais de 60 *terabytes* de dados (após a compressão), para a análise de dados, recomendações ao utilizador, geração de relatórios, criação de *business intelligence dashboards*), entre outros. Existe uma lista de instituições extensa (Community, 2013) que utilizam o Hadoop para fins educacionais e de produção.

[11] http://www.facebook.com

Capítulo 3

SDW Baseados em MapReduce

3.1 Estrutura típica de um *data warehouse*

Desde os anos 70, o modelo de dados relacional, proposto por Edgar Frank Codd (Codd, 1970), tem sido adotado pelas bases de dados relacionais. No esquema relacional, uma relação é habitualmente designada por tabela que caracteriza uma estrutura bidimensional formada por zero ou mais instâncias, também designadas por tuplos ou registos. O número de tuplos de uma relação denomina-se por cardinalidade. As relações são compostas por um ou mais atributos, cada um definido num dado domínio de valores. O número de atributos de uma relação é designado por grau da relação. Analisemos o pequeno exemplo apresentado na Figura 10, que mostra uma relação designada por "Cliente". Esta relação é composta por 5 tuplos (cardinalidade da relação) e 3 atributos (grau da relação).

Cliente	*(chave*	*nome*	*idade)*
	1	Cliente 1	20
	2	Cliente 2	25
	3	Cliente 3	18
	4	Cliente 4	30
	5	Cliente 5	19

Figura 10: A relação "Cliente"

Cada tuplo numa relação é identificado de forma unívoca por um atributo ou por um conjunto de atributos, que é designado por chave primária (ex. "clienteID" na relação "Cliente"). Este identificador garante que as instâncias de uma dada relação são distintas e que permite instituir ligações entre relações, por exemplo:

$$\boldsymbol{Cliente}\ (clienteID, nome, idade)$$
$$\boldsymbol{Venda}\ (vendaID, clienteID, quantidade)$$

As relações "Cliente" e "Venda" têm em comum o atributo "clienteID" que estabelece a ligação entre as duas relações. Na relação "Venda", o atributo "clienteID" é uma chave estrangeira.

O modelo relacional apresenta regras que normalizam o conjunto de atributos com o objetivo de reduzir a redundância nos dados. Desta forma, a manutenção dos dados entre as relações é reduzida, bem como o desempenho e o espaço para armazenamento. Este modelo tem as características ideais para modelar e processar dados de forma transacional. A normalização certifica a integridade e consistência dos dados, preservada pelas transações. Cada transação utiliza um número pequeno de tuplos, o que torna difícil realizar uma análise de dados em tempo útil. As consultas podem abranger várias tabelas e como tal exigem frequentemente o estabelecimento de operações de junção. A junção de tabelas traz custos excessivos no desempenho e, consequentemente no tempo de resposta às questões colocadas.

Um DW é orientado a um assunto, integrado, não volátil e variável ao longo do tempo (Inmon, 2002). O assunto está diretamente focado na organização da informação, isto é, a modelação é orientada à análise de dados para facultar uma visão simples da matéria de que se trata. Integrado, uma vez que tem a capacidade de incluir dados provenientes de diversas fontes de dados heterogéneas e conjugá-los de forma a tornarem-se consistentes, por exemplo, através de técnicas de limpeza e transformação. Este também deverá permitir uma leitura histórica da informação para apresentar uma evolução do assunto ou do negócio. Por último, um DW é não volátil, o que significa que é um repositório físico e separado do ambiente operacional, e deve apenas suportar o carregamento de dados e permitir o seu acesso, não os alterar.

Um DW suporta vários níveis de detalhe e diferentes perspetivas para os dados, com tempos de resposta ínfimos. O espaço multidimensional é usualmente suportado por uma base de dados

relacional e encontra-se orientado a um assunto, ao contrário de um sistema OLTP, que se centra no registo de todas as transações realizadas por uma organização. Os sistemas OLTP são caracterizados por possuírem estruturas otimizadas para leituras e escritas de dados, com níveis de redundância baixos. Neste sistema, as *queries* mais comuns são de leitura e escrita, de um número reduzido de registos espalhados por várias tabelas diferentes. Pelo contrário, as *queries* em DW leem um número elevado de registos para o cálculo de agregações. Este tipo de *queries* requer que os dados do DW se encontrem representados por um dado modelo multidimensional (Chaudhuri and Dayal, 1997).

Um dos conceitos importantes é o facto. Este conceito representa um conjunto de eventos que ocorreram na empresa. Um exemplo de um evento é a venda simples, ou seja um cliente realizar uma compra de um conjunto de produtos. Cada facto é descrito por valores de um conjunto de medidas pertinentes que fornece uma descrição quantitativa dos eventos. Receita e desconto da venda são exemplos de medidas. A dimensão é outro conceito relevante para o modelo multidimensional. Uma dimensão é um eixo de análise para os factos (ex. produto ou cliente, numa venda simples). Normalmente, estas dimensões encerram em si hierarquias que representam vários níveis de agregação. Os níveis são definidos por atributos. Uma hierarquia modelo para produto é formada pelos atributos: "produto -> tipo -> categoria" (Golfarelli and Rizzi, 2009).

Um *data mart* é uma parte de um DW que suporta os requisitos de um determinado área específica de tomada de decisão de uma organização (Kimball et al., 1998). Quando comparados com o DW, os *data marts* têm menos dados e, assim, as resposta às *queries*, bem como a transformação e a integração dos dados num *data mart* são mais rápidas do que num DW. A simplicidade e o custo de um *data mart*, torna-o num projeto mais desejável, relativamente a um DW, mas o aumento do seu volume provoca uma degradação do desempenho. O aumento do número de *data marts* numa organização exige uma administração forte e como tal mais complexa (ex. manter a consistência, integridade, segurança, etc. de mais sistemas de dados).

Numa implementação relacional de um DW, o modelo de dados multidimensional é armazenado numa base de dados relacional e as operações são expressas normalmente em SQL. O desenho desta base de dados está otimizado para realizar carregamentos e consultas de dados eficientemente. Normalmente, este desenho traduz-se num esquema em estrela (*star schema*) ou

num floco de neve (*snowflake schema*). Existem algumas diferenças entre estas duas configurações de esquemas. O esquema em floco de neve (Inmon, 2002) normaliza as tabelas dimensão até à terceira forma normal. Já o esquema em estrela (Kimball, 1996) mantém as tabelas dimensão altamente desnormalizadas (uma tabela por cada dimensão). A maioria dos DWs usam o esquema em estrela para representar o modelo de dados multidimensional (Chaudhuri et al., 2011).

3.1.1 Esquema em estrela

Numa base de dados relacional, a configuração multidimensional que regula a organização dos factos e das dimensões para armazenamento, corresponde, geralmente, a um esquema em estrela. Este é composto por um conjunto de tabelas dimensão e por uma tabela de factos (Figura 11).

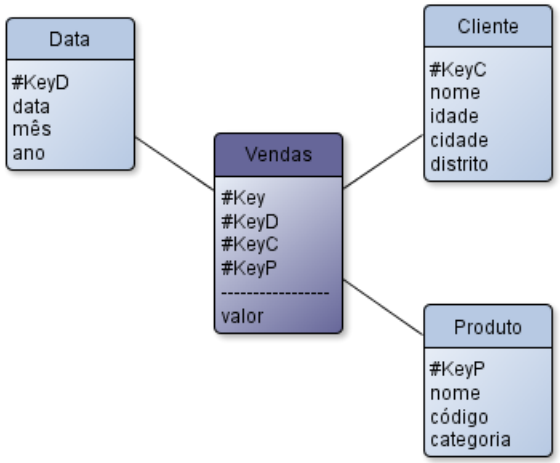


Figura 11: Esquema em estrela

Num esquema em estrela, as tabelas dimensão ("Data", "Cliente" e "Produto") contêm atributos que fornecem um dado contexto de análise para a informação incorporada numa dada tabela de factos. Nas *queries*, tais atributos especificam quais os factos a agregar. O seu valor fornece um contexto às medidas. Nas configurações mais vulgares, as tabelas dimensão não se encontram na terceira forma normal. No entanto, o esquema em estrela não fornece um suporte explícito para os atributos que compõem uma hierarquia. O esquema em floco de neve, ilustrada pela Figura 12, faculta um refinamento do esquema em estrela, em que as hierarquias são explicitamente

representadas pela normalização das tabelas dimensão. Isto conduz a uma redução do custo de manutenção das tabelas dimensão (Chaudhuri et al., 2011). Na Figura 12, a normalização da tabela dimensão "Cliente" até à terceira forma normal origina uma nova tabela, a tabela "Cidade". Esta nova tabela é designada de *Outriger*.

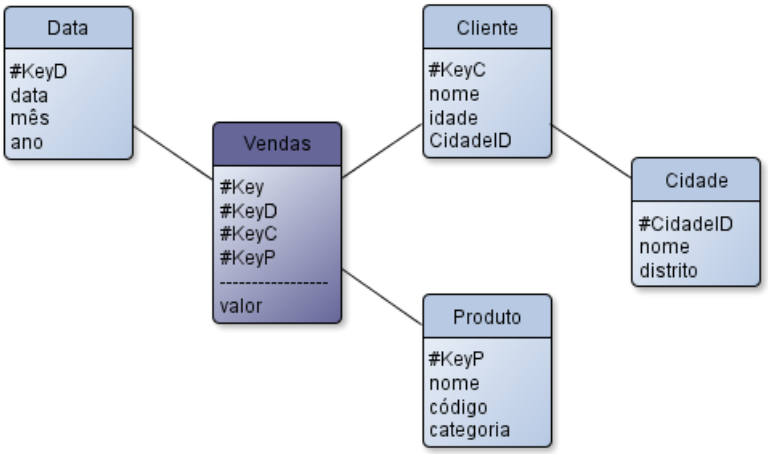


Figura 12: Esquema em floco de neve

No centro de um esquema em estrela reside uma tabela de factos. Esta reúne as chaves estrangeiras das tabelas dimensão associadas a si. Por vezes, o conjunto das chaves de substituição forma a chave primária da tabela de factos. Cada linha da tabela de factos armazena um facto, com um nível de detalhe específico, representando o elemento de dados mais detalhado, que caracteriza o grão da tabela de factos (ex. vendas diárias de um produto a um cliente). Através de agregações, a informação contida nestas tabelas pode ser assimilada numa variedade de níveis diferentes, como vendas mensais ou anuais. A granularidade da tabela de factos pode também ser modificada com a adição ou a remoção de dimensões. Outra configuração possível (e mais complexa) de um esquema, o esquema em constelação (*constellation schema*), pode incorporar várias tabelas de factos que partilham tabelas de dimensão - este esquema pode também ser visto como uma coleção de esquemas em estrela. A Figura 13 representa um esquema em constelação no qual se pode identificar as tabelas de factos "Vendas" e "VendaProds", partilhando as tabelas dimensão "Data" e "Produto".

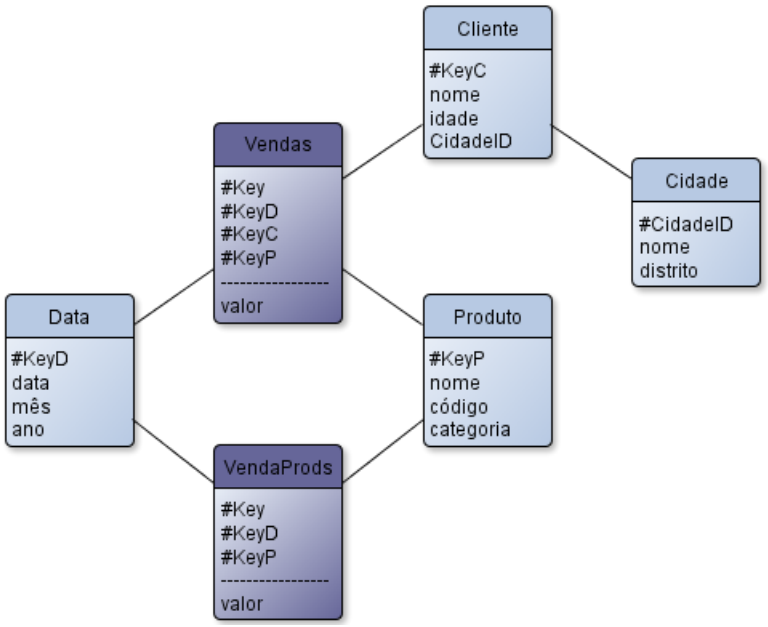


Figura 13: Esquema em constelação

Geralmente, o modelo multidimensional assume que os valores que povoam as tabelas de dimensão são estáticos, ou seja, são independentes do tempo, o que nem sempre é verdade no mundo real. *Slowly changing dimensions* (Kimball, 1996), ou dimensões de variação lenta é o termo que caracteriza uma dimensão que contém atributos que variam ao longo do tempo, como a mudança de morada de um cliente ou a atribuição de uma nova categoria a um produto. Estes eventos – capturados nos sistemas operacionais – são modelados no esquema lógico para acomodar estas alterações. Para cada atributo é necessário especificar uma estratégia que registe tais mudanças. As estratégias mais comuns são:

- **Tipo 1**: substituir o valor antigo pelo atual no registo da tabela dimensão. A Figura 14 exemplifica esta estratégia para o atributo *categoria*. No dia '01/01/2000', o 'ProdC' pertencia à categoria 'CAT03', contudo, no dia '01/03/2000' este produto foi classificado com a categoria 'CAT04'. A solução do tipo 1 para este cenário atualiza o registo já existente com a nova categoria. Todos os fatos ficam agora associados com o novo valor. Este tipo é fácil de implementar, no entanto não mantem um histórico dos valores anteriores.

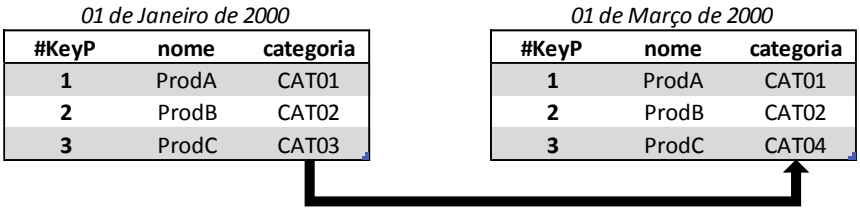


Figura 14: Tipo 1

- **Tipo 2**: adicionar um registo com o novo valor à tabela dimensão. A Figura 15 demonstra a resposta à alteração do 'ProdC' do dia '01/03/2000'. Esta abordagem cria dois ou mais registos, separados fisicamente pela chave de substituição "KeyP", que se relacionam com um registo lógico. Todos os factos posteriores ao dia '01/03/2000' associam-se, agora, ao registo novo com o valor da chave de substituição superior aos registos mais antigos. Ao contrário do tipo 1, o tipo 2, bem como os tipos 3 e 4, mantêm um histórico.

#KeyP	nome	categoria
1	ProdA	CAT01
2	ProdB	CAT02
3	ProdC	CAT03
4	ProdC	CAT04

Figura 15: Tipo 2

- **Tipo 3**: adicionar um ou mais atributos à tabela dimensão, como a Figura 16 expõe. Esta estratégia regista a versão anterior – atributo "prevCategoria" – juntamente com a data da última modificação; o valor atual encontra-se armazenado em categoria. No exemplo da Figura 16, o histórico está limitado a apenas uma versão dos dados, contudo pode ser estendido a várias versões através da adição de atributos. Os tipos 1 e 2 preservam a estrutura da tabela original, contudo esta estratégia altera com a adição das colunas.

#KeyP	nome	categoria	prevCategoria	dataModificação
1	ProdA	CAT01	*NULL*	*NULL*
2	ProdB	CAT02	*NULL*	*NULL*
3	ProdC	CAT04	CAT03	01/03/2000

Figura 16: Tipo 3

Outra estratégia comum é o tipo 4. Esta é representada por duas tabelas: a tabela dimensão que armazena os dados correntes (sem alterar a estrutura base) e outra tabela que armazena os dados que expiraram (Figura 17).

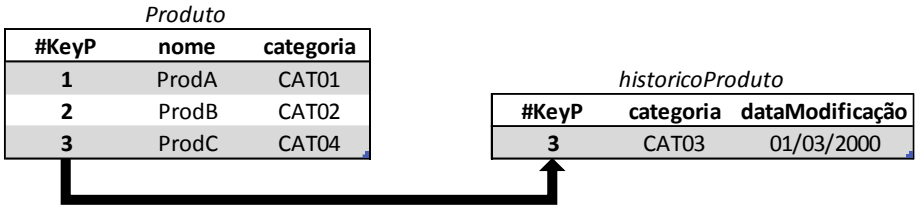


Figura 17: Tipo 4

O tipo 4, assim como o tipo 2, regista um número ilimitado de versões dos dados, mas este armazena as versões antigas numa tabela diferente (historicoProduto). O valor mais recente mantem-se na tabela dimensão. Cada variação do valor do atributo categoria cria um registo adicional na tabela separada (o valor anterior e a data da modificação) com a chave de substituição que identifica registo modificado na tabela dimensão. Esta tabela é relativamente compacta, portanto o custo de armazenamento é menor do que o tipo 2. No entanto, esta estratégia é menos intuitiva. Existem mais abordagens para modelar a variação do valor destes atributos, em particular as abordagens ditas híbridas que combinam de alguma forma as técnicas mais básicas apresentadas.

3.2 Técnicas para a satisfação de *queries*

O objetivo principal de um qualquer SDW é facultar meios capazes de sustentar os processos de tomada de decisão de uma organização. Isto requer uma estrutura de dados desenhada de forma a reduzir o tempo de resposta das consultas aos dados. Um modelo lógico bem desenhado (proveniente de um modelo conceptual adequado), aliado aos avanços da tecnologia e das técnicas de modelação, cria as condições para melhorar o tempo de processamento de *queries* (Golfarelli and Rizzi, 2009).

A *desnormalização* é um das técnicas mais comuns para otimizar o acesso aos dados. Com um custo de armazenamento maior, esta técnica evita a junção de tabelas, frequentemente utilizadas

em *queries*. Assim, o custo de junções é poupado. Não existem cálculos adicionais para aceder a atributos presentes em outra tabela (ex. *outrigers* num esquema em floco de neve). No esquema em estrela, as tabelas dimensão adotam esta técnica. Apesar de a *desnormalização* eliminar grande parte da complexidade nas consultas, esta traz um custo elevado de manutenção e armazenamento dos dados (originada pela redundância dos dados), como anomalias oriundas da inserção, remoção ou atualização de registos.

As bases de dados e os DWs partilham algumas das técnicas para minimizar os tempos de resposta às interrogações. Estre estas, as mais eficazes para melhorar o desempenho de um DW são as vistas materializadas e os índices. Geralmente, um bom modelo físico combina de forma cautelosa estas duas técnicas (Chaudhuri et al., 2011).

Usualmente, as tabelas de factos de um DW têm milhões de registos. Desta forma, analisar todos os registos consome demasiado tempo. Uma estratégia para acelerar este processo consiste em pré-calcular agregações dos dados. Em base de dados, uma vista materializada (Gupta and Mumick, 1995) equivale a uma tabela que armazena os tuplos obtidos de uma pré-computação. Esta oferece um acesso mais rápido aos dados, porque limita-se a percorrer todos os tuplos de uma tabela que responda à questão colocada, sem realizar qualquer cálculo. Esta técnica tem sido amplamente estudada. No entanto tem vários problemas técnicos, por exemplo, as pré-computações consomem tempo e espaço em disco.

Ao longo do tempo, as tabelas referidas nas vistas materializadas sofrem alterações (ex. inserções na tabela de factos ou dimensão). Neste caso, as vistas materializadas tornam-se desatualizadas. Estas, para manter consistência com as tabelas base, são recalculadas a partir do zero ou atualizadas incrementalmente (Huyn, 1997). Na sua maioria, proceder com um novo cálculo sempre que as tabelas bases são modificadas é um desperdício de recursos, já que estas modificações afetam parte da vista envolvida. Assim, estas manutenções costumam realizar-se através de atualizações incrementais.

Um DW contém várias perspetivas de análise e, consequentemente, várias vistas para possível materialização. Um conjunto de vistas materializadas que represente todas as perspetivas de análise é, muitas vezes, um cenário insuportável devido ao limite do espaço em disco e aos custos de manutenção. Ocasionalmente, vistas diferentes partilham os mesmos dados das tabelas base.

Desta forma, este aspeto é explorado para selecionar um conjunto de vistas que traduza um bom desempenho, sem realizar grandes manutenções (Yang et al., 1997).

O modelo lógico oferece a possibilidade de criar diferentes variantes do esquema em estrela para representar vários níveis de agregação. Outra técnica consiste em armazenar os dados relativos a diferentes conjuntos "*group-by*" em tabelas de factos separadas (Golfarelli and Rizzi, 2009). Por exemplo, na Figura 13 pode-se ver a representação de dois níveis de agregação distintos (as tabelas de factos "Vendas" e "VendaProds"). Esta configuração permite realizar operações de *drill down* de "VendaProds" para "Vendas" (Moody and Kortink, 2000), uma vez que a tabela "VendaProds" representa uma granularidade superior (com as chaves estrangeiras das tabelas dimensão "Data" e "Produto") e a tabela "Vendas" que representa o nível de detalhe maior (granularidade inferior). A Figura 18 apresenta mais uma dessas possíveis configurações.

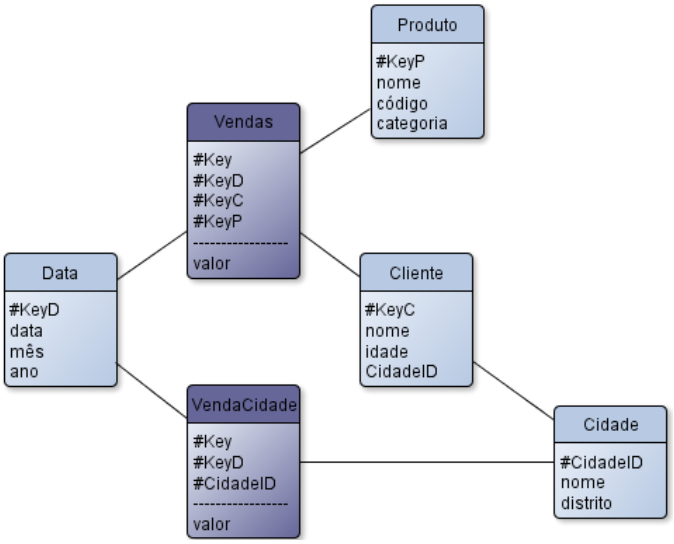


Figura 18: Esquema em floco de neve com dados agregados

À semelhança do esquema apresentado na Figura 13, o esquema da Figura 18 mostra duas tabelas de factos que mantêm níveis de agregação separados. A diferença reside no facto que a tabela com a granularidade superior, agora a "VendaCidade", armazena os factos das vendas relativas à venda diária a clientes de uma determinada cidade. Esta configuração apresenta duas vistas para a hierarquia presente na tabela dimensão "Cliente": "nome -> cidade". O nível "nome" é representado pela tabela "Vendas", já o nível cidade é pela tabela "VendaCidade". Como estas

abordagens usam duas tabelas para responder às *queries*, o desempenho do DW aumenta, ao contrário do que verifica com uma tabela de factos que solucione todas as *queries*.

3.2.1 Índices

Durante as últimas décadas, os índices foram alvo de um estudo intensivo no domínio dos SGBD relacionais. Um índice é uma estrutura de dados que permite o acesso aleatório a registos. Sempre que possível, os índices devem ser utilizados para evitar percorrer todos os tuplos de uma tabela. O cálculo da seleção destes registos pode tirar partido dos índices. Grande parte dos índices é definida para as tabelas dimensão. Estes aumentam o seu volume, quando a tabela cresce, ou seja, os índices requerem espaço específico para o seu armazenamento. Um grande número de índices afeta o processo de carregamento das tabelas, já que é necessário criar novos índices para cada registo inserido (Ponniah, 2004). As vistas materializadas também incorporam índices (Gupta et al., 1997).

A árvore B+ (Comer, 1979), o tipo de índice mais usado e implementado pela maioria dos SGBDs, evoluiu da árvore B (Bayer and McCreight, 1972). A Figura 19 ilustra parte de uma árvore B+, que constrói uma chave numérica, a partir de todos os valores de um atributo. As subárvores da esquerda contêm as chaves numéricas inferiores ou iguais à chave numérica da raiz, todavia as subárvores da direita contêm as chaves numéricas superiores à chave raiz. Assim, a estrutura mantém-se sempre ordenada.

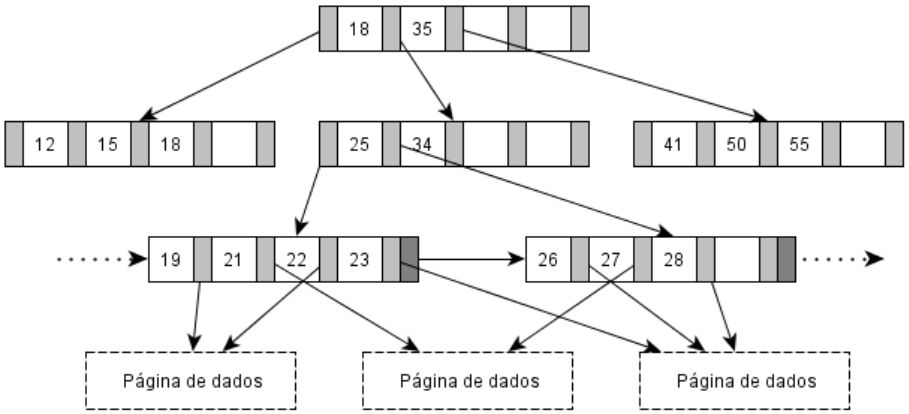


Figura 19: Árvore B+ (adaptada de (Comer, 1979))

Uma árvore B+ pode indexar uma chave primária (índice primário) ou qualquer outro atributo (índice secundário). Caso seja *clustered*, esta ordena os tuplos fisicamente para os atributos definidos por este índice. Caso contrário, a árvore B+ denomina-se por *unclustered*. Um índice *clustered* melhora o desempenho de uma tabela, pois este junta fisicamente os índices e os registos num segmento, logo adquire-se os dados numa leitura. As técnicas tradicionais requerem duas leituras: uma primeira leitura para obter o segmento correspondente aos índices e a segunda leitura para obter os dados, ou seja, os índices e os dados encontram-se em segmentos diferentes. Com um índice *clustered*, as *queries* executam mais rapidamente quando é procurado um intervalo de valores ou um valor em particular (Golfarelli and Rizzi, 2009).

Um atributo composto por muitos valores distintos (ex. cidade) é o mais apropriado para ser indexado por uma árvore B+, porque os valores numa folha (página de dados) são únicas, o que conduz a linhas de dados distintas e não a uma série de linhas por folha. Para os atributos constituídos por poucos valores distintos (seletividade baixa), por exemplo distrito, as árvores B+ não são eficientes. Todavia, combinar o atributo distrito com outro atributo (ex. cidade) aumenta a seletividade. Esta concatenação cria um índice árvore B+ em ambos os atributos, conjuntamente (Ponniah, 2004).

Outra estrutura de dados, os índices Bitmap (O'Neil, 1989), são os ideais para casos de seletividade baixa. Um bitmap é uma sequência de *bits*, uma para cada valor do atributo indexado. Esta estrutura ocupa significativamente menos espaço em relação às árvores B+ para os atributos de seletividade baixa. Geralmente, as *queries* são constituídas por um ou mais predicados que filtram os registos. Por exemplo, uma consulta possível para o esquema da Figura 11 é: "o valor total das vendas para o ano de 2000 e o mês de Janeiro". O processamento desta *query*, através do índice bitmap, obtém os registos que se qualificam pelo cálculo do operador lógico *AND bit* a *bit* dos vetores respetivos. No entanto, caso ocorra uma inserção de um novo valor do atributo indexado, o índice bitmap tem obrigatoriamente de ser reestruturado (Ponniah, 2004).

3.2.2 Particionamento de dados

Outra técnica para a satisfação de *queries*, a partição dos dados, permite a divisão de um conjunto de dados em parcelas. As tabelas, do mesmo modo que os índices, são repartidas em unidades mais pequenas e, assim, mais cómodas de administrar (ex. operações de *backup* e de

carregamento de dados). Esta técnica teve origem nos sistemas centralizados que tiveram a necessidade de partir ficheiros (ex. ficheiros demasiado grande para armazenar num disco). O particionamento de dados é utilizado por bases de dados distribuídas para repartir fragmentos de uma tabela por vários nodos de uma rede. Este, também permite às bases de dados paralelas explorar a largura de banda de múltiplos discos, já que permite leituras e escritas em paralelo (DeWitt and Gray, 1992).

Os dois esquemas mais vulgares para particionar os dados são (Chaudhuri et al., 2011): *hash* e *range*. Estas duas formas de partir os dados são determinadas por valores de *hash*, de um intervalo ou a combinação de vários atributos. Desta forma, é possível mapear uma partição num nodo específico, caso a pesquisa se baseie numa chave. No entanto, este critério pode criar um desequilíbrio (*data skew*) dos dados. Assim, o algoritmo de alocação deve ter em consideração este problema. Um outro esquema comum é o *round-robin*. Este é uma das formas mais simples de particionar os dados e tipicamente dispõe os dados de forma equilibrada (menos suscetível a situações de *data skew*). Todavia, uma pesquisa precisa de aceder a todas as partições, mesmo para as consultas que envolvam uma pequena fração dos dados, uma vez que os dados são alocados aleatoriamente (Furtado, 2011).

O particionamento é um conceito simples de implementar, contudo impõe novos desafios ao desenho do nível físico de uma base de dados. As tabelas têm uma estratégia de particionamento e um conjunto de fragmentos em disco. Tipicamente, aumentar o grau do particionamento resulta numa redução do tempo de resposta das *queries*. Para os varrimentos sequenciais, o tempo de resposta aumenta, visto que são utilizados mais recursos (processadores e discos) para a execução das *queries*. Já para localizar tuplos por associação, o tempo de resposta melhora, porque existem menos tuplos armazenados por cada nodo e, por conseguinte, o tamanho dos índices a consultar diminui.

Um dos objetivos do particionamento dos dados é reduzir a quantidade de dados que devem ser manuseados para dar resposta a uma *query*. Por exemplo, quando o particionamento dos dados se baseie nos valores do atributo ano, uma *query* que solicite os factos de um ano em particular requer apenas o processamento de uma partição para satisfazer a *query*. Agrupar e armazenar um conjunto de tuplos relacionados (*clustering*), no nível físico (isto é, numa página), tem sido alvo também de atenção no domínio dos SGBD (ex. clientes do mesmo distrito). O particionamento

baseado em intervalos (*range*) tende a armazenar os dados desta forma. Assim, o acesso sequencial, bem como os acessos *"point query"* e por intervalos de valores de um atributo executam de forma eficiente (DeWitt and Gray, 1992). A Figura 20 representa um esquema de particionamento *range* da tabela "Vendas" baseado no atributo "Ano".

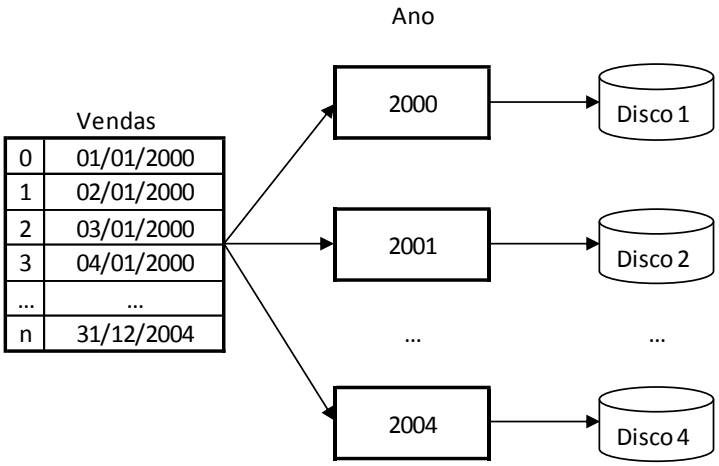


Figura 20: Particionamento baseado no atributo "Ano"

A tabela vendas armazena datas entre os intervalos '01/01/2000' e '31/12/2004'. Com este esquema, os dados são partidos em quatro partições, uma para cada ano, distribuídas por quatro discos. Assim, uma *query* que execute uma análise para o ano de '2000' necessita apenas de consultar o disco 1 (o disco que armazena a partição dos dados correspondente ao ano de '2000').

3.3 Integração do MapReduce em *data warehouse*

MR surge como uma plataforma promissora para o processamento em paralelo e o acesso aos dados, em grande escala. Na introdução (Secção 1.3.) foi discutido possíveis impactos positivos com a introdução de MR em SDW: a escalabilidade e a tolerância a falhas. Os autores da ferramenta Apache Hive, que será abordada adiante, adotaram a tecnologia Hadoop para endereçar os problemas relacionados com a escalabilidade do seu DW, nomeadamente tempos de resposta demorados – por parte das análises – e custos elevados das soluções SDWs tradicionais – os SGBDs relacionais – para fazer frente ao rápido crescimento dos dados nos seus DWs.

Observa-se que, na literatura da área, existem vários estudos realizados comparando as tecnologias MR e SGBD relacionais para o processamento intensivo de dados. Estes apresentam um compromisso entre eficiência e tolerância a falhas, as diferenças fundamentais destas duas tecnologias para o processamento em paralelo (Lee et al., 2012). O MR aumenta a tolerância a falhas por meio da replicação e de *checkpoints* frequentes das tarefas concluídas. Todavia, este I/O extra – requerido pela tolerância a falhas – limita a eficiência do MR. Contrariamente, os SGBD relacionais têm como objetivo a eficiência (em detrimento de tolerância a falhas). Para isso, estes sistemas tiram partido do *pipelining* dos resultados intermédios das operações relacionais das *queries*. Porém, esta abordagem constitui um perigo potencial para que uma grande quantidade de operações tenha que ser refeita, caso ocorra uma falha durante a execução da *query*.

Os SGBDs comerciais adotam a estratégia "*one size fits all*" – conforme sugerido por muitos investigadores – e não são adequados para a resolução de tarefas de processamento de dados em grande escala (Lee et al., 2012). Apesar disso, em (Pavlo et al., 2009) demonstra-se uma comparação entre os dois paradigmas. Os testes realizados pelos autores revelam que o Hadoop é entre 2 a 50 vezes mais lento do que os SGBD paralelos. Já para o carregamento de dados, o MR supera os SGBD paralelos, pois cada nodo limita-se a copiar os ficheiros do disco local para o HDFS e distribuir as réplicas desses ficheiros. Os autores do MR argumentam, em (Dean and Ghemawat, 2010), que o estudo realizado por (Pavlo et al.), relativamente ao desempenho, se baseia em suposições erradas do MR (contudo, assumem que existem vários aspetos a melhorar, por exemplo, a latência de inicialização dos trabalhos MR). Eles destacam algumas vantagens do MR sobre os SGBDs paralelos, entre as quais a capacidade de processar e o carregar dados em sistemas heterogéneos, com sistemas de armazenamento de dados diferentes; e a possibilidade de executar funções mais complicadas do que as suportadas diretamente em SQL.

Uma implementação em MR, ao contrário dos SGBDs, no qual os dados são carregados em estruturas previamente definidas (o paradigma relacional), não exige que os dados se encontrem organizados para os analisar (Chaudhuri et al., 2011). Os trabalhos MR são capazes de realizar análises, diretamente e de forma escalável, sobre uma coleção de uma ou mais partições, gerada a partir dos dados de *input*, armazenada no sistema de ficheiros distribuídos. O MR não fornece qualquer tipo de esquema base. Isto pode criar dificuldades insuperáveis, caso o conjunto de dados seja partilhado por várias aplicações. Este conjunto deve ser estruturado para facilitar a sua compreensão por vários programadores. As bases de dados relacionais separam o esquema das

aplicações e armazena-o num conjunto de catálogos do sistema para serem consultados. Assim, as operações, como a adição ou a atualização, não colocam em causa a integridade dos dados. O *framework* MR e o sistema distribuído de ficheiros não têm a noção de restrição (ex. as quantidades sempre positivas). Isto conduz a um risco de perda de consistência dos dados de *input*, dado que esta *framework* não força os programadores a cumprir as restrições que asseguram a exatidão do conjunto de dados (Pavlo et al., 2009).

Em (Stonebraker et al., 2010) argumenta-se que a utilização de sistemas MR para desempenhar tarefas adequadas para os SGBDs produzem resultados menos satisfatórios. O estudo salienta ainda que o MR assemelha-se mais a um sistema *extract-transform-load* (ETL) do que um SGBD, visto que processa e carrega grandes quantidades rapidamente de dados de forma *ad hoc*. Por esse motivo, o MR não é uma tecnologia concorrente, mas um complemento aos SGBDs, já que as bases de dados não são projetadas para executar tarefas ETL de forma eficiente.

3.3.1 Armazenamento de dados

No caso mais comum, os dados de *input* dos trabalhos MR encontram-se armazenados no sistema de ficheiros distribuído, assim como o output gerado por estes trabalhos. O Apache Hadoop (Secção 2.4.) contém uma camada dedicada ao armazenamento (HDFS). Porém, o modelo de processamento MR é independente de qualquer sistema de armazenamento (Lee et al., 2012). Assim, os dados, teoricamente, podem assumir qualquer estrutura. Esta flexibilidade permite lidar com dados não estruturados com mais comodidade do que um SGBD. O programador tem a possibilidade de estruturar os dados como entender, porém este poder obriga a reescrever as análises com frequência.

A configuração padrão do Hadoop contribui para uma degradação do desempenho, porque o Hadoop armazena os dados em HDFS, no mesmo formato textual em que os dados foram gerados. Por conseguinte, esta estratégia de armazenamento obriga o utilizador a produzir um *parser* para analisar os atributos de cada registo do conjunto de dados de *input*. Esta tarefa de análise é imposta, repetidamente, a cada *mapper* e *reducer* e converte os atributos *String* para tipos mais apropriados. O Hadoop proporciona a possibilidade de armazenar os dados em conjuntos de pares chave/valor binários, em ficheiros simples designados por SequenceFile (White, 2012). Todavia, estes ficheiros continuam a impor aos programadores que codifiquem *parsers*, caso o registo este

tenha mais do que um atributo e, assim, o desempenho das consultas diminui com os *parsers* repetitivos realizados em MR. Contrariamente, os SGBDs analisam o conjunto de dados no momento do seu carregamento, evitando assim futuras análises. Este *parser* inicial permite aos SGBDs gerir a disposição dos registos para que os atributos tenham a possibilidade de serem diretamente endereçados durante a execução, da forma mais eficiente. Assim sendo, os registos não têm de ser interpretados durante a execução de uma *query* em bases de dados paralelas.

À semelhança das bases de dados, o MR tem a faculdade de analisar o conjunto de dados, numa fase inicial, e armazena-los em estruturas de dados otimizadas. Isto é, investir algum tempo no carregamento dos dados para, posteriormente, baixar o tempo de execução das *queries* (Stonebraker et al., 2010). Por exemplo, os dados podem ser armazenados (em HDFS) no formato Protocol Buffer[12], uma plataforma neutra e extensível do Google para a serialização de dados estruturados (Dean and Ghemawat, 2010), ou outros formatos, como JSOM, XML, Apache Avro[13] ou Apache Thrift[14]. Estes formatos são capazes de suportar modelos de dados aninhados e não estruturados (Lee et al., 2012). Alternativamente, o armazenamento dos dados pode ser suportado por um SGBD. Para cada nodo, o HadoopDB (Abouzeid et al., 2009) armazena os dados numa base de dados relacional, substituindo a camada de armazenamento do Hadoop (HDFS) por um SGBD com o armazenamento otimizado para dados estruturados.

Como já explicado anteriormente, na Secção 1.2., o armazenamento dos dados cumpre uma função importante na otimização do desempenho de um DW. À semelhança dos SGBDs relacionais, o MR tem a oportunidade de implementar várias técnicas para reduzir o custo I/O, tais como o armazenamento orientado à coluna, a compressão ou os índices (Lee et al., 2012). Estas técnicas cumprem um papel preponderante em DW – suportados por bases de dados relacionais – para ir de encontro a um dos principais requisitos, redução dos tempos de resposta das *queries*.

Linhas e colunas

As bases de dados usam duas formas claras de mapear a informação de uma tabela: linha a linha e coluna a coluna. Estas disposições da informação em disco podem também ser exploradas pelo

[12] https://code.google.com/p/protobuf/
[13] https://avro.apache.org/
[14] https://thrift.apache.org/

modelo de programação MR baseados em SDW para o processamento de dados eficiente (He et al., 2011). Ambos os sistemas de armazenamento e de processamento – Hadoop e SGBD relacionais – partilham as mesmas vantagens e desvantagens relativas às estratégias de armazenamento.

Em HDFS, o armazenamento orientado à linha garante que todos os atributos relativos a um tuplo se encontram localizados no mesmo nodo do *cluster*, isto é, num bloco HDFS. Relativamente ao armazenamento orientado à coluna, otimizado para leituras em DW, uma tabela é partida em vários subconjuntos, que podem conter um atributo ou um grupo de atributos. Esta estratégia evita ler atributos desnecessários para a execução das *queries*, com a capacidade de obter facilmente uma taxa de compressão alta. Porem, a construção de um tuplo pode causar uma sobrecarga nos sistemas baseados em MR e, assim, prejudicar o tempo de processamento das *queries*. Além disso, o armazenamento dos atributos de um tuplo pode localizar-se em nodos diferentes do *cluster*. Por exemplo, a tabela dimensão "Cliente" da Figura 21 é partida em 2: o grupo de atributos "nome" e "idade" (armazenados no Nodo A, Bloco 1); o atributo "*email*" (armazenado no Nodo B, Bloco 2). A reconstrução dos tuplos da tabela dimensão "Cliente" causa uma grande quantidade de dados transferidos através da rede entre os nodos A e B. Estas transferências excessivas sobre a rede podem conduzir a um congestionamento e devem ser evitadas (Dean and Ghemawat, 2004). Um grupo de atributos pode evitar a carga excessiva de uma reconstrução dos tuplos (Stonebraker et al., 2005).

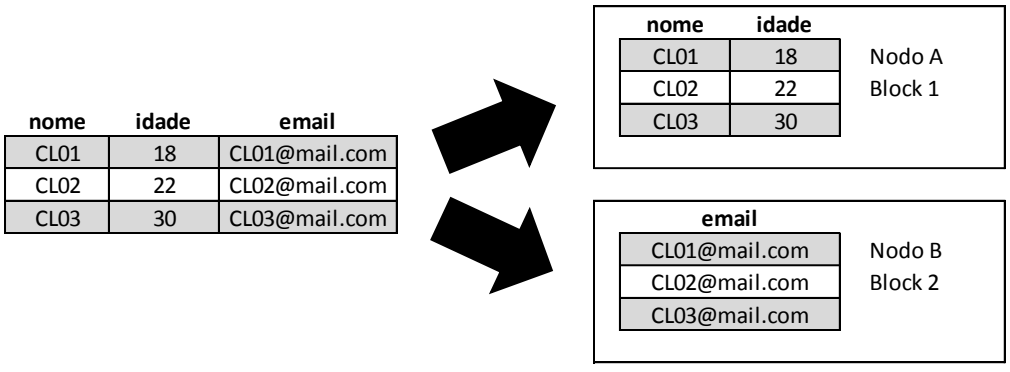


Figura 21: Ilustração do armazenamento orientado à coluna em HDFS

O *Record Columnar File* (RCFile) representa uma estrutura de dados para os SDW baseados em MR (He et al., 2011). Esta estrutura aplica o conceito do PAX (Ailamaki et al., 2001) que se traduz na partição do conjunto de dados de horizontalmente (primeiro) e, em seguida, verticalmente. Este

combina as vantagens dos armazenamentos linha a linha e coluna a coluna: garantir que os atributos do mesmo tuplo ficam localizados no mesmo nodo, logo tem um costo muito reduzido na reconstrução do tuplo (armazenamento orientado à linha); e explorar a compressão e evitar a leitura dos atributos desnecessários (armazenamento orientado à coluna).

O RCFile foi projetado e implementado sobre o HDFS. A Figura 22 demostra um exemplo de como os dados de uma tabela são armazenados em RCFile. Consoante a estrutura do HDFS, cada tabela pode encontrar-se partida por vários blocos e, para cada bloco, os tuplos são armazenados em grupos. Isto é, a tabela é dividida em grupos de tuplos, todos com o mesmo tamanho. Um bloco HDFS pode conter um ou vários grupos de tuplos, dependendo dos tamanhos dos grupos de linhas e dos blocos HDFS. Um grupo de linhas inclui três secções: um marcador de sincronização, um cabeçalho de metadados e a secção que armazena os dados da tabela. O marcador de sincronização é utilizado sobretudo para a divisão de dois grupos de linhas contínuas num bloco HDFS. O cabeçalho de metadados armazena a informação acerca do grupo (ex. quantos tuplos compõem o grupo, espaço em disco ocupado pelos atributos, entre outros). Por último, a secção dos dados da tabela com um armazenamento orientado à coluna. A Figura 22 ilustra o armazenamento dos valores do atributo "A", na primeira parte da secção dos dados, em seguida os valores do atributo "B", na segunda parte da secção dos dados, e assim por diante. As secções cabeçalho de metadados e dados da tabela podem ser comprimidas.

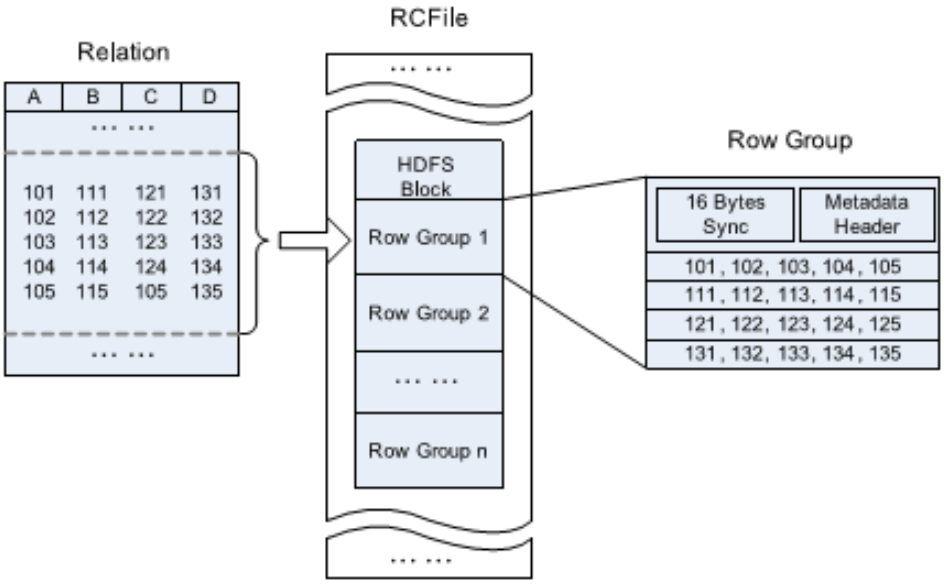


Figura 22: Disposição dos dados do RCFile num bloco HDFS (retirado de (He et al., 2011))

Índices

O MR pode tirar partido de índices pré-gerados (Dean and Ghemawat, 2010).A Bigtable, do mesmo modo que o HBase (descrito como a base de dados do Hadoop (Lars, 2011)), permite indexar um conjunto de dados através de uma chave linha e colunas (ver a Secção 2.4.2.). Os dados são armazenados, arbitrariamente, em *Strings* que podem assumir diversas formas. Caso seja necessário analisar um subconjunto de dados contido no HBase, basta consultar um grupo de chaves linha que representa este subconjunto em vez de varrer o conjunto de dados completo. O HBase tem a capacidade de armazenar os dados do mesmo modo que os SBGDs orientados à coluna (ex. Vertica[15]), assim, este lê apenas os atributos necessários para a análise e utilizar a compressão de dados.

Individualmente, uma chave linha é unívoca e pode endereçar uma ou mais colunas (a unidade mais fundamental em HBase). Uma tabela é formada por um conjunto de chaves linha. As colunas podem conter várias versões, com cada valor distinto armazenado discriminadamente em células. Esta descrição assemelha-se ao modelo relacional utilizado pelas bases de dados típicas, no entanto o HBase adiciona características como uma dimensão extra (a capacidade de uma célula ter vários valores), as chaves linha encontram-se, sempre, ordenadas lexicograficamente, entre outras. O HBase tem a possibilidade acomodar, na sua arquitetura, as relações um para um, um para muitos e muitos para muitos. O suporte de dados esparsos e desnormalizados, bem como o armazenamento coluna a coluna, do HBase elemina a necessidade de utilizar as operações de junção custosas para agregar os dados no momento da consulta.

O ficheiro HFile é utilizado para o armazenamento de dado do HBase. Dentro destes ficheiros, encontram-se sucessões de blocos com um *block index* armazenado no final. O *block index* é usado para localizar os blocos (o índice é carregado e preservado em memória, quando o HFile é acedido). O acesso a estes ficheiros podem ser a valores específicos, assim como varrer um intervalo de valores definido pelas chaves linha inicial e final. As pesquisas podem ser executadas com um acesso ao disco: primeiro, localiza-se o bloco indicado pela execução de uma pesquisa binária do índice em memória, e ler o bloco apropriado do disco (Lars, 2011).

[15] http://www.vertica.com/

3.3.2 Operações sobre os dados

O MR é uma ferramenta de processamento pura, que lê pares chave/valor a partir do sistema de armazenamento, através dos *mappers*. Os *mappers* leem os registos do sistema de armazenamento e "quebra-o" em pares chave/valor – independentemente da estrutura utilizada pelo conjunto de dados – para o processamento posterior. Este formato é definido explicitamente pelo programador, que tem a oportunidade de explorar estruturas de dados mais complexas, tal como as chaves compostas. Os operadores relacionais como a projeção, seleção, junção e agregação oferecem muitas oportunidades para o paralelismo (operações exigidas para o suporte à decisão). O paradigma básico consiste no paralelismo de dados, isto é, aplicar os operadores relacionais sobre subconjuntos de dados separados – partições – e, no final, combinar os resultados obtidos (Chaudhuri et al., 2011). Dado o sucesso do MR no processamento de consultas analíticas complexas, este deve ser examinado para a manipulação de dados relacionais (Lin and Dyer, 2010).

Seleção (Where)

Em MR, a seleção de valores é uma operação muito simples. A função *map* percorre todos os registos e emite aqueles que satisfazem uma determinada condição (Figura 23). A seleção não necessita da fase *reduce*.

```
map (String chave, Registo r):
     Se r satisfaz a condição Então
          emitirPar(Registo r, null);
```

Figura 23: Exemplo de uma operação de seleção em MR

Projeção (*Select*)

A projeção de atributos é um pouco mais complexa do que a seleção. Esta operação recebe uma relação e devolve uma outra relação com uma lista de atributos específicos da relação original. A função *map* extrai os valores dos atributos a projetar e emite-os como chave. O *reducer* limita-se a eliminar registos repetidos (Figura 24). Tal como a seleção, a operação projeção também pode ser implementada por *mappers*.

```
map (String chave, Registo r):
     Registo s = extrairAtributos(r);
     emitirPar(Registo s, null);

reduce (Registo r, Lista<null> l):
     Emitir(Registo r, null);
```

Figura 24: Exemplo de uma operação de projeção em MR

Agrupamento e agregação (Group By)

O MR é simples e eficiente para processar agregações. Desta forma, este modelo de programação é muitas vezes comparado com o processamento de uma *query* "seleção seguido de agrupamento e agregação", de um SGBDs relacional (Lee et al., 2012). O agrupamento e agregação podem ser executados por um trabalho MR (Figura 25). Para cada registo, a função *map* extrai os valores dos atributos a agrupar (*groupBy*) e os valores dos atributos a agregar (*aggregateBy*), para emitir o par <valores a agrupar, valores a agregar>. O *reducer* recebe os valores a agregar, já agrupados, e aplica-lhe a função de agregação. Durante o processo de agrupamento, o *framework* MR ordena os "valores a agrupar" a automaticamente e de forma lexicográfica (porém, o utilizador pode alterar a forma de ordenar as chaves dos pares produzidos pela função *map*). Assim, uma operação *"Order By"* é, também executada na fase *"shuffle & sort"*.

```
map (String chave, Registo r):
     Atributo groupBy = extrairAtributosGroupBy(r);
     Atributo aggregateBy = extrairAtributosAggregateBy(r);
     emitirPar(Atributo groupBy, Atributo aggregateBy);

reduce (Atributo groupBy, Lista<Atributo> aggregateBy):
     Valor resultadoAgregação = funçãoAgregação(aggregateBy);
     Emitir(Atributo groupBy, Valor resultadoAgregação);
```

Figura 25: Exemplo de uma operação de agregação em MR

Junções (*Join*)

O MR é utilizado para várias aplicações analíticas de dados. A execução de uma análise de dados (ex. o processamento de ficheiros *log*) inclui operações, tal como o cruzamento de diversos conjuntos de dados. O *framework* MR não foi inicialmente projetado para combinar a informação de duas ou mais fontes de dados e, dessa forma, as operações de junção são trabalhosas. Alguns dos algoritmos de junção bem conhecidos foram adaptados (com alguma dificuldade) para terem a

possibilidade de executar em MR, tirando proveito do estudo dos algoritmos de junção em bases de dados paralelas e distribuídas com a arquitetura *shared-nothing* (Blanas et al., 2010). Por exemplo, um *two-way join* combina os tuplos de dois conjuntos de dados baseado num atributo. A expressão seguinte:

$$Data \bowtie_{Data.KeyD = Vendas.KeyD} Vendas$$

representa um *equi-join* (a junção dos tuplos com o mesmo valor) entre as tabelas "Data" e "Vendas" baseado no atributo "KeyD". Desta operação resulta um terceiro conjunto de dados com os tuplos que satisfazem a condição de junção. Em MR, estas junções são perfeitamente possíveis de executar. O MR implementa várias estratégias de junção, sem alterar a *framework*, tais como as estratégias básicas (Blanas et al., 2010): a *repartition join* e a *broadcast join*.

A *repartition join* é a estratégia de junção mais comum em MR. Nesta estratégia, as tabelas são dinamicamente particionadas pelo atributo de junção, com os pares correspondentes a esta partição reunidos. A Figura 26 ilustra a estratégia de junção *repartition join*, semelhante à estratégia de junção *sort-merge* particionada dos SGBDs paralelos. O algoritmo *repartition join* é fornecido pelo Hadoop.

A estratégia *repartition join* pode ser implementada por um trabalho MR. A fase *map* varre as duas tabelas (do HDFS) para extrair as chaves dos registos e marca-o com uma *tag* para identificar a tabela a que pertence. No final, o *map* emite a chave extraída e o registo marcado como um par. O conjunto de pares resultante do *mapper* é particionado, ordenado e agrupado automaticamente pela chave de junção pela *framework*. Para cada chave de junção, o reducer recebe todos os registos e separa-os em dois grupos, de acordo com a sua *tag*. No final, o *reducer* cruza os dois conjuntos e emite os registos resultantes da junção.

Esta abordagem tem algumas desvantagens, entre estas o gasto de memória. Todos os registos de ambas as tabelas, de uma determinada chave de junção, têm de ser armazenados em memória, mesmo as chaves de junção que não têm par. A dimensão das partições podem variar (*data skew*) e, assim, as partições com uma dimensão elevada pode não caber na memória Perante estes problemas, esta estratégia sofreu alguns ajustes importantes, como:

- Na função *map*, a chave emitida é alterada para uma composição <chave de junção, *tag* da tabela>. Para cada chave de junção, a *tag* é gerada de forma a garantir que os registos da tabela "Data" permaneçam à frente dos registos da tabela "Vendas".
- A função de particionamento é modificada para que o *hashcode* seja calculado a partir da chave de junção. Assim, os registos de uma determinada chave de junção são atribuídos ao mesmo *reducer*.
- A função de agregação do *reducer* é também modificada para que os registos sejam reunidos baseado na chave de junção. Nesta agregação, a parte da chave correspondente à *tag* da tabela é ignorada.
- Para uma chave de junção, os registos encontram-se ordenados pela *tag* da tabela. Desta forma, os registos da tabela "Date" (a tabela mais pequena) serão atendidos primeiro. O *reducer* lê os registos da tabela "Date" e armazena-os na memória. Os registos da segunda tabela (tabela "Vendas") são lidos diretamente do HDFS e, para cada registo, é executada a junção com os registos da mesma chave de junção da tabela "Date" (em memória).

Estas melhorias originaram uma nova estratégia, a *improved repartition join*, que corrige os problemas de memória da versão básica da estratégia *repartition join*. No entanto, tanto o *repartition join* como o *improved repartition join* têm de ordenar e enviar (através da rede) as tabelas durante a fase de particionamento e agregação por chave do MR. Este *overhead* pode provocar alguma degradação de desempenho.

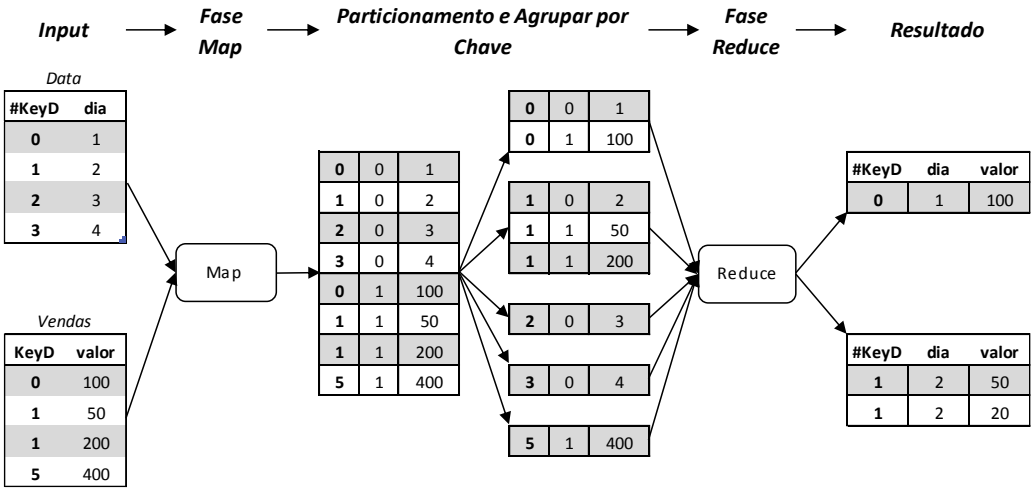


Figura 26: Algoritmo de junção *repartition join*

Em grande parte das aplicações, a dimensão de uma das tabelas é muito menor (no nosso caso, a tabela dimensão "Data"). Assim, com a possibilidade desta tabela caber em memória, pretende-se evitar a ordenação das duas tabelas e o *overhead* que ocorre com transmissão da tabela maior (a tabela de factos "Vendas") através da rede, como os algoritmos baseados no *repartition join* implementam.

A estratégia *broadcast join* é executada apenas na fase *map*. O conjunto de dados mais pequeno é replicado por todas as máquinas, para que cada *mapper* carregue os dados deste conjunto para a memória a partir do disco local. Esta operação é executada para cada *mapper* pela função *init* (função *setup*[16], em Hadoop) e pode se tornar inviável, se o tamanho do conjunto de dados replicado for muito grande. A função *init* começa por verificar se existe uma cópia da tabela "Date" no disco local, caso contrário copia a tabela do HDFS, particiona a tabela na chave de junção e armazena estas partições no disco local. Esta abordagem visa evitar carregar todas as partições da tabela "Date" para a memória, numa tabela de *hash*, durante a junção.

O *broadcast join* determina, dinamicamente, qual a tabela que vai construir uma tabela de *hash*. Se a tabela "Data" é menor do que parte da tabela "Vendas", a função *init* é utilizada para carregar todas as partições de "Data" em memória para construir a tabela de *hash*. Logo após, a função map extrai a chave de junção dos registos da tabela "Vendas" e, para cada registo usa esta chave para consultar a tabela de *hash* e originar o *output* da junção. Caso contrário, a parte da tabela Vendas é menor do que a tabela "Data", a junção não é realizada na função *map*. Esta particiona a tabela "Vendas" da mesma forma do que a tabela "Data". Depois, a função *close* (função *cleanup*[17], em Hadoop) junta as partições correspondentes das tabelas "Data" e "Vendas". O carregamento das partições da tabela "Data" é poupado, caso a partição da tabela "Vendas" correspondentes não tiverem registos. Esta otimização é útil para o caso do domínio da chave de junção ser muito grande. Tanto a tabela "Data" como parte da tabela "Vendas" podem ser escolhidas para construir a tabela de *hash*, porque se presume que cabe em memória. O tamanho típico de uma parte é menor do que 100 MB.

[16] http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Mapper.html
[17] http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Mapper.html

As junções típicas das aplicações DW (*star join*), os dados das tabelas dimensão são reduzidas com base em restrições, em seguida o produto de várias tabelas dimensão são unidas com a tabela de factos. Os algoritmos apresentados acima implementam uma junção simples, ou seja, a junção entre duas tabelas. A junção com mais de duas tabelas (*Multi-way Join*) pode ser expressa, por exemplo:

$$Data \bowtie_{Data.KeyD = Vendas.KeyD} Vendas \bowtie_{Vendas.KeyP = Produto.KeyP} Produto$$

A extensão dos algoritmos *repartition join* e *broadcast join* podem implementar estas junções (Blanas et al., 2010). A junção acima apresentada pode ser implementada por uma sequência de dois *two-way joins*. A primeira junção pode dividir-se em dois planos de execução: executar a junção entre as tabelas "Data" e "Venda" e, depois, juntar a tabela "Produto" com o resultado obtido da primeira junção; ou executar a junção entre as tabelas "Venda" e "Produto" e depois juntar a tabela "Data". Ambos os planos de execução podem ser implementados pela estratégia *repartition join*. A execução destas junções implica criar vários trabalhos MR para unir os conjuntos de dados, dois de cada vez. Um algoritmo alternativo envolve juntar as três tabelas num único trabalho MR. Os *mappers* enviam cada tuplo das tabelas "Data" e "Venda" para os *reducers*, contudo os tuplos da tabela "Produto" são enviados, apenas, para um *reducer*. A duplicação dos dados aumenta o custo da transmissão, mas traz vantagens pelo facto de não transferir o resultado da primeira junção. O algoritmo *broadcast join* processa, simultaneamente, a junção de várias tabelas.

Como os atributos em comum de um *star join* são as chaves das tabelas dimensão, não é esperado que a dimensão da junção aumente muito. Normalmente, as tabelas de facto são enormes, com um volume muito superior às tabelas dimensão, dessa forma, é claramente benéfico não ter de transmitir o resultado das junções intermédias (*output* gerado por uma junção), onde a junção é realizada entre tabela de factos e cada tabela dimensão. Mesmo replicando de forma significativa as tabelas de dimensão, o custo da comunicação das tabelas dimensão pode ser bem menor do que o custo de transmissão da tabela de factos. O Aster Data particiona a tabela de factos através dos nodos replicando as tabelas de dimensão por todos os nodos, em que cada tuplo, das tabelas dimensão, está unido com um ou mais tuplos da tabela de factos. O particionamento da tabela de factos tem em consideração os valores particulares dos dados (Afrati and Ullman, 2010).

3.3.3 Implementações

Os SDWs baseados em MR, nomeadamente na *framework* Apache Hadoop, têm sido implementados com sucesso em grandes serviços *Web*. Estes desempenham diariamente um papel essencial na execução de análises *clickstreams* ou no suporte a aplicações de mineração de dados, entre outros. Dois dos SDW baseados em Hadoop extensamente utilizados são o Apache Hive e o Apache Pig (He et al., 2011). Estes sistemas usam o HDFS como camada de armazenamento, assim como linguagens script que permitem realizar as análises sobre os dados. Outro sistema, o HadoopDB (Abouzeid et al., 2009), reutiliza grande parte do Apache Hive, exceto o armazenamento de dados. Este utiliza um SGBD em cada nodo em vez de um sistema de ficheiros distribuído, conforme referido anteriormente (Secção 3.3.1.).

O Apache Pig

O Apache Pig (Olston et al., 2008) é uma plataforma orientada para a análise de grandes conjuntos de dados. A análise expressa-se através de uma linguagem de alto nível, chamada Pig Latin, que cria programas capazes de executarem em ambientes paralelos. Os programas, desenvolvidos em Pig Latin, são compilados em trabalhos MR e executados servindo-se do Hadoop. Um programa desenvolvido em Pig Latim é composto por uma sucessão de operações, ou transformações, aplicadas aos dados de *input* para produzirem um output. As operações traçam o fluxo de dados que o Pig traduz num plano de execução. O Pig foi inicialmente desenvolvido pelo Yahoo! e, mais tarde, tornou-se parte da Apache Software Foundation. O modelo de dados do Pig é constituído por 4 tipos:

- ***atom***, que contém um valor atómico simples (ex. '1' ou 'ProdA');
- ***tuple***, que corresponde a uma sequência de elementos, cada um destes capar de representar qualquer tipo de dados e.g., ('ProdA', '1');
- ***bag***, que é um conjunto de tuplos com possíveis duplicados. Os tuplos não têm necessariamente de seguir um esquema; esta flexibilidade permite que os tuplos "('ProdA', '1')" e "('ProdA', ('CAT01', '10'))" pertençam a um *bag*;
- ***map***, que é uma coleção de pares chave/valor. O seu esquema é igual ao bag, no entanto as chaves são atómicas, o que torna as pesquisas mais eficientes. Uma chave pode mapear um *bag* ou simplesmente um valor atómico.

Um processo de análise começa com a especificação dos ficheiros de entrada e como o seu conteúdo será convertido para o modelo de dados do Pig. Normalmente, assume-se que o conteúdo de um ficheiro é composto por uma sequência de tuplos, ou seja, um *bag*. Por exemplo, o ficheiro que representa a tabela *Produto* pode ser representado pelo seguinte *bag*, que resulta do comando LOAD do Pig Latin:

$$(1, 'ProdA', 'CAT01')$$
$$(2, 'ProdB', 'CAT02')$$
$$(3, 'ProdC', 'CAT03')$$
$$(4, 'ProdD', 'CAT02')$$

Um tuplo representa uma linha da tabela, em que cada elemento do tuplo representa um valor, separado por vírgulas, inserido na linha. O comando LOAD, no qual se especifica o ficheiro e a forma como deve ser lido, não implica o carregamento dos dados em tabelas. Este retorna um *bag*, que representa o modelo lógico dos dados do ficheiro e que é atribuído a uma variável para servir de *input* a comandos sequentes, como o próximo comando, o *GROUP*, agrupa os tuplos por *categoria*:

$$('CAT01', \{(1, 'ProdA')\})$$
$$('CAT02', \{(2, 'ProdB'), (4, 'ProdD')\})$$
$$('CAT03', \{(3, 'ProdC')\})$$

Após o agrupamento, cada tuplo representa um valor diferente da *categoria* - primeiro campo representa a *categoria*; o segundo campo representa um *bag* desordenado com os tuplos para a respetiva *categoria*: ("KeyP", "nome"). O resultado do comando *GROUP* é atribuído a outra variável para dar seguimento à análise dos dados (o resultado pode também ser materializado num ficheiro através do comando *STORE*). Este resultado é representado por uma estrutura de dados aninhados map< *categoria*, lista< tuplo >>. Por outro lado, as bases de dados relacionais utilizam apenas tabelas, a não ser que se viole a primeira forma normal. Para sustentar a informação resultante da operação *GROUP* acima, em conformidade com a primeira forma normal, é

necessário normalizar os dados por duas tabelas (no entanto, esta estrutura pode ser reconstruída através da junção das tabelas em "CategoriaID"):

$$\boldsymbol{categoria}\ (CategoriaID, descrição, \dots)$$
$$\boldsymbol{produto}\ (KeyP, CategoriaID, nome, \dots)$$

Para além do comando *GROUP*, o Pig Latin também suporta outras operações relacionais, como o *FILTER*, o *ORDER*, o *JOIN* ou o *GROUP*. O *FILTER* retém um subconjunto de dados de interesse (correspondente à seleção). A projeção é executada pelo comando *LOAD* ou durante a análise. As junções ficam encarregues ao operador *JOIN*. Este executa um *equi-join* de duas ou mais relações baseadas num atributo em comum. O Apache Pig implementa as estratégias de junção *repartition join* e *broadcast join* (Blanas et al., 2010). As agregações necessitam de dois comandos: o *GROUP* e o *FOREACH* (aplica um grupo de operações a cada registo de um conjunto de dados). Ao contrário do que se verifica no SQL, no Pig Latin não existe uma conexão direta entre o agrupamento e as funções de agregação. O operador *GROUP*, como é exposto acima, limita-se a recolher todos os registos com o mesmo valor chave num *bag* (Gates, 2011). Por exemplo, a agregação retorna o número de produtos para cada categoria:

$$grupo\_cat = GROUP\ produtos\ BY\ categoria;$$
$$count = FOREACH\ grupo\_cat\ GENERATE\ categoria, COUNT(produtos);$$

O Pig Latin é descrito como uma linguagem capaz de expressar *data flows*. Esta linguagem oferece a possibilidade de construir uma série de operações declarativas, bem como o utilizador desenvolver as suas próprias funções de leitura processamento e escrita. As classes mais comuns do processamento feito pelo Apache Pig envolve o ETL, o processamento iterativo (Gates, 2011), ou o cálculo de vários tipos de agregações *roll-up* sobre registos de atividades dos utilizadores (*logs*), ou outros conjuntos de dados. A principal motivação para utilizar o Pig para estas agregações resume-se ao facto destes conjuntos de dados caracterizam-se por ser excessivamente volumosos e contínuos para serem tratados e carregados numa SGBD relacional. Um exemplo é o cálculo de termos pesquisados agregados ao longo dos dias, semanas, ou meses, e também sobre as localizações geográficas (implicitamente pelo endereço de IP). Algumas tarefas exigem duas ou mais agregações sucessivas, outras trefas impõem uma junção seguido de uma agregação. Para estes casos, o Pig prepara uma sequência de trabalhos MR (Olston et al., 2008).

Apache Hive

O Hive (Thusoo et al., 2009) é uma implementação *open source* de um DW em cima do Hadoop que converte uma linguagem declarativa, idêntica ao SQL, numa sequência de trabalhos MR. Assim, não é necessário de escrever código Java para realizar consultas aos dados armazenados em HDFS. Este foi criado pelo Facebook que substituiu um DW, implementado por uma base de dados relacional, inadequado para o processamento diário dos dados (algumas tarefas tinham uma duração superior a um dia). A necessidade de uma infraestrutura capaz de escalar juntamente com os seus dados, levou a adotar o Hadoop para o processamento dos dados. Desta forma, os trabalhos que duravam dias a concluir são agora completados em poucas horas. A sua arquitetura inclui um catálogo de sistema – Metastore – que contém os esquemas e estatísticas, que são úteis para a exploração dos dados, a otimização de *queries* e a compilação de *queries*.

As diferenças entre o modelo de dados, os tipos de dados e a linguagem de pesquisa do Hive, e das bases de dados tradicionais são pequenas. O Hive estrutura os dados em alguns dos conceitos encontrados em bases de dados, como tabelas, colunas, linhas e partições. Este suporta tipos de dados primitivos (inteiro, vírgula flutuante, string, data e booleano), bem como tipos complexos (lista, *map* e estruturas (*structs*)) e composições aninhadas destes (ex. lista <map <inteiro, struct <p1:int, p2:int>>>), que podem ser esquematizados por uma tabela. O Hive também permite que o utilizador defina tipos de dados e funções. A linguagem de pesquisa HiveQL é idêntica ao SQL e, consequentemente pode ser facilmente compreendida e utilizada pelos utilizadores habituais do SQL para efetuar análises no Hive.

A linguagem de pesquisa HiveQL é constituída por um subconjunto de comandos SQL e algumas extensões úteis para suportar análises expressas em programas MR codificados pelo utilizador, numa linguagem de programação à sua escolha, que podem ser ligados às consultas em HiveQL sem problemas. As operações relacionais habituais do SQL, como a seleção, a junção, o agrupamento, a agregação, a criação de tabelas e as funções úteis sobre os tipos de dados primitivos e complexos, assemelham-se (ou são exatamente iguais) às operações encontradas no HiveQL. Os metadados úteis, como a descrição de tabelas, também fazem parte, bem como a capacidade de inspecionar os planos de execução das *queries* (ainda que estes planos de execução tenham uma aparência bastante diferente aos encontrados nos SGBDs relacionais tradicionais).

Uma tabela em Hive é logicamente construídas pelo conjunto de dados que armazena e os metadados associados que descreve o seu *layout*. Os dados residem em HDFS e são mapeados para uma diretoria, por exemplo:

$$< diretoria\ raiz >/data$$
$$< diretoria\ raiz >/data$$

As tabelas podem ser particionadas por um ou mais atributos. As partições determinam a distribuição dos dados por subdiretorias, dentro da diretoria da tabela. Por exemplo, se a tabela "data" é particionada pelo atributo "ano" (Figura 11), os dados do ano '2000' são armazenados na diretoria com o caminho (sem espaços):

$$< diretoria\ raiz >/data/ano = 2000$$

As partições são criadas com juntamente com a tabela, mas existe a possibilidade de, posteriormente, adicionar ou remover partições. Existe uma partição para cada valor distinto do atributo selecionado. Os atributos do particionamento não fazem parte dos dados da tabela e os seus valores são codificados no caminho da diretoria da respetiva partição (estes são também armazenados nos metadados da tabela). Com a adição do atributo mês, a diretoria adota o caminho:

$$< diretoria\ raiz >/data/ano = 2000/\ mes = 01$$
$$< diretoria\ raiz >/data/ano = 2000/\ mes = 02$$
$$\ldots$$
$$< diretoria\ raiz >/data/ano = 2000/\ mes = 12$$

Existe outra abordagem para organizar os dados de uma tabela (diretorias) ou as partições (subdiretorias), o *bucket*. O utilizador seleciona um atributo da tabela para determinar cada *bucket* que corresponde a um ficheiro no interior de uma diretoria. No momento de criar uma tabela, o utilizador pode especificar um atributo e o número de *buckets*. Para cada registo da tabela, o Hive usa um atributo (selecionado pelo utilizador) para calcular o *bucket* a que o registo pertence, através de uma função de *hash*. A gama de valores corresponde ao número de *buckets* definido

também pelo utilizador (White, 2012). Por exemplo, a tabela "data" é composta pelas instâncias seguintes:

#KeyD	data	mês	ano
0	01/01/2000	1	2000
1	02/01/2000	1	2000
2	03/01/2000	1	2000
3	04/01/2000	1	2000

Figura 27: Instância da tabela "data"

O atributo selecionado é o "KeyD"; o número de *buckets* é '2'. Assim, o primeiro *bucket* contém os registos com a chave '0' e '2', já que o resto da divisão destes valores com '2' é igual a '0'. Os restantes registos pertencem ao segundo *bucket*, uma vez que o resto é '1'. No interior de cada *bucket*, os dados podem estar ordenados por um ou mais atributos.

Estas estratégias permitem ao Hive otimizar o tempo de acesso aos dados. O compilador do Hive usa a informação sobre o particionamento de dados de uma tabela – os metadados – para "podar" as diretorias que necessitam de ser acedidas, com o objetivo de avaliar uma *query*. Por exemplo, a *query*

$$SELECT * FROM\ data\ WHERE\ ano = 2000\ AND\ mes = 01;$$

só irá varrer os ficheiros dento da diretoria de caminho "<diretoria raiz>/data/ano=2000/mês=01". Esta "poda" dos dados tem um impacto significativo no tempo que demora a executar uma *query*. De certa forma, este esquema de particionamento é semelhante aos utilizados por muitas bases de dados (Thusoo et al., 2009). Os *buckets* impõem uma estrutura de dados extra na tabela que o Hive pode tirar proveito para a execução de algumas *queries*, em especial a junção de duas tabelas que criaram *buckets* a partir do mesmo atributo. Assim, estas junções podem ser executadas de forma eficiente com os algoritmos de junção da fase *map* (White, 2012).

O Hive pode utilizar uma implementação da interface Java SerDe (Serialização e desserialização de Objectos) fornecida pelo utilizador e associá-la a uma tabela ou partição. Em consequência, o formato dos dados pode ser facilmente interpretado e consultado. Por omissão, a implementação SerDe padrão é o LazySerDe (assume que os dados são armazenados num ficheiro de tal modo

que as linhas são delimitadas pelo caracter *newline* e os atributos delimitados por *ctrl-A (ascii code 1)* ou outro caracter). Relativamente ao armazenamento de dados, os ficheiros Hadoop podem ser armazenados em vários formatos, que especificam como os registos se encontram dispostos dentro destes. O Hadoop suporta ficheiros de texto (ex. armazenados em TextInputFormat[18]), ficheiros binários (ex. armazenados em SequenceFileInputFormat[19]), ou ficheiros formatados pelo utilizador. O Hive não impõe qualquer tipo de formato dos ficheiros de entrada. O Hive tem também a capacidade de utilizar o RCFile para armazenar os dados (He et al., 2011).

A arquitetura do Hive (Thusoo et al., 2009) é composta por vários componentes: o Metastore, o Query Compiler, o Execution Engine, entre outros. O Metastore armazena num SBGD relacional o catálogo de sistema e os metadados acerca das tabelas, atributos, partições, etc. este componente é muito importante para o Hive, uma vez que impõe uma estrutura aos ficheiros Hadoop. O Query Compiler utiliza a informação armazenada no Metastore para gerar o plano de execução. Este é semelhante aos compiladores das bases de dados tradicionais, ou seja, analisa e compila o HiveQL para produzir um plano lógico de execução sobre a forma de um grafo acíclico dirigido (*directed acyclic graph*) de tarefas *map* e *reduce*. O Execution Engine é o componente que executa as tarefas produzidas pelo compilador pela ordem das suas dependências, interagindo com o Hadoop.

A fase de otimização do componente Query Compiler envolve uma cadeia de transformações que produz o plano lógico de execução. Essas transformações são:

- Projetar apenas as colunas necessárias para o processamento da *query*.
- As seleções são executadas, se possível, numa fase inicial do processamento para filtrar os registos que não satisfazem os predicados.
- Eliminar os ficheiros das partições através dos atributos do particionamento que não satisfazem os predicados.
- Junções na fase *map*, através da sugestão MAPJOIN (tabela).
- Assegurar que a operação de junção não exceda a memória do *Reducer*. Para isso, as tabelas de maior dimensão são transmitidas e não materializadas em memória do *Reducer*, ao contrário das tabelas de menor dimensão que são mantidas em memória.

[18] https://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/lib/input/TextInputFormat.html
[19] https://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/lib/input/SequenceFileInputFormat.html

Além da sugestão para utilizar a junção na fase *map*, o utilizador pode também fornecer outras sugestões ou definir parâmetros para melhorar o desempenho do sistema. Pode-o fazer em situações como: um conjunto de dados assimétrico (*data skew*), através da repartição dos dados no processamento de um agrupamento (*Group By*); ou com a redução da quantidade de dados transferidos entre os *mappers* e *reducers*, através de agregações parciais na fase *map*.

Capítulo 4

Um Caso de Estudo

4.1 Star Schema Benchmark

O DW selecionado para estudar a utilização de MR em sistemas de suporte à decisão foi o *Star Schema Benchmark* (O'Neil et al., 2007), que se baseia no teste de referência do TPC-H (Poess and Floyd, 2000). O estudo que iremos descrever consistiu num conjunto de *queries* adaptadas do TPC-H, especialmente orientadas a um negócio específico, com um elevado grau de complexidade, para a análise de um vasto conjunto de dados.

O esquema do TPC-H apresenta uma estrutura composta por oito tabelas, das quais seis são tabelas de dimensão, que se encontram normalizadas até à terceira forma normal, e duas tabelas de factos. No SSB, estas tabelas de factos ("ORDERS" e "LINEITEM") são combinadas, prática vulgar em DWs (Kimball and Ross, 2002), para originar uma nova tabela de factos, designada por "LINEORDER" que evita junções desnecessárias. As restantes alterações efetuadas, esclarecidas em (O'Neil et al., 2007), para obter o esquema em estrela, o *Star Schema Benchmark* (SSB), são: a remoção da tabela dimensão "PARTSUPP" e a adição da tabela dimensão "DATE".

O SSB é composto por uma tabela de factos, chamada "LINEORDER", com duas dimensões degeneradas, "ORDERKEY" e "LINENUMBER", e quatro dimensões, "DATE", "PART", "CUSTOMER" e "SUPPLIER" (Figura 28). A tabela "LINEORDER" tem 17 colunas, com a informação das vendas individuais, e uma chave primária composta constituída pelos atributos "ORDERKEY" e

"LINENUMBER". As restantes colunas da tabela integram 5 chaves estrangeiras, referentes às tabelas dimensão "DATE" ("L_ORDERDATE" e "L_COMMITDATE"), "PART" ("L_PARTKEY"), "CUSTOMER" ("L_CUSTKEY") e "SUPPLIER" ("L_SUPPKEY") e os atributos relativos às vendas, incluindo "L_QUANTITY", "L_DISCOUNT", "L_REVENUE", entre outros.

Do mesmo modo que o TPC-H, cada tabela do SSB tem um fator de escala (*scale factor*) que permite às tabelas escalarem até uma determinada dimensão. Por exemplo, um *scale factor* = 2 gera uma tabela de factos e uma tabela dimensão "PART" com 12.000.000 e 400.000 registos, respetivamente.

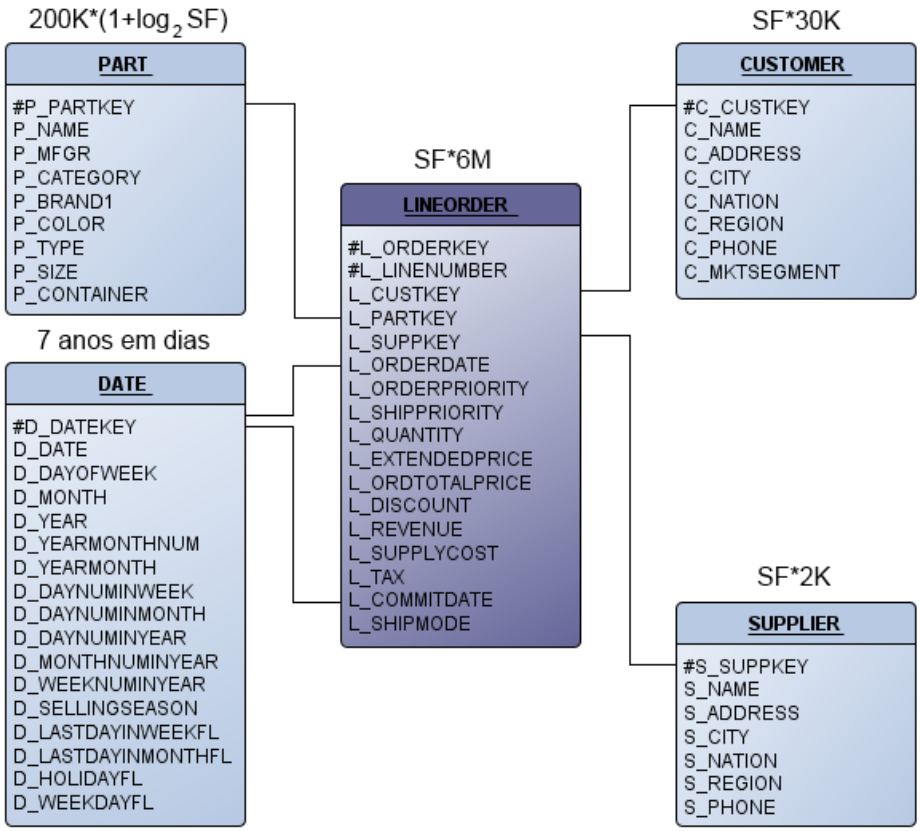


Figura 28: *Star Schema Benchmark* (adaptado de (O'Neil et al., 2007))

O grão da tabela de factos "LINEORDER" corresponde à venda diária de um produto, de um dado fornecedor a um dado cliente. As tabelas das dimensões são simples, à exceção da tabela "DATE" que é do tipo *Role-playing dimension,* pelo facto de representar dois papéis diferentes na tabela de

factos: a data de encomenda e a data de entrega prometida ao cliente. As hierarquias definidas para as dimensões são, respetivamente:

- **DATE**: YEARMONTH -> YEAR
- **PART**: BRAND1 -> CATEGORY -> MFGR
- **CUSTOMER** e **SUPPLIER**: CITY -> NATION -> REGION

O conjunto de *queries* do SSB (Apêndice B) encontra-se dividido em quatro categorias ou *"flights"*, que são: Q1, Q2, Q3 e Q4. No total, este conjunto é formado por treze *queries*, representando cargas de trabalho típicas de um DW. O conjunto de *queries* do SSB usa o formato do TPC-H, na tentativa de fornecer as características de *Functional Coverage* e *Selectivity Coverage*, isto é, um conjunto de *queries* que abranja as tarefas tipicamente realizadas sobre um esquema em estrela e que varie de acordo com a consulta de diferentes parcelas da tabela de factos. As 4 categorias definidas foram as seguintes:

- **Categoria 1 (3 *queries*):** as *queries* têm restrições aplicadas sobre a tabela dimensão "DATE" e sobre a tabela de factos para os atributos "L_DISCOUNT" e "L_QUANTITY". Estas *queries* calculam os ganhos em receita ("L_EXTENDEDPRICE – L_DISCOUNT") para vários níveis de descontos e quantidades.
- **Categoria 2 (3 *queries*):** as *queries* têm restrições em duas tabelas dimensão: "PART" e "SUPPLIER". Estas *queries* calculam o somatório das receitas para as categorias dos produtos, em determinadas regiões, agrupados por ano e marca do produto.
- **Categoria 3 (4 *queries*):** as *queries* têm restrições aplicadas em três tabelas dimensão: "DATE", "CUSTOMER" e "SUPPLIER". Estas *queries* envolvem o somatório das receitas para uma determinada região e por um período de anos, agrupados por cidade do cliente, cidade do fornecedor e ano. Os resultados são ordenados por ano (de forma ascendente) e receita (de forma descendente).
- **Categoria 4 (3 *queries*):** as *queries* têm aplicam restrições sobre todas as tabelas dimensão. Estas *queries* envolvem o cálculo do lucro ("L_REVENUE – L_SUPPLYCOST"), agrupado por ano e nação do cliente (*query 4.1*); ano, nação do fornecedor e categoria do produto (*query 4.2*); e ano, cidade do fornecedor e marca do produto (*query 4.3*).

4.2 Integração de MapReduce

Para integrar o modelo de programação MR num SDW convencional, importa considerar algumas questões, nomeadamente: o armazenamento de dados, as estruturas de acesso aos dados e as técnicas para acelerar o processamento de dados. Os SDW baseados em MR, mais propriamente em Hadoop, não controlam diretamente o armazenamento de dados nos discos que compõem o *cluster*. Estes utilizam o HDFS – o sistema de ficheiros distribuído – para armazenar os dados das tabelas. A forma como os dados se encontram organizados em HDFS afeta o desempenho destes DWs. Em (He et al., 2011), os autores identificaram quatro requisitos fundamentais para organizar os dados de um DW baseado através de MR, nomeadamente:

1. O carregamento rápido de dados - a redução do tempo de carregamento de dados é desejável, uma vez que durante a execução deste processo são utilizados recursos (ex. os tráfegos de rede e disco) e interferem com a execução normal das *queries*.
2. O processamento rápido de *queries* - para muitas *queries*, o tempo de resposta é crítico, no intuito de se satisfazer os requisitos (os pedidos em tempo real de *websites* e as cargas de trabalho pesadas originadas pelas *queries* de suporta à decisão submetidas por vários utilizadores em simultâneo).
3. A utilização eficiente do espaço de armazenamento - com o espaço em disco limitado, o armazenamento tem de ser bem gerido, de forma a maximizar o espaço disponível para acolher a quantidade crescente de dados gerados da atividade dos utilizadores.
4. Capacidade de adaptação a padrões de cargas de trabalho dinâmicos - grande parte das cargas de trabalho não seguem qualquer padrão (algumas análises de dados são processos de rotina que são executados periodicamente e de forma estática, ao passo que outras análises são executadas por *queries ad hoc*).

O RCFile (Secção 3.3.1.) foi desenhado para atender estes requisitos. Esta estrutura de dados – um particionamento horizontal, seguido de um particionamento vertical de um conjunto de dados – foi integrada com sucesso em plataformas para a análise de dados baseadas em MR. O Apache Hive, a implementação de um DW em Hadoop, assemelha-se a uma base de dados tradicional (Secção 3.3.3.). A base do Hive – o HDFS e o MR – tem algumas diferenças arquitetónicas que influenciam diretamente as características que este suporta, o que por sua vez afeta as aplicações

do Hive. As atualizações, transações e os índices servem de apoio às bases de dados tradicionais. Até há pouco tempo, estas características não têm sido consideradas como uma parte do Hive, uma vez que este foi construído para executar operações sobre os dados armazenados em HDFS utilizando o MR. Neste ambiente, a operação padrão são os *full-table scans* e as atualizações de tabelas são atingidas através da transformação dos dados para uma nova tabela.

O Hive não fornece atualizações, inserções e remoções ao nível do registo, nem transações (Capriolo et al., 2012). No entanto, existem casos em que é necessário recorrer a atualizações (ou inserções (*appends*)), ou a índices que produziria ganhos significativos no desempenho do sistema. Relativamente às transações, o Hive não define uma semântica clara para o acesso concorrente às tabelas. Isto significa que, as aplicações têm de construir o seu sistema de controlo de concorrência, a nível aplicacional, ou um mecanismo de *locks*. O RCFile permite apenas adicionar dados no final do ficheiro. Não permite escritas arbitrárias (He et al., 2011). O Apache Pig não é eficiente para as *queries* que se foquem num subconjunto de dados pequeno. Este tem de percorrer o conjunto de dados na íntegra. Já o HBase tem características de armazenamento diferentes para o HDFS. Este tem a capacidade de realizar atualizações a linhas e indexar atributos. As transações são efetuadas a nível de linha, ou seja, as atualizações – armazenadas em memória e periodicamente escritos em disco – a linhas são atómicas, não interessa o número de colunas que integram a transação. Desta forma, o mecanismo de *lock* mantem-se simples. Assim, o HBase poderá ser uma das soluções a seguir para a indexação e o armazenamento dos dados (White, 2012).

O cálculo das agregações produzidas pelas *queries* do SSB não requerem todos os registos da tabela de factos. Estas exigem somente um subconjunto de registos. Como se pode constatar pela Tabela 1, as *queries* do SSB têm uma percentagem de registos que satisfazem todos os predicados na tabela de factos, isto é, a seletividade – o *Filter Factor* (FF) – obtida pelas restrições nas tabelas dimensão e tabela de facto. Por exemplo, a *query 3.1* – a *query* menos seletiva – utiliza aproximadamente 0,034*6 000 000 = 20 400 registos, caso o SF seja igual a '1'. Neste caso em concreto (um conjunto de *queries* com uma ou mais condições de filtro), as estruturas, como os índices, devem ser exploradas para permitir o acesso eficiente aos dados e, assim, evitar o varrimento completo da tabela de factos (Chaudhuri et al., 2011).

Query	Restrições FF LINEORDER	FF dos predicados mais restritivos das tabelas dimensão				Combinado FF LINEORDER
		DATE	PART	SUPPLIER	CUSTOMER	
1.1	0,47*3/11	1/7				0,019
1.2	0,2*3/11	1/84				0,00065
1.3	0,1*3/11	1/364				0,000075
2.1			1/25	1/5		1/125 = 0,0080
2.2			1/125	1/5		1/625 = 0,0016
2.3			1/1000	1/5		1/5000 = 0,00020
3.1		6/7		1/5	1/5	6/175 = 0,034
3.2		6/7		1/25	1/25	6/4375 = 0,0014
3.3		6/7		1/125	1/125	6/109375 = 0,000055
3.4		1/84		1/125	1/125	1/1312500 = 0,00000076
4.1			2/5	1/5	1/5	2/125 = 0,016
4.2		2/7	2/5	1/5	1/5	4/875 = 0,0046
4.3		2/7	1/25	1/25	1/5	2/21875 = 0,000091

Tabela 1: A seletividade das *queries* do SSB (adaptado de (O'Neil et al., 2007))

O teste proposto pelo SSB consiste no teste de várias cargas de trabalho - *"short scans"*. Em (Cooper et al., 2010), os autores realizam um *benchmark* que testa cargas de trabalhos diferentes para vários sistemas de armazenamento e gestão de dados na *cloud*. Para a carga de trabalhos que o SSB apresenta, o *benchmark* demonstra que o HBase tem um bom desempenho, comparado com os sistemas Apache Cassandra (Lakshman and Malik, 2009), também baseado no Bigtable, e Yahoo!'s PNUTS (Cooper et al., 2008), para as *range scans* que obtêm mais de 100 registos em média. O HBase armazena os dados em disco de forma mais compacta, o que melhora o

desempenho para grandes *ranges* ou intervalos. Assim, optou-se por utilizar o HBase para armazenar a tabela de factos "LINEORDER". No entanto, o mesmo estudo revela que o HBase não é eficiente quando as cargas de trabalhos são leituras intensivas. Nestes trabalhos, as bases de dados têm latências inferiores ao HBase. Num processo de leitura, o HBase tem de reconstruir os fragmentos dos registos de múltiplas páginas. A latência na leitura é grande, devido à sua implementação do armazenamento *log-structured*. O HBase descarrega as tabelas que se encontram em memória para o disco em ficheiros separados. Assim, o HBase tem de percorrer os ficheiros à procura os fragmentos que formam um registo.

O modelo de dados do HBase (Secção 2.4.2.) tem duas estruturas fundamentais de indexação (Lars, 2011): a chave linha e a chave coluna. Estas estruturas oferecem uma noção, quer dos dados que armazenam, quer da ordenação explorada. Para modelar a tabela de factos "LINEORDER" em HBase, primeiro há que considerar o esquema lógico da tabela. A maior unidade de divisão de uma tabela é a coluna família (estas colunas têm um sentido diferente das colunas encontradas nas bases de dados orientadas à coluna). A Figura 29 representa uma tabela em HBase nos níveis lógico e físico.

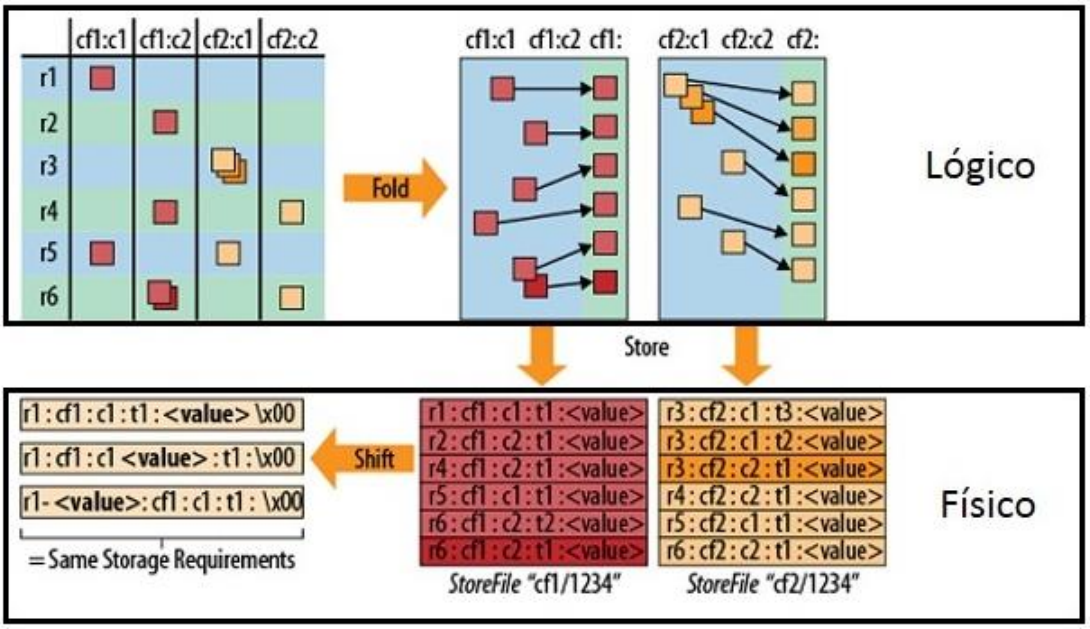


Figura 29: Armazenamento de linhas em HBase (adaptado de (Lars, 2011))

O nível lógico mostra o esquema lógico dos dados (as linhas e as colunas). As colunas representam a união dos nomes das colunas família e *qualifier*, isto é, a chave coluna em HBase. As linhas têm uma chave para ter a possibilidade de endereçar todas as colunas numa linha lógica. Esta tabela é dobrada (*Fold*), antecipando, assim, o armazenamento físico da tabela. Para cada linha, as células são armazenadas contiguamente em ficheiros separados (um por cada coluna família). As diferentes versões de uma célula são também armazenadas consecutivamente, em separado, por ordem decrescente do *timestamp*. O HBase não armazena valores nulos, ou seja células sem um valor definido. O ficheiro contém os dados que foram explicitamente definidos.

A Figura 30 mostra uma célula completa (KeyValue, na terminologia do HBase), com a informação complementar referente à estrutura, ordenada por chave linha primeiro, depois por chave coluna, e os efeitos do aproveitamento dos campos KeyValue. A API do HBase permite efetuar consultas aos dados armazenados, com uma granularidade decrescente da esquerda para a direita. A seleção de linhas, através de uma chave linha específica ou um intervalo de chaves linha, permite reduzir efetivamente a quantidade de dados a analisar numa consulta. Ao especificar uma coluna família numa *query*, exclui-se a necessidade de analisar os ficheiros separados, correspondentes às colunas famílias não especificadas. O *timestamp* é também um critério de seleção, que evita a análise de dados para um determinado intervalo de tempo. O próximo nível de filtro para uma *query* é a coluna *qualifier*. Neste nível, bem como para o campo Value, é necessário analisar célula a célula para determinar se satisfazem as condições da *query*. Visto que a eficiência dos critérios de seleção diminui bastante da esquerda para a direita, parte do valor pode deslocar-se para uma posição mais significativa (ex. a chave linha) – o *Shift* na Figura 29 – sem alterar a quantidade de dados armazenados. Esta alteração da disposição dos campos transforma cada valor numa linha lógica separada. Desta forma, estas chaves linha – compostas – permitem aceder aos dados com uma granularidade mais baixa.

A API do HBase permite também efetuar o *scan* parcial às chaves linha. Para isso, é necessário especificar uma chave linha inicial (inclusive) e uma chave linha de paragem (exclusive). Os *scans* parciais têm a possibilidade de iterar sobre o conjunto de chaves linha e extrair os valores pretendidos.

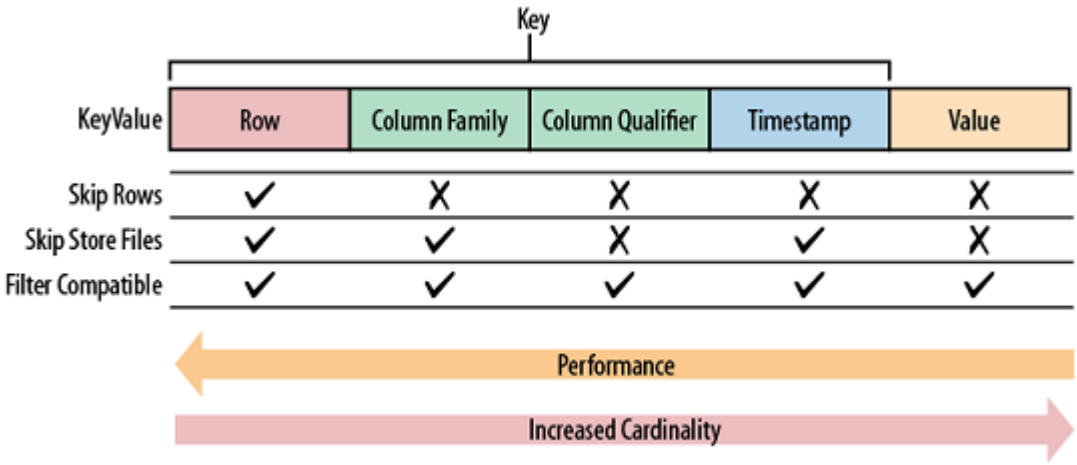


Figura 30: Desempenho de uma consulta em HBase (retirado de (Lars, 2011))

Considerando o modelo de dados do HBase, a tabela de factos "LINEORDER" adota o seguinte esquema:

- A chave linha combina os atributos, com o maior nível de granularidade, das tabelas dimensão.
- As colunas família separam as medidas em 2 conjuntos.
- As colunas *qualifier* armazenam os factos com a granularidade da tabela de factos.
- Os valores representam as medidas.

Para a mesma chave, o HBase tem a capacidade de suportar versões diferentes das medidas, porém esta propriedade (*timestamp*) não é considerada para a consulta, já que um facto tem apenas uma versão, isto é, um facto é unívoco e não volátil. Assim, a chave resume-se a <Chave linha, Chave coluna>.

Chave linha

O desenho da chave linha tem em consideração o conjunto de *queries* do teste SSB. Nestas, as dimensões em análise são "CUSTKEY", "PARTKEY", "SUPPKEY" e "ORDERDATE". A dimensão "COMMITDATE" não íntegra a chave linha, visto que não participa nas *queries* do teste. Para as dimensões que participam, os atributos das hierarquias, com o maior nível de granularidade ("C_REGION", "P_MFGR", "S_REGION" e "D_YEAR"), são associados. A ordem destes atributos é um aspeto fundamental, visto que define a disposição dos factos em disco. Por exemplo, a chave

<"D_YEAR" + "P_MFGR"> agrupa primeiro os factos para o atributo "D_YEAR", em seguida, associa estes factos aos valores de "P_MFGR" (Figura 31). Esta chave é ordenada lexicograficamente e, assim, reúne todas as chaves linha com o mesmo prefixo.

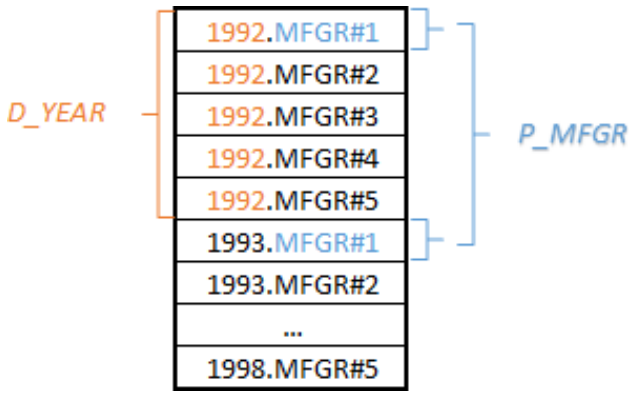


Figura 31: Parte da chave <"D_YEAR"."P_MFGR">

O número de ocorrências dos atributos dimensão, nas *queries*, definiu o arranjo da chave linha. Dos atributos selecionados para compor a chave linha, o mais frequente é o "D_YEAR" que pertence à dimensão "ORDERDATE" – representa a dimensão tempo no SSB (importante em SSDs), uma vez que revela as tendências do esquema (Chaudhuri and Dayal, 1997). Deste modo, o atributo "D_YEAR" é o primeiro elemento da chave linha. Os factos são agrupados por ano, o que permite uma análise rápida das vendas de um determinado ano ou de um intervalo de anos. A restante ordem dos elementos é: "S_REGION", seguido de "C_REGION" e no final "P_MFGR". O mesmo critério foi aplicado para determinar a posição destes atributos, ou seja, o número de participações nas *queries* do teste. O cálculo deste número teve em consideração o número de ocorrências que estes atributos ou os atributos que integram uma mesma hierarquia (ex. os atributos "S_REGION" e "S_NATION" fazem parte da mesma hierarquia e representam a dimensão "L_SUPPKEY"). No final, a chave linha assumiu a seguinte forma:

$$< D\_YEAR >.< S\_REGION >.< C\_REGION >.< P\_MFGR >$$

Chave coluna

A chave coluna é multidimensional, o que significa que é composta por vários elementos: coluna família e coluna *qualifier*. As colunas família separam as medidas em 2 grupos, distribuindo as medidas por 2 ficheiros:

- 1º Grupo: QUANTITY, EXTENDEDPRICE, DISCOUNT, REVENUE e SUPPLYCOST;
- 2º Grupo: ORDERPRIORITY, SHIPPRIORITY, ORDTOTALPRICE, TAX e SHIPMODE;

Esta abordagem permite explorar o particionamento vertical da tabela de factos, tentando simular um armazenamento orientado à coluna (Abadi, 2008). Porém, uma partição para cada atributo requer um mecanismo de reconstrução de registos. Para evitar estas reconstruções, optou-se por organizar todos os atributos de uma tabela em grupos de atributos diferentes (Stonebraker et al., 2005). O primeiro grupo ficou com as medidas utilizadas no conjunto de *queries* do teste SSB, enquanto que o segundo grupo com as medidas excluídas das análises. Assim, para satisfazer as *queries* do teste SSB basta analisar um ficheiro, ou seja, processar metade das medidas. Isto reduz o acesso ao disco durante a execução de uma *query*, caso o cálculo abranja medidas de apenas um grupo. Caso contrário é necessário processar os dois ficheiros. As colunas *qualifier* representam o nível mais detalhado deste modelo, agrupando e combinando as dimensões ("ORDERDATE", "SUPPKEY", "PARTKEY", "CUSTKEY" e "COMMITDATE"), com as chaves primárias da tabela de factos ("ORDERKEY" e "LINENUMBER"). Esta composição dos atributos representa o grão da tabela de factos.

A Tabela de factos "LINEORDER"

A Figura 32 mostra a representação dos dados em HBase para o facto com a chave primária "ORDERKEY" = '4373889' e "LINENUMBER" = '5'.

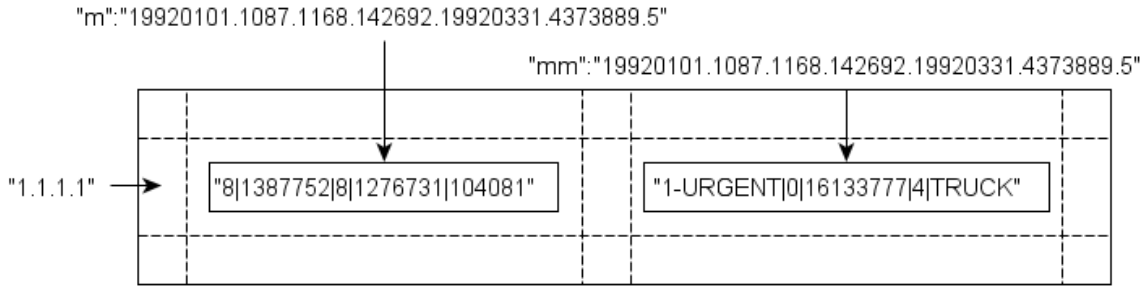


Figura 32: Representação dos dados de um facto

Os atributos que compõem a chave linha empregam chaves de substituição, que se encontram armazenadas em tabelas HBase. Para cada uma das hierarquias do SSB (Secção 4.1.) gera-se uma

chave de substituição para os atributos com maior nível de granularidade. Os restantes atributos da hierarquia partem da chave de substituição para criar uma árvore, através da técnica *materialized path expressions* (Abadi et al., 2007), armazenando, através de uma expressão, o caminho completo até à raiz (ex. o caminho absoluto de um ficheiro num sistema de ficheiros). A Figura 33 apresenta a representação dos dados de uma hierarquia, estruturada em pares chave/valor.

D_YEAR

"1992"	"1"
"1993"	"2"
"1994"	"3"
"1995"	"4"
…	…
"1998"	"7"

D_YEARMONTHNUM|D_YEARMONTH

"199201|Jan1992"	"1.1"
"199202|Feb1992"	"1.2"
"199203|Mar1992"	"1.3"
…	…
"199811|Nov1998"	"7.11"
"199812|Dec1998"	"7.12"

Figura 33: Parte da hierarquia da tabela dimensão "Date"

A estrutura da esquerda indexa o atributo "D_YEAR". Esta armazena os valores distintos deste atributo e as respetivas chaves de substituição gerada. A estrutura da direita armazena os valores de dois atributos ("D_YEARMONTHNUM" e "D_YEARMONTH") distintos que representam o mesmo nível de detalhe e o caminho. Esta estrutura coloca à disposição a chave de substituição para as consultas à tabela de factos (ex. os valores '199811' e 'NOV1998' pertencem ao ano de '1998', e assim a chave de substituição é igual '7'). As restantes hierarquias são indexadas da mesma forma. A hierarquia "CITY -> NATION -> REGION" pode ser indexada através de duas tabelas dimensão ("CUSTOMER" e "SUPPLIER"). Neste caso, optou-se por utilizar a tabela dimensão "SUPPLIER" para o povoamento da estrutura, porque tem uma cardinalidade inferior à tabela dimensão "CUSTOMER".

Tabelas dimensão

As tabelas dimensão encontram-se armazenadas num SGBD relacional, essencialmente devido a dois motivos:

1. O HBase não suporta junções, pelo menos da mesma forma do que os SGBDs relacionais (White, 2012).

2. Apesar do teste SSB não propor dimensões de variação lenta, estas devem ser tidas em consideração.

O modelo de dados em causa apenas suporta quatro operações: *Get*, *Put*, *Scan* e *Delete*. As junções têm de ser implementadas ao nível aplicacional - na Secção 3.3.2. são discutidas várias implementações de algoritmos de junção em ambientes MR, entre os quais o *star join*. Em (Afrati and Ullman, 2010), os autores sugerem que as tabelas dimensão devem ser replicadas por todas as máquinas do *cluster*, já que este cenário pode ter um custo bem menor do que o custo da transmissão do resultado das junções intermédias – junção realizada entre uma tabela de factos e cada uma das tabelas dimensão – pela rede. Como as tabelas dimensão se encontram armazenadas nos discos locais, é possível explorar outros sistemas de armazenamento para além do HDFS ou mesmo do HBase. Assim, optou-se por utilizar um SGBD relacional para armazenar estas tabelas. Estes sistemas são adequados, tanto em termos das cargas de trabalhos realizadas nas tabelas dimensão como no acomodar das atualizações dos atributos que variam ao longo do tempo (Kimball, 1996). Em síntese, a arquitetura definida para o DW pode ser observada na Figura 34:

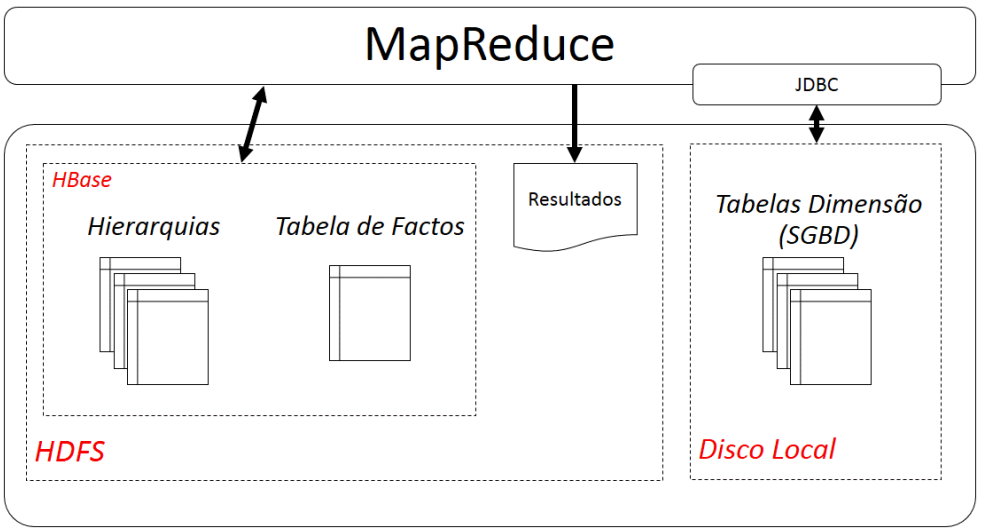


Figura 34: Arquitetura do DW

A tabela de factos "LINEORDER" encontra-se armazenada em HBase e particionada por uma chave linha composta pelos atributos das tabelas dimensão. Estas tabelas, por sua vez, estão

armazenadas num SGBD relacional. As hierarquias, armazenadas em HBase, auxiliam a determinar as chaves linha que se têm que consultar para o cálculo das agregações. Os resultados produzidos pelas análises são armazenados em HDFS no formato CSV. Nesta arquitetura coexistem um SGBD relacional e um sistema MR + HBase.

4.2.1 O Processo de Povoamento

O *paper* original do MR aborda a indexação (Secção 2.1.1.), descrevendo aí a construção de um índice invertido. A estratégia para indexar uma tabela de factos não difere muito da abordagem utilizada num índice invertido. Mas esta não implementa a fase *reduce*. Aí, a estrutura de dados é construída em três passos:

1. Povoar as quatro tabelas dimensão na base de dados relacional.
2. Povoar as 3 estruturas de dados hierarquia.
3. Povoar a tabela de factos.

Após o gerador SSB produzir os dados que compõem o DW, estes são armazenados em 5 ficheiros TBL (4 tabelas dimensão e uma tabela de factos). Os dados das tabelas dimensão são carregados diretamente a partir dos ficheiros gerados. Quanto à tabela de factos "LINEORDER", o ficheiro TBL é copiado para o HDFS para, posteriormente, ser processado por um trabalho MR. Porém, antes de povoar a tabela de factos, é necessário povoar as três estruturas de dados da hierarquia. O povoamento das estruturas de dados hierarquia envolve oito tabelas em HBase, nomeadamente:

- **"year"** e **"month":** originária da tabela dimensão "DATE";
- **"region"**, **"nation"** e **"city":** originárias da tabela dimensão "SUPPLIER";
- **"mfgr"**, **"category"** e **"brand":** originárias da tabela dimensão "PART".

Este povoamento ocorre imediatamente antes – numa fase de configuração do trabalho MR – do povoamento da tabela de factos "LINEORDER" em HBase, ou melhor, ambos os povoamentos são executados pelo mesmo trabalho MR. A Figura 35 ilustra o povoamento da hierarquia originada da tabela dimensão "PART". Este processo começa pela seleção dos atributos "P_MFGR", "P_CATEGORY" e "P_BRAND1" da tabela dimensão "PART". Para cada registo obtido é gerada uma

materialized path expression (Figura 33) que é armazenada numa estrutura HashMap<"Chave", "Path">. Em seguida, esta estrutura é separada em 3 três listas ("mfgrs", "cats" e "brands") que povoarão as tabelas hierarquia "mfgr", "category" e "brand", respetivamente. De realçar que "Path" representa uma estrutura de dados que armazena:

- uma chave e o seu caminho correspondente para o atributo com o menor nível de detalhe (a raiz da árvore);
- e uma subárvore – outra estrutura "Path" – que armazena os restantes atributos com um nível de detalhe superior à raiz.

Após o povoamento das tabelas hierarquia, segue-se o povoamento da tabela de factos, já com as chaves de substituição, através do *bulkimport* do HBase (Lars, 2011). Esta estratégia utiliza um trabalho MR (apenas a fase *map*), que produz um grupo de ficheiros (HFILES) contendo os dados preparados, para, posteriormente, importar da tabela HBase (Figura 36).

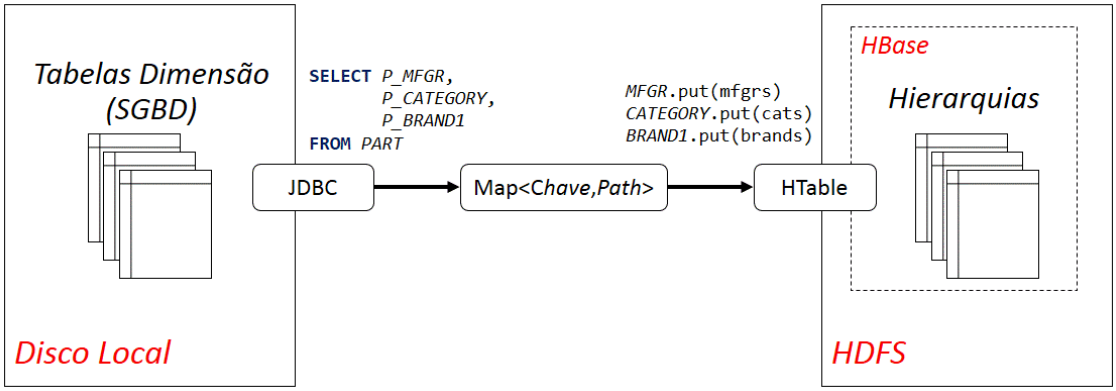


Figura 35: Representação do povoamento das hierarquias

Antes de iniciar o *scan* à tabela de factos (o ficheiro TBL gerado pelo teste SSB), o povoamento começa com a construção de quatro estruturas *HashMap*[20], que armazenam as chaves primárias de todas as tabelas dimensão e as chaves de substituição armazenadas nas estruturas de dados hierarquia "year", "region" (tabelas dimensão "SUPPLIER" e "CUSTOMER") e "mfgr". Esta relação,

[20] http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html

a chave primária da tabela dimensão (oriunda do SGBD) e a chave de substituição (oriunda da estrutura de dados hierarquia), é estabelecida através dos valores dos atributos "D_YEAR", "S_REGION", "C_REGION" e "P_MFGR" das tabelas dimensão "DATE", "SUPPLIER", "CUSTOMER" e "PART", respetivamente. Este processo é executado pela função *setup* (função executada, uma vez, no início de cada *mapper*).

A função *map* lê cada linha do ficheiro TBL correspondente à tabela de factos e que está armazenada temporariamente em HDFS. No final do carregamento, este ficheiro é removido do HDFS. Para cada registo da tabela de factos são extraídos os atributos "L_ORDERDATE", "L_SUPPKEY", "L_CUSTKEY" e "L_PARTKEY". Estes atributos são essenciais, uma vez que determinam a chave-linha a que pertence. A chave-linha é calculada a partir das quatro estruturas *HashMap*, produzidas pela função *setup*, isto é, as chaves de substituição – armazenadas nas *HashMap* – que compõem a chave linha e que são adquiridas a partir dos valores dos atributos extraídos. Os restantes atributos são divididos para formar as chaves colunas e os grupos de medidas (Figura 32). No final, o *mapper* emite os dados no formato interno de armazenamento do HBase para que estes sejam carregados na tabela eficientemente.

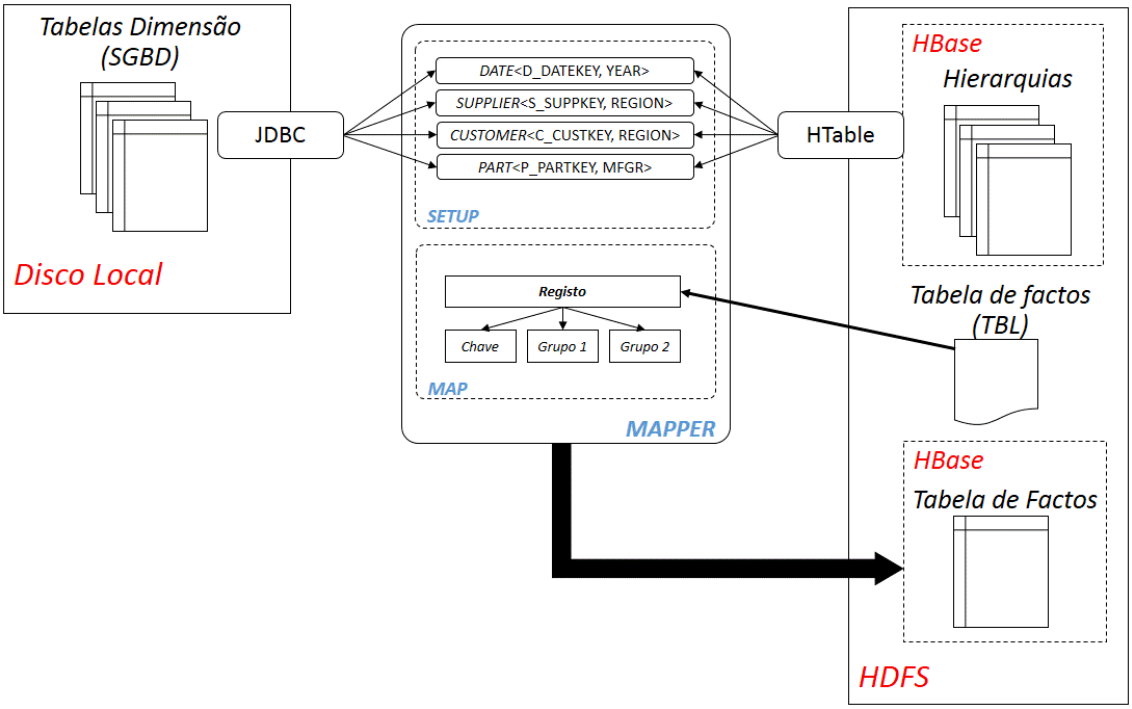


Figura 36: Ilustração do povoamento da tabela de factos

Atualizações

Depois do primeiro povoamento, um DW é regularmente atualizado (Golfarelli and Rizzi, 2009). O teste SSB não especifica dimensões de variação lenta. As tabelas dimensão, encontrando-se armazenadas num SGBD relacional, acomodam os atributos suscetíveis a estas mudanças através das técnicas especificadas anteriormente (Secção 3.1.1.). Para as tabelas dimensão, optou-se por utilizar o tipo 4 (Kimball, 1996). No entanto, a alteração de um atributo que compõe uma ou mais hierarquias, indexadas nas tabelas HBase, requer uma manutenção diferente. As modificações que advêm dessa situação implicam a adição de um novo par chave/valor, no qual a chave corresponde ao novo valor do atributo, correspondendo o valor à *materialized path expression* do valor antigo.

4.2.2 O Processamento de *Queries*

A construção de um índice <Chave linha, Chave coluna> para a tabela de factos "LINEORDER" tem como objetivo reduzir o tempo de processamento das *queries*. Com este tipo de estrutura, o processamento de *queries* fica restrito aos registos da tabela de factos que contém, pelo menos, um critério de seleção da *query*. Assim, o processamento de uma *query* pode explorar os *range scans* que o HBase suporta eficientemente. As *queries* têm também a faculdade de cruzar os dados da tabela de factos, de forma expedita, caso os atributos de seleção se encontrem incluídos no índice (ex. a consulta que seleciona os registos da tabela de fatos para o ano de '1993' e os fornecedores da região 'EUROPE').

O HBase suporta duas operações de pesquisa: o *Get* e o *Scan*. Dada uma chave linha, a operação *Get* devolve o seu valor. Esta operação é ideal para sondar as estruturas tipo hierarquia, uma vez que as consultas sobre estas estruturas resumem-se à aquisição da chave de substituição para um dado valor de um atributo (isto é, uma chave linha). As tabelas hierarquia são mantidas em memória para que a operação seja executada mais rapidamente. Já a operação *Scan* é utilizada para iterar sobre intervalos de chaves linha que indexam a tabela de factos. Esta operação tem a capacidade de adquirir vários valores (subconjuntos da tabela de factos) para as chaves linha e chaves coluna pré-determinadas.

De seguida, apresenta-se a adaptação do conjunto de *queries* do teste SSB para o modelo de programação MR.

Queries da primeira categoria

A primeira *query* do SSB (Apêndice B) executa uma junção, entre as tabelas dimensão "DATE" e a tabela de factos "LINEORDER", e calcula um somatório – a receita ("EXTENDEDPRICE*DISCOUNT") – através da seleção dos registos da tabela de factos com base em três parâmetros:

- "D_YEAR" = '1993'.
- Os valores da medida "DISCOUNT" entre '1' e '3'.
- Os valores da medida "QUANTITY" inferiores a '25'.

Esta *query*, para além de filtrar os registos através de uma junção, exclui registos através do valor das medidas "DISCOUNT" e "QUANTITY". O processo de adaptação para MR começa por determinar as chaves linha e coluna que armazenam o conjunto de registos pretendidos para o cálculo da *query*. Relativamente à chave coluna, adiciona-se o Grupo 1 de medidas para o *Scan*. O segundo grupo não é utilizado neste conjunto de *queries* teste, visto que as medidas que reúne não fazem parte das análises. Quanto à chave linha, o atributo "D_YEAR" pertence à seleção desta *query*. Dessa forma, determina-se a sua chave de substituição – o valor '2' – para filtrar as chaves linha a consultar (Figura 37). Definido o conjunto de chaves linha (neste caso as linhas iniciadas por '2', uma vez que o atributo "D_YEAR" ocupa a primeira posição da chave linha composta), termina a fase de configuração do trabalho MR (exceto para as configurações habituais).

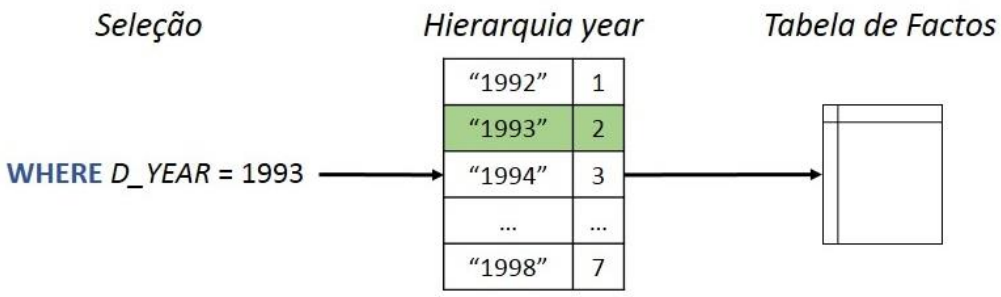


Figura 37: Obtenção da chave de substituição para o ano de '1993'

O volume do conjunto de chaves linha é calculado pela cardinalidade dos atributos que o compõem (Tabela 2). No total, o conjunto tem 875 linhas, valor este que resulta da multiplicação das cardinalidades destes atributos. Para a primeira *query* do SSB, o valor de "D_YEAR" é fixado. Desta forma, o número de chaves linha a consultar é reduzido para 125 (1*5*5*5), que abrange as linhas entre '2.1.1.1' e '2.5.5.5', inclusive.

Atributo	Cardinalidade
D_YEAR	7
S_REGION	5
C_REGION	5
P_MFGR	5

Tabela 2: Cardinalidade dos atributos chave linha

O *mapper* processa os pares chave/valor de *input*, extraídos do HBase. Este *mapper* é composto por uma função *map* que, para cada KeyValue, separa os campos, de forma a obter o valor das medidas "L_EXTENDEDPRICE", "L_DISCOUNT" e "L_QUANTITY", necessários para o cálculo da *query*. Depois, estes valores são testados, de acordo com as restantes cláusulas WHERE, para verificar se o registo satisfaz os critérios de seleção. No final, o *mapper* emite um conjunto de pares chave/valor, com a chave 'rev' e o valor "L_EXTENDEDPRICE*L_DISCOUNT". O *reducer* é implementado pela classe *LongSumReducer*[21] e calcula o somatório dos valores produzido pelo *mapper*. O trabalho MR para a primeira *query* do SSB está ilustrado na Figura 38.

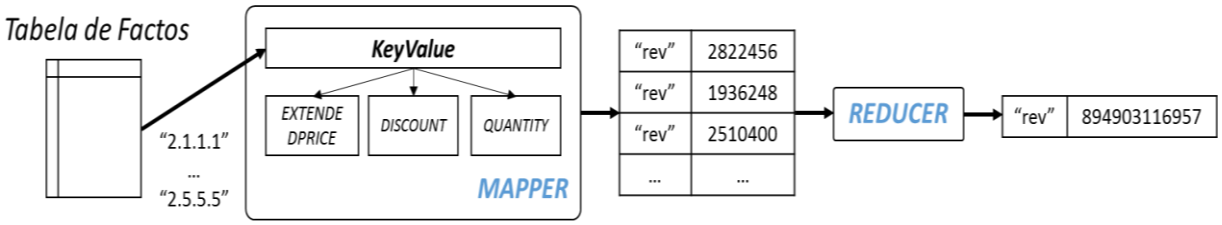


Figura 38: Trabalho MR que expressa a primeira *query* do SSB

[21] http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/lib/reduce/LongSumReducer.html

Esta *query* é a mais simples de adaptar para um trabalho MR. A sua implementação evita a junção com a tabela dimensão "DATE", uma vez que o conjunto de chaves linha devolve todos os registos correspondentes ao ano de '1993'. A segunda *query*, da mesma categoria que a primeira, implementa uma junção simples. A estrutura das *queries* são semelhantes, no entanto esta seleciona um atributo que não faz parte da chave linha, o "D_YEARMONTHNUM". Este atributo encontra-se indexado na estrutura hierarquia, criada para a tabela dimensão "DATE", e permite o cálculo da chave de substituição, da mesma forma que a *query 1.1*. Ainda assim, tem-se que determinar quais os registos que satisfazem as novas restrições, nomeadamente a "D_YEARMONTHNUM = '1994'". O trabalho MR para a segunda *query* é idêntico ao apresentado na Figura 38, porém o *mapper* é diferente (Figura 39).

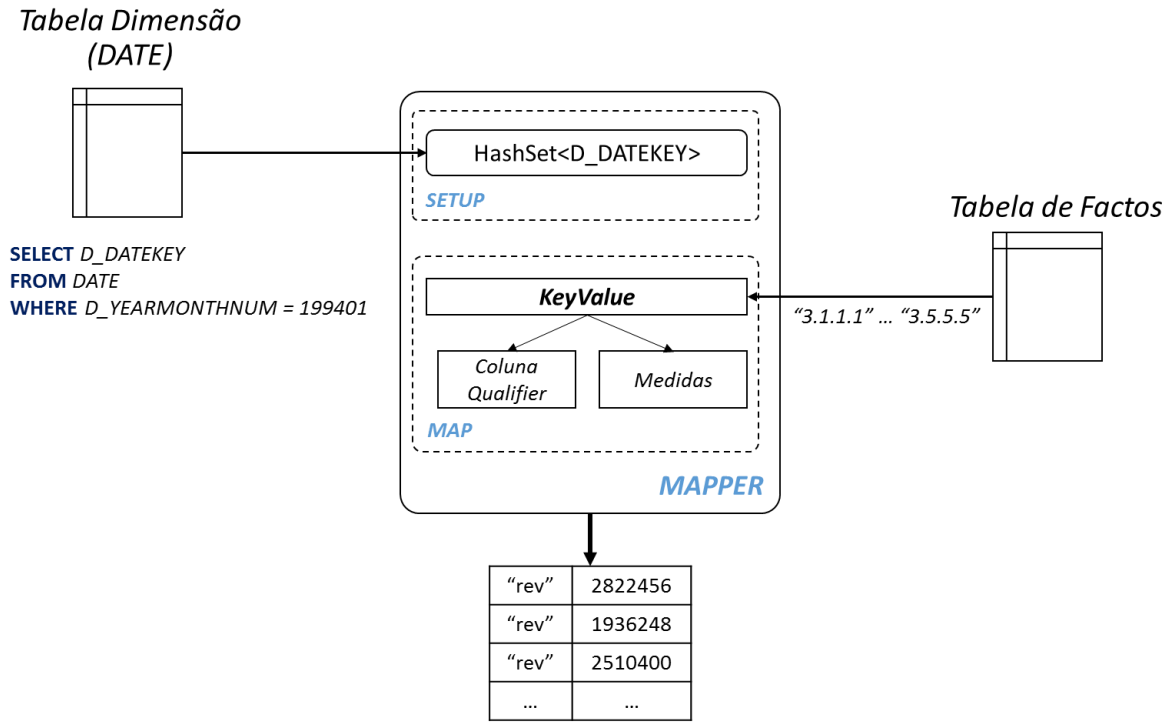


Figura 39: O *mapper* da segunda *query*

Este *mapper* implementa um algoritmo de junção semelhante ao *broadcast join* (Blanas et al., 2010). O algoritmo implementa junções entre conjuntos de dados, caso um destes caiba em memória (Secção 3.3.2). O conjunto de dados de menor dimensão – a tabela dimensão "DATE" – é carregado do disco local para a memória pela função *setup* - esta estratégia também é aplicada no

povoamento da tabela de factos. A Figura 40 apresenta a função *setup* para a *query 1.2*. Esta função é executada uma vez, antes de cada função *map*, e lê a tabela dimensão "DATE" do SGBD, para recolher as chaves primárias que satisfazem restrição "D_YEARMONTHNUM = 199401". As chaves primárias são armazenadas numa estrutura de dados "datas" (linha 6).

```
01 public void setup(Context context)
    throws IOException, InterruptedException {

02   try {
03     Connection con = DriverManager.getConnection("…");
04     PreparedStatement pst = con.prepareStatement("
          SELECT D_DATEKEY
          FROM dat
          WHERE d_yearmonthnum = 199401");
05     ResultSet rs = pst.executeQuery();
06     while (rs.next()) {datas.add(rs.getInt(1));}
07     pst.close(); rs.close(); con.close();
08   } catch (SQLException e) {System.out.println(e.getMessage());}
09 }
```

Figura 40: A função *setup* para a *query 1.2*

Terminada a função *setup*, a função *map* realiza a junção entre as chaves primárias recolhidas para a memória (*HashSet*[22]) e os KeyValue, extraídos do HBase (Figura 41).

```
01 public void map(ImmutableBytesWritable row,
                   Result columns,
                   Context context)
    throws IOException, InterruptedException {

02    for (KeyValue kv : columns.list()) {
03        String[] q = Bytes.toStringBinary(kv.getQualifier()).split("\\.");
04        int dataKey = Integer.parseInt(q[0]);
05        if(datas.contains(dataKey)) {
06            String[] m = Bytes.toStringBinary(kv.getValue()).split("\\|");
07            int disc = Integer.parseInt(m[2]);
08            int qunt = Integer.parseInt(m[0]);
09            int extp = Integer.parseInt(m[1]);
10            if((disc >= 4) && (disc <= 6))
11                if((qunt >= 26) && (qunt <= 35))
12                    context.write(one, new LongWritable(extp*disc));
13          }
14      }
15 }
```

Figura 41: A função *map* para a *query 1.2*

[22] http://docs.oracle.com/javase/6/docs/api/java/util/HashSet.html

Para cada KeyValue, a função *map* separa os campos, de forma a obter a coluna *qualifier* (linhas 3 e 4) – mais propriamente o valor de "L_ORDERDATE" – e o "Value" ou as medidas (linha 6). Antes de extrair as medidas "L_EXTENDEDPRICE", "L_DISCOUNT" e "L_QUANTITY" (linhas 7 a 9), o registo da tabela de factos é testado. Caso a estrutura de dados "datas" contenha o valor de "L_ORDERDATE" (linha 5), então o registo satisfaz o critério de junção entre a tabela de factos e a tabela dimensão "DATE". Caso contrario, o registo é rejeitado. Os valores das medidas extraídas são testados de acordo com as restantes cláusulas WHERE, para verificar se o registo continua a satisfazer os critérios de seleção (linhas 10 e 11). No final, o *mapper* emite um conjunto de pares chave/valor, com a chave 'rev' – representada pela variável textual "one" – e o valor "L_EXTENDEDPRICE*L_DISCOUNT" (linha 12). O *mapper* da *query 1.3* é semelhante ao apresentado pela Figura 39, no entanto, mudam-se as cláusulas de seleção.

***Queries* das restantes categorias**

A primeira categoria está representada apenas por *queries* com junções simples ou *Two-way join* (Blanas et al., 2010). As *queries* das categorias seguintes propõem novos desafios, como a primeira *query* da segunda categoria – a *query 2.1* –, que agrupa e ordena, de forma ascendente, os diferentes valores dos atributos "D_YEAR" e "P_BRAND1", para o somatório dos valores da medida "L_REVENUE". Nesta *query* participam três tabelas dimensão: "DATE", "PART" e "SUPPLIER". Através da chave linha, os registos da tabela de factos são filtrados por dois atributos: "S_REGION = 'AMERICA'" e "P_MFGR = 'MFGR#1'", obtido a partir de "P_CATEGORY = 'MFGR#12'". Assim, o número de linhas a consultar é de 35 (7*1*5*1), uma vez que são fixados dois atributos da chave linha.

Do ponto de vista estrutural, o *mapper* para a *query 2.1* é semelhante ao apresentado pela Figura 39. Com a projeção dos atributos "D_YEAR" e "P_BRAND1", a função *setup* passa a construir estruturas de dados pares chave/valor, isto é, estruturas Map[23]. A Figura 42 apresenta a função *setup* que povoa duas estruturas de dados (*HashMap*), armazenando os valores dos atributos projetados, com as respetivas chaves para a junção. A estrutura de dados "datas" armazena o atributo "D_YEAR", com a chave de substituição e o ano da estrutura de dados hierarquia, em

[23] http://docs.oracle.com/javase/6/docs/api/java/util/Map.html

HBase (linha 6). No fundo, a estrutura "datas" armazena uma cópia da estrutura de dados hierarquia invertida, isto é, a estrutura <valor, chave>. Este povoamento é excecional, visto que o atributo "D_YEAR" faz parte da chave linha. Da mesma forma, ler a tabela dimensão "DATE" do SGBD povoava a "datas", porém a abordagem implementada é a mais eficiente, porque processa menos informação. O atributo "P_BRAND1" é indexado em "parts" pela tabela dimensão "PART" (linha 16), com a chave "P_PARTKEY", para os registos que satisfazem restrição "P_CATEGORY = 'MFGR#12'".

```
01 public void setup(Context context)
    throws IOException, InterruptedException {

02    Configuration config = context.getConfiguration();
03    HTable ano = new HTable(config, "dy");
04    ResultScanner scanner = ano.getScanner(new Scan());
05    for (Result rr = scanner.next(); rr != null; rr = scanner.next()) {
06      datas.put(Bytes.toString(rr.getValue(Bytes.toBytes("k"), Bytes.toBytes("p"))),
                  Bytes.toString(rr.getRow()));
07    }
08    scanner.close(); ano.close();

09    try {
10      Connection con = DriverManager.getConnection("...");
11      PreparedStatement pst = con.prepareStatement("
12        SELECT P_PARTKEY, p_brand1
13        FROM part
14        WHERE p_category = 'MFGR#12'");
15      ResultSet rs = pst.executeQuery();
16      while (rs.next()) {parts.put(rs.getInt(1), rs.getString(2));}
17      pst.close(); rs.close(); con.close();
18    } catch (SQLException e) {System.out.println(e.getMessage());}
19 }
```

Figura 42: A função *setup* para a *query 2.1*

A função *map* da *query 2.1* (Figura 43) é semelhante à *query 1.2*, no entanto a *query 2.1* integra operações de projeção, de agrupamento e de ordenação. Para cada KeyValue (já filtrado por "S_REGION" e "P_MFGR"), a função *map* filtra os registos da tabela de factos – através da operação de junção (linha 5), emitindo um conjunto de pares chave/valor (linha 9). A chave combina os atributos projetados pela *query* – o "D_YEAR" e o "P_BRAND1" – pela ordem de agrupamento e ordenação. Os valores destes atributos são obtidos a partir das estruturas de dados construídas pela função *setup*. O valor dos pares emitidos corresponde à medida "L_REVENUE". O conjunto de pares <"D_YEAR,P_BRAND1", "L_REVENUE"> é posteriormente agregado pelo *reducer*, que também é implementado pela classe *LongSumReducer*.

```
01 public void map(ImmutableBytesWritable row,
                   Result columns,
                   Context context)
     throws IOException, InterruptedException {

02    for (KeyValue kv : columns.list()) {
03      String[] q = Bytes.toStringBinary(kv.getQualifier()).split("\\.");
04      int partKey = Integer.parseInt(q[3]);
05      if(parts.containsKey(partKey)) {
06        String[] m = Bytes.toStringBinary(kv.getValue()).split("\\|");
07        String dataKey = Bytes.toStringBinary(kv.getRow()).split("\\.")[0];
08        int rev = Integer.parseInt(m[3]);
09        context.write(new Text(datas.get(dataKey)+","+parts.get(partKey)),
                        new LongWritable(rev));
10      }
11    }
12 }
```

Figura 43: A função *map* para a *query 2.1*

Agrupamento e ordenação

Os pares produzidos pela função *map* são agrupados e ordenados pelo próprio MR (Dean and Ghemawat, 2004), tal como a Figura 44 exemplifica. A *query 2.1* agrupa e ordena os atributos projetados "D_YEAR" e "P_BRAND1", respetivamente. Esta disposição do resultado final é alcançada pelo MR, no momento da emissão da chave da função *map*. Esta função emite uma chave <"D_YEAR", "P_BRAND1">, que impõe a ordenação do conjunto. Primeiro por "D_YEAR" e depois por "P_BRAND1". O *reducer* calcula os valores agregados, para cada chave e escreve o resultado num ficheiro com as chaves respetivas, ordenadas lexicograficamente.

"1992,MFGR#121"	2032150
"1992,MFGR#1210"	4184467
"1992,MFGR#121"	1197864
...	...

→

"1992,MFGR#121"	2032150,1197864
"1992,MFGR#1210"	4184467
...	...

Figura 44: Agrupar parte dos valores pelas chaves ("D_YEAR" e "P_BRAND1") ordenadas

A cláusula GROUP BY do SQL agrupa os atributos pela ordem definida, enquanto que a cláusula ORDER BY ordena os valores de atributos. Para as categorias 2 e 4, as *queries* implementam a abordagem descrita acima, uma vez que os atributos são agrupados e ordenados da mesma forma. Porém, as *queries* da categoria 3 requerem uma nova abordagem, uma vez que o ORDER BY inclui o resultado do somatório ordenado decrescentemente. Este resultado é obtido após a execução da função *reduce*.

O algoritmo de execução, para *queries* da terceira categoria, implementa dois trabalhos MR. O primeiro implementa um algoritmo semelhante ao descrito para a *query 2.1*. Contudo, a estrutura não se altera:

- A função *setup* constrói três estruturas de dados: <Chave de substituição, "D_YEAR">, <"C_CUSTKEY", "C_CITY"> e <"S_SUPPKEY", "S_CITY">.
- A função *map* emite os pares <"D_YEAR. C_CITY. S_CITY", "L_REVENUE">, para os registos da tabela de factos que satisfação, tanto as condição de junção (isto é, se o "L_CUSTKEY" e "L_SUPPKEY" do registo estão contidos nas estruturas *HashMap* criadas na função *setup*) como os critérios de seleção das *queries*.

O output do primeiro trabalho é escrito num ficheiro em HDFS, que serve como *input* do segundo trabalho (Figura 45). Para ordenar os registos da forma solicitada, a função *map* utiliza uma estrutura de dados auxiliar, um par. Esta estrutura auxiliar tem como objetivo separar as diferentes ordenações do conjunto de dados. O Hadoop permite definir a ordenação das chaves através de uma classe que dita a ordenação de uma estrutura. O par utilizado para a ordenação dos dados é formado por dois campos: o ano que é ordenado de forma ascendente; e a composição dos atributos "L_REVENUE", "C_CITY" e "S_CITY" que são ordenados de forma descendente. A conjugação destes campos é emitida como chave no conjunto intermédio. Quanto ao valor, este não será incluído na operação de ordenação. Assim, a chave do conjunto intermédio fica ordenada da forma desejada. O *reducer* limita-se a rearranjar as posições dos atributos ("C_CITY", "S_CITY", "D_YEAR", "L_REVENUE") para escrever o output final. As configurações do segundo trabalho MR foram alteradas para realizar a ordenação pretendida. Por omissão, as chaves são ordenadas lexicograficamente.

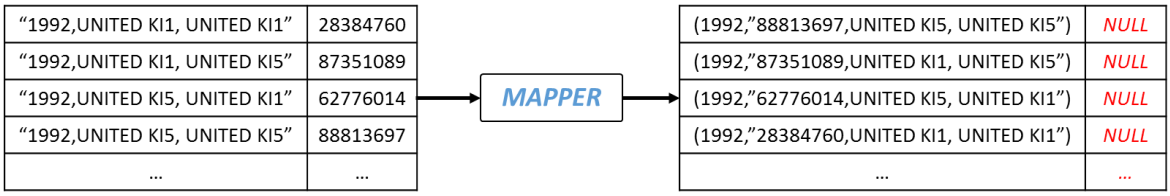


Figura 45: Segundo trabalho para as *queries* da terceira categoria (fase *map*)

Existe uma alternativa para processar as *queries* da terceira categoria em apenas um trabalho MR.

O Hadoop tem uma função, denominada *cleanup*, que executa no final das funções *map* e *reduce*. Este trabalho MR é semelhante ao primeiro trabalho descrito acima, apenas com algumas alterações ao nível do *reducer*. Depois do cálculo do somatório, a função *reduce* armazena o par numa estrutura de dados em memória (ex. *TreeMap*[24]). No final, a função *cleanup* ordena o output – conforme especificado pelas *queries* da terceira categoria – e emite os pares. Esta solução é mais eficiente, já que não implica um segundo trabalho MR.

Multi-way Join

As *queries* das categorias 2, 3 e 4 calculam junções com mais de duas tabelas. Algumas *queries* evitam estas junções através dos atributos que compõem a chave linha (ex. *query 2.1*). No entanto, o filtro que a chave linha produz não é suficiente. Em alguns casos, as restrições colocadas pelas *queries* conduzem à necessidade de calcular junções adicionais no *mapper*. Um exemplo deste tipo é a *query 4.3*:

```
if(parts.containsKey(partKey) && suppliers.containsKey(suppKey))
```

A função *map* apresentada pela Figura 43 implementa, na linha 5, o cálculo da junção entre um subconjunto da tabela de factos – obtido a partir da chave linha – e a tabela dimensão "PART". A expressão apresentada acima é utilizada com o mesmo propósito, isto é, calcular os registos da tabela de factos que satisfazem determinadas restrições. Porém, o número de restrições da *query 4.3* exige um acréscimo de tabelas dimensão para o cálculo. As condições ("S_NATION = 'UNITED STATES'" e "P_CATEGORY = 'MFGR#14'") requer a junção do subconjunto da tabela de factos com as tabelas dimensão "PART" e "SUPPLIER". As variáveis da expressão "parts" e "suppliers" correspondem a estruturas de dados *HashMap*, com as chaves de junção – chaves primárias – e os atributos projetados pela *query* para as tabelas dimensão (já calculados pela operação de seleção).

4.3 Avaliação dos resultados

Para avaliar o desempenho da estrutura de dados concebida para o teste SSB foi elaborado um estudo comparativo entre uma implementação MR e a de uma relativa a um SGBD relacional. Este

[24] http://docs.oracle.com/javase/6/docs/api/java/util/TreeMap.html

estudo foi organizado em três fatores de escala distintos. Isto é, foram gerados três DWs para representar três escalas diferentes. No final de cada teste, o DW utilizado foi removido para carregar um DW com um fator de escala diferente. Todos os resultados calculados pelas *queries* do SGBD e do MR foram comparados e validados. Os testes foram realizados numa única máquina em *Linux* baseado no *Debian* e no sistema de ficheiros ext4. As especificações da máquina e do *software* utilizado eram:

Hardware:

- CPU: Core 2 Duo T9550 2.66 GHz
- Memória: 4 GB
- Sistema operativo: Ubuntu 12.04 LTS 32 bit
- Disco rígido: SATA 5400 RPM

Software:

- Hadoop: 1.0.4
- Hbase: 0.94.2
- Java: versão 6 update 31
- MySQL: Community Server 5.5.29

O Hadoop, do mesmo modo que o HBase, foi configurado para ser executado em modo *pseudo-distributed*, isto é, cada *daemon* do Hadoop corresponde a um processo Java, que corre em separado, de forma a simular um *cluster* em pequena escala (White, 2012). Os dados do *Namenode* e *Datanode* foram armazenados em diretorias do disco local (sistema de ficheiros ext4). O Hadoop e o HBase utilizaram o Java 1.6, com algumas opções diferentes do padrão. Estas alterações na configuração estão apresentadas na Tabela 3.

Opção	Valor	Padrão
dfs.replication	1	3
dfs.block.size	128 MB	64 MB
mapred.child.java.opts	-Xmx1G	-Xmx200M
mapreduce.map.output.compress	true	false

Tabela 3: A configuração Hadoop

A primeira propriedade, a *dfs.replication*, definiu o número de réplicas para um bloco em HDFS. O valor '1' significa que o HDFS não replicou os blocos. Quando existe apenas um *Datanode*, o HDFS não tem a capacidade de replicar blocos por três *Datanode*, por essa razão optou-se por modificar essa propriedade. O aumento do tamanho de um bloco HDFS – o *dfs.block.size* – de 64 MB (por defeito) para 128 MB aliviou a pressão sobre a memória do *Namenode* e aumentou a quantidade de dados que um *mapper* tem a processar. Contudo, o paralelismo foi potencialmente reduzido, uma vez que o número de blocos diminuiu e, consequentemente, o número de *mappers* a o correr também diminuiu. A terceira propriedade, a *mapred.child.java.opts*, fixou o limite máximo de 1GB de memória (*JVM heap size*) para as tarefas *mapper* e *reducer*. Por último, a propriedade *mapreduce.map.output.compress* ativou a compressão dos dados gerados pelos *mappers*.

O *codec* – uma implementação de um algoritmo de compressão/descompressão – utilizado foi o *org.apache.hadoop.io.compress.SnappyCodec,* que representa o algoritmo de compressão Snappy[25]. O Hadoop suporta diferentes formatos de compressão, cada um com características diferentes. Os formatos LZO e o Snappy são otimizados para proporcionarem uma velocidade de processamento elevada e compressão razoável, apesar de uma compressão menos eficaz do que a do gzip. Porém, optou-se pelo Snappy, porque é significativamente mais rápido do que o LZO para a descompressão (White, 2012). Esta decisão estendeu-se ao armazenamento de dados nas tabelas em HBase. O Snappy supera o LZO, quando usado em Hadoop e HBase (Lars, 2011).

Relativamente à configuração do MySQL (Corporation and affiliates, 2013), optou-se pela configuração padrão, exceto a propriedade *buffer pool* – a memória disponível para armazenar os dados e os índices das tabelas – que passou a ter o valor de 1 GB. Por omissão, o *buffer pool* é de 128 MB. As tabelas do SSB dependem do mecanismo de armazenamento (*storage engine*) InnoDB (o *storage engine* por omissão). Este *storage engine* foi projetado para alcançar o máximo desempenho durante o processamento de grandes conjuntos de dados, suportar transações ACID (Gray, 1981) e integridade referencial. O InnoDB organiza os dados das tabelas em disco de forma a otimizar as consultas com base em chaves primárias. Cada tabela tem um índice *clustered* (uma árvore B+ tipicamente no atributo chave primária) que organiza os dados para minimizar os I/Os durante as pesquisas às chaves primárias.

[25] https://code.google.com/p/snappy/

Inicialmente, considerou-se o uso do *storage engine* MyISAM, mesmo sabendo-se que este *storage engine* disponha de melhores tempos de execução para as *queries* simples (as *queries* da primeira categoria), cerca de 55% relativamente ao InnoDB, para as restantes *queries* do teste SSB, o InnoDB supera o MyISAM. No geral, o InnoDB detém os tempos de execução mais baixos: cerca de 28% menos que o MyISAM.

4.3.1 Resultados

Nesta altura, o primeiro aspeto a considerar é o fator de escala (SF). Os gráficos apresentados na Figura 46 mostram o desempenho (em segundos) do conjunto de *queries* SSB para os diferentes fatores de escala. Os tempos de execução apresentados foram obtidos pelo cálculo do tempo médio de três amostras. Em MR, a execução de *queries* é efetuada por trabalhos constituídos por um *mapper* e um *reducer*. A *cache* das *queries* foram desativadas, para ambos os sistemas.

Para o caso de SF = 2, o MySQL foi claramente o "vencedor". Como era previsível, a latência dos trabalhos MR, desde o momento do envio do trabalho MR para o JobTracker até ao início da execução dos *mappers,* acrescentou segundos importantes no tempo de execução das *queries*. Excluindo o tempo de execução dos *mappers* e *reducers*, os trabalhos MR consumiram mais de 13 segundos, em média, apresentando uma variação entre 8 e 18 segundos. Este "*cold start*" é sintomático da implementação do Hadoop, não sendo uma característica inerente ao próprio modelo de programação MR. Em contraste, os SGBDs relacionais iniciam-se durante o *boot* do sistema operativo e, por essa razão, são considerados continuamente "*warm*", estando desde esse instante a aguardar a execução de *queries*. De referir que, a redução do tempo *startup* foi uma das primeiras otimizações realizadas nos SGBDs (Pavlo et al., 2009).

Para SF = 4, os tempos de resposta foram equilibrados. O MR foi o "vencedor" à exceção das *queries* do Flight 1 e 3. O Flight 3 processa as *queries* através de dois trabalhos MR. Isto fez com que o *overhead* – a latência dos trabalhos MR – aumenta-se para o dobro. Relativamente ao Flight 1, os tempos de resposta foram similares. Já para SF = 8, o MR foi também o "vencedor". Este apresentou um ganho entre 26% e 67% em relação aos tempos de resposta obtidos pelo MySQL. A resposta do MySQL em termos de fator de escala foi linear, enquanto que o MR respondeu de forma idêntica aos fatores de escala 2 e 8, aumentando ligeiramente para o 8.

Figura 46:Avaliação de desempenho – tempos de execução em segundos

A Figura 46 permite que se identifique outro aspeto importante: os Flights. O número de chaves linha a consultar para a satisfação das *queries* do teste SSB, representado pela Tabela 4, influenciaram o desempenho do MR. O Flight 1 representou, com 125 chaves linha, um número bastante superior aos demais, para aceder aos registos da tabela de factos (valores). O facto do cálculo das *queries* desta categoria se cingir a uma junção entre a tabela dimensão "DATE" e a tabela de factos, com seleção de duas medidas, não permitiu reduzir o número de chaves linha. A adição de tabelas dimensão – os Flights 2, 3 e 4 – no cálculo de *queries* possibilitou especificar de forma mais concreta as chaves linha. A seleção de atributos originários de várias tabelas dimensão diminui, de forma acentuada, o número de chaves linha.

Flight 1			**Flight 2**			**Flight 3**				**Flight 4**		
1.1	**1.2**	**1.3**	**2.1**	**2.2**	**2.3**	**3.1**	**3.2**	**3.3**	**3.4**	**4.1**	**4.2**	**4.3**
125	125	125	35	35	35	30	30	30	5	14	4	2

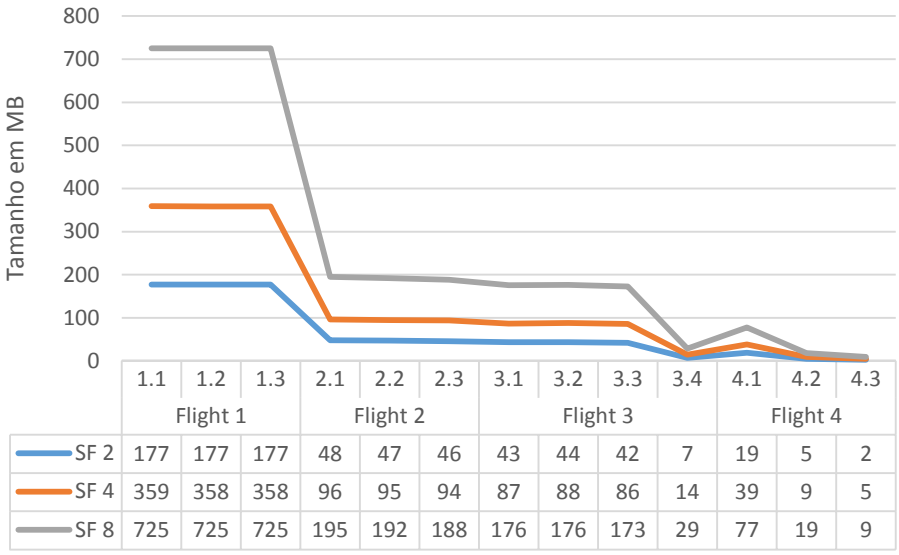
Tabela 4: Número de chaves linha acedidas

O número de chaves linha a aceder foi independente do fator de escala. No entanto, o volume de dados associados a cada chave aumentou de acordo com a escala. A Figura 47 apresenta esta relação, entre o fator de escala e o tamanho do *input* para as *queries* em MR. Nesta figura pode-se verificar uma redução progressiva do tamanho do *input* dos trabalhos MR à medida que a complexidade da *query* aumentava. As *queries* do Flight 1 – com o maior *input* – tiveram os tempos de execução mais elevados do teste, à exceção do Flight 3, com dois trabalhos MR. Já as *queries* pertencentes aos Flights 2, 3 e 4 beneficiaram claramente do filtro proporcionado pela chave linha. Para a *query 4.3*, a quantidade de dados a processar variou entre os 2 MB (SF = 2) e os 9 MB (SF = 8). De facto, a integração de todas as dimensões no cálculo de *queries* (em particular a *4.2* e *4.3*), bem como a *query 3.4*, permitiu selecionar um subconjunto da tabela de factos inferior a 1%. Desta forma, confirmam-se os tempos de resposta relativamente às *queries 4.2* e *4.3*.

Quanto ao output produzido pelos *mappers*, este também influenciou o tempo de resposta. Para cada *query*, o processamento do conjunto de KeyValues obtidos com o HBase produziu outro conjunto de pares chave/valor – o conjunto intermédio armazenado no disco local – de dimensões distintas, tanto no número de pares como no espaço em disco. A *query* que mais se destaca, dos três fatores de escala, é a *query 3.1*. Para o caso de SF = 8, esta *query* produziu 1 765 151 pares chave/valor e 50 MB de espaço em disco, valores bastante superiores aos produzidos pela média das *queries* do teste SSB (328 910 pares chave/valor e 7,684 MB). Dentro do mesmo Flight, destaca-se também a *query 3.4* que produziu o menor número de pares chave/valor e espaço em disco, com os valores 38 e 0,001 MB (SF = 8), respetivamente.

Relativamente aos tempos de resposta do MySQL, estes cresceram de acordo com a complexidade das *queries*, ao contrário dos tempos registados pelo MR. Isto deve-se ao facto das *queries* recorrerem a *full table scans*, enquanto o MR processa apenas uma fração do DW. Melhor dizendo, a principal razão para a degradação do desempenho de uma *query* no MySQL foi o processamento de uma quantidade de dados elevada, através de um número de leituras não indexadas (*Handler_read_rnd_next*) também elevado. Uma *query* é constituída por subtarefas que consomem tempo. Ao examinar as *queries* do teste SSB, as subtarefas que consumiram mais tempo foram a subtarefa *"Sending data"* (para o Flight 1) e a subtarefa *"Copying to tmp table"* (para os restantes Flights). A subtarefa *"Sending data"* pode representar várias atividades durante a execução de uma *query*, nomeadamente: o envio de dados entre as etapas da *query*, produzir o resultado ou

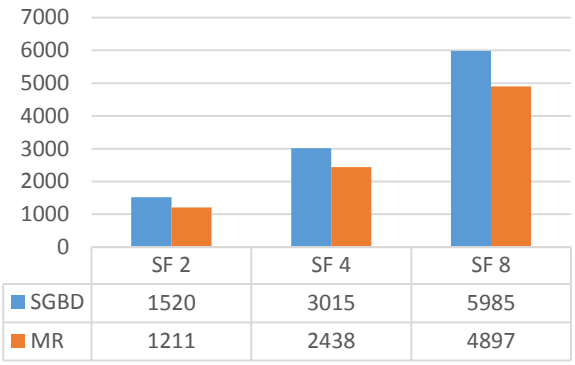
devolver o resultado ao cliente. Esta subtarefa inclui também a pesquisa de registos correspondentes em uma junção. Já a subtarefa *"Copying to tmp table"* representa o processamento da *query* e a cópia dos resultados para uma tabela de dados temporária, que é necessária para a cláusula GROUP BY (Schwartz et al., 2012).



	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2	3.3	3.4	4.1	4.2	4.3
	Flight 1			Flight 2			Flight 3				Flight 4		
SF 2	177	177	177	48	47	46	43	44	42	7	19	5	2
SF 4	359	358	358	96	95	94	87	88	86	14	39	9	5
SF 8	725	725	725	195	192	188	176	176	173	29	77	19	9

Figura 47: Tamanho dos *inputs* para as *queries* em MR

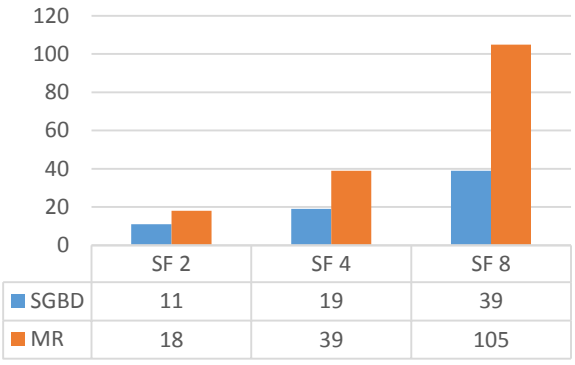
O espaço ocupado pelos DWs foi outro fator analisado. O gráfico da Figura 48 apresenta os vários tamanhos dos DWs em disco para os três fatores de escala. O espaço ocupado pelas tabelas dimensão foram incluídos no gráfico, apesar deste consumir os mesmos recursos tanto para o SGBD como para o MR. As estruturas de dados que armazenam as hierarquias estão incluídas no valor de MR. Quanto à tabela de factos "LINEORDER", o MR apresentou um espaço de armazenamento inferior ao SGBD. Isto deve-se simplesmente ao facto do MR utilizar a compressão de dados. É também possível verificar uma redução na proporção entre os valores, isto é, para SF = 2, o MR apresentou uma redução de 20% em relação ao SGBD, para SF = 4, 19%, e para SF = 8, 18%. Outro aspeto relacionado com o povoamento em MR foi a excessiva quantidade de dados produzida na fase intermédia do trabalho MR. Para SF = 8, os *mappers* produziram um conjunto de dados comprimido com o tamanho de 12,5 GB. Outras métricas indicam que, durante o trabalho MR, o total de dados lidos foi 35 GB e o total de dados escritos foi 47,5 GB, ou seja, para realizar um povoamento em MR foi necessário recorrer a bastante espaço em disco.



	SF 2	SF 4	SF 8
SGBD	1520	3015	5985
MR	1211	2438	4897

Figura 48: Espaço em disco (em MB)

Outra questão pertinente foi o tempo de povoamento. O gráfico da Figura 49 mostra os tempos obtidos para os três fatores de escala considerados. O HBase armazena os dados ordenados fisicamente (*clustered*) por chave. As importações são, normalmente, operações complexas e de longa duração (White, 2012). Como referido anteriormente, o trabalho MR para o povoamento da tabela de factos, mais as estruturas de dados que armazenam as hierarquias, lidam com uma quantidade de dados volumosa, o que causou um "pobre" desempenho por parte do MR. Este resultado era esperado devido à complexidade do povoamento. Este resultado contraria o observado em (Pavlo et al., 2009), no qual o MR se limita a copiar os dados do disco local para o HDFS. No entanto, o teste referido não utiliza o HBase, nem reorganiza o *input*. De referir, que o tempo da cópia do ficheiro que contém a tabela de factos para o HDFS não está representado na Figura 49.



	SF 2	SF 4	SF 8
SGBD	11	19	39
MR	18	39	105

Figura 49: Tempo do povoamento (em minutos)

4.3.2 Algumas Possíveis Otimizações

Perante os resultados do SGBD, importa referir que estes foram obtidos sem qualquer otimização, à exceção do relativo ao *buffer pool* a 1 GB. O processamento de *full table scans* degradou os tempos de resposta do MySQL. A fim de evitar o acesso a dados desnecessários, introduziram-se índices para verificar o seu impacto no desempenho das *queries* utilizadas no nosso teste. O InnoDB utiliza uma árvore B+ para estruturar os índices (Corporation and affiliates, 2013). Assim, de forma a tirar partido desta estrutura de indexação, foram adicionados vários índices para as tabela que pertencem ao DW. A tabela dimensão "DATE" não utilizou qualquer índice devido ao seu tamanho reduzido e não variável em função do fator de escala. Para as restantes tabelas foram criados índices nos seguintes atributos:

- **LINEORDER**: L_ORDERDATE, L_PARTKEY, L_CUSTKEY e L_SUPPKEY;
- **PART**: P_BRAND1, P_CATEGORY, P_MFGR;
- **CUSTOMER**: C_CITY, C_NATION, C_REGION;
- **SUPPLIER**: S_CITY, S_NATION, S_REGION.

Porém, a inclusão destes índices não se revelou benigna, uma vez que algumas das *queries* do SSB tiveram um desempenho muito pior (ex. para SF = 4, as *query 3.1*, *4.1* e *4.2* têm um tempo de execução superior a 3 horas). Para as tabelas de grande dimensão esta técnica não é eficiente (Schwartz et al., 2012). Mesmo com um *buffer pool* maior (2 GB), o *overhead* criado pelos índices, bem como o esforço necessário para posteriormente se utilizarem os índices, acrescentou trabalho na busca dos registos requeridos para a satisfação das *queries*. Para SF = 2, os índices, juntamente com o *buffer pool*, demonstraram uma melhoria do desempenho na generalidade das *queries*. Porém, quando o fator de escala aumenta para 4, unicamente, as *queries 2.3* e *3.4*, beneficiaram dos índices, com reduções dramáticas no tempo de resposta. Estas duas *queries* tiveram tempos de resposta inferiores a 1 segundo (para SF = 2) e inferiores a 8 segundos (para SF = 4). Ainda que o *query optimizer* identifique-se os índices para as *queries* do Flight 1, este decidiu não prosseguir com a sua utilização na execução das *queries*. Genericamente, os tempos de povoamento e o espaço em disco dos DWs aumentaram cerca de 200% e 166%, respetivamente. Perante estes resultados, não se realizaram testes para o caso de SF = 8.

Para se compreender um pouco melhor o fraco desempenho dos índices, a execução da *query 4.1* foi examinada (para SF = 4). Neste caso, o *query optimizer* escolheu o plano de execução que está apresentado na Figura 50.

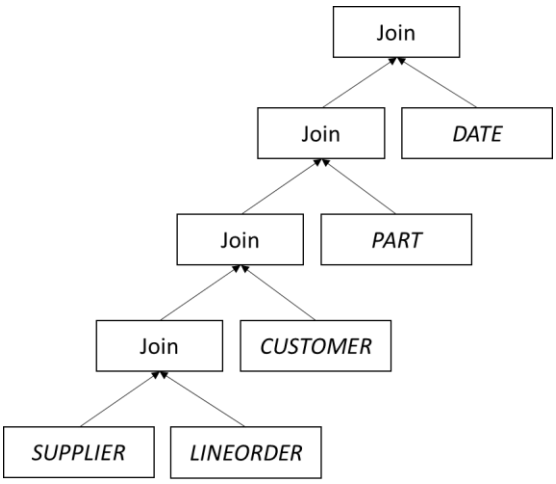


Figura 50: O plano de execução da *query 4.1*

O MySQL começou pela tabela dimensão "SUPPLIER", que percorre o índice (*index lookup*) criado para o atributo "S_REGION" e retorna as chaves primárias correspondentes a um valor - neste caso o valor 'AMERICA'. Estas chaves primárias foram os valores de referência utilizados para a operação de junção com a tabela "LINEORDER". Por sua vez, a tabela de factos utilizou o índice definido para o atributo "L_SUPPKEY" para calcular os registos que satisfazem a junção. As restantes junções foram calculadas sem qualquer recurso a índices, à exceção dos índices da chave primária. A utilização do índice na tabela "SUPPLIER" permitou calcular as chaves primárias, necessárias na operação de junção, sem aceder à tabela "SUPPLIER". Caso seja projetado um atributo (como é o caso da *query 4.2*, que projeta o atributo "S_NATION"), a execução da *query* requeria o *lookup* a dois índices: primeiro, ao índice secundário (criado para o atributo "S_REGION"), que obtém os valores das chaves primárias e, segundo, ao índice – uma árvore B+ *clustered* – produzido para o atributo definido como chave primária. Ou seja, realizou o dobro do trabalho (Schwartz et al., 2012).

As *queries* que apresentaram os piores desempenhos tiveram um número muito elevado de pedidos para a leitura de registos com base em índices (*Handler_read_key*), o que indica que as tabelas se encontravam indexadas corretamente para as *queries* do teste (Corporation and

affiliates, 2013). No entanto, em algumas circunstâncias, o acesso sequencial é mais rápido do que o acesso aleatório (Schwartz et al., 2012). Um teste de leitura ao disco (realizado pelo *sysbench*) demonstrou este facto. Para um *input* de 5 GB, o acesso sequencial teve um rendimento (*throughput*) de 39.166 Mb por segundo, que é um valor muito superior ao alcançado pelo acesso aleatório de 2.6441 Mb por segundo. O acesso aleatório beneficiou dos sistemas de *caching* disponíveis. Estes ajudam a evitar muitos dos acessos dispendiosos ao disco, atuando como uma primeira linha de disponibilização de dados. Porém, o tamanho do DW com o fator de escala 4 (5 GB) é muito superior ao tamanho do *buffer pool* (2 GB). Os acessos aleatórios foram executados, tipicamente, para encontrar um registo. Contudo, é necessário ler uma página de dados inteira, em que a maior parte dos dados não são necessários para a satisfação da *query*. Assim, muito do trabalho realizado é desperdiçado.

Uma possível solução alternativa seria a construção de um índice *clustered*, sobre atributos diferentes. Todavia, o InnoDB seleciona, em primeiro lugar, o atributo definido como chave primária para o índice *clustered*. Isto é, para usufruir desses índices adicionais, é necessário "desrespeitar" o esquema do próprio DW. As *queries* do teste SSB, assim como a generalidade das *queries* OLAP, executam usualmente várias agregações sobre os dados. Assim, normalmente, a satisfação destas *queries* não requer apenas um registo, mas sim um grupo de registos. O particionamento adequa-se para o processamento de agregações em tabelas extremamente grandes, uma vez que pode apontar para conjuntos de registos de interesse para as *queries* (Schwartz et al., 2012).

Com o intuito de explorar o desempenho do MR alterou-se a chave linha. A nova chave linha baseia-se na organização sugerida pelas *queries* teste SSB (O'Neil et al., 2007), que propõe a ordenação dos registos da tabela de factos pelas dimensões "DATE", "PART" e "CUSTOMER" ou "SUPPLIER". A chave linha gerada inclui os mesmos atributos da chave linha anterior, mas agora dispostos da seguinte forma:

$$< D\_YEAR >.< P\_MFGR >.< C\_REGION >.< S\_REGION >$$

Esta chave foi testada somente para o caso de SF = 8. Todavia, apresentou os resultados esperados - o mesmo desempenho. Os registos da tabela de factos mantêm-se ordenados pelo atributo "D_YEAR" – utilizado pela generalidade das *queries* – e, por essa razão, o desempenho foi

semelhante. Para o segundo Flight, onde o atributo "D_YEAR" não está incluído, o HBase foi capaz de filtrar os factos pelos atributos "P_MFGR" e "S_REGION", ocupando a segunda e quarta posição da chave linha, respetivamente (precisamente a ordem inversa da chave linha anterior). Outro aspeto verificado neste teste é a divisão da coluna família m em duas: a primeira com as medidas "L_QUANTITY", "L_EXTENDEDPRICE", "L_DISCOUNT" e a segunda coluna com as medidas "L_REVENUE" e "L_SUPPLYCOST". Assim, o DW passa a ter três colunas família, o que permite evitar algumas das medidas desnecessárias para o processamento das *queries*. Esta decisão resultou numa redução dos dados processados pelas *queries* em cerca de 13%. No entanto, o tempo de povoamento, bem como o espaço ocupado em disco foram superiores. A tabela de factos passou a ter um tamanho de 6306 MB, com um tempo de povoamento de 126 minutos.

Capítulo 5

Conclusões e Trabalho Futuro

5.1 Uma Apreciação Crítica

A aquisição de dados em organizações e empresas tornou-se progressivamente mais fácil. O aumento da disponibilidade e o interesse dos dados em texto (ex. emails ou revisões de produtos) conduziram a novos desafios no domínio da *Business Intelligence*. DWs com um volume de dados de 10 a 100 *terabytes*, ou superior, é um cenário cada vez mais comum. Tudo isto faz crescer o desejo de edificar uma plataforma de dados de baixo custo que seja capaz de sustentar enormes volumes de dados. Por entre as vistas multidimensionais produzidas sobre os dados armazenados num DW, os sistemas de processamento analítico permitem aos utilizadores realizarem operações de *roll-up*, *drill-down*, *slice and dice* e *pivot*. Neste âmbito, um SGBD relacional de *backend* (ROLAP) é, tradicionalmente, escolhido para armazenar e consultar os dados de um DW. De forma a sustentar as operações OLAP, os sistemas ROLAP mapeiam o modelo multidimensional, bem com as operações sobre relações e *queries* SQL. A maioria dos sistemas ROLAP utilizam um esquema em estrela para representar o modelo de dados multidimensional. Nestes casos, a base de dados é vulgarmente composta por uma única tabela de factos rodeada por tabelas dimensão. Cada registo numa tabela de factos é constituído por um conjunto de chaves estrangeiras para as tabelas dimensão e um conjunto de atributos que asseguram as diversas métricas de análise tendo em conta as dimensões definidas. Cada dimensão representa um eixo de análise para o modelo multidimensional (Chaudhuri et al., 2011).

Durante as últimas décadas assistiu-se ao desenvolvimento de várias estruturas de dados, otimizações e técnicas de processamento de *queries* especialmente orientadas para o suporte à execução de *queries* SQL sobre grandes volumes de dados. Um tempo de execução reduzido é um requisito fundamental para as aplicações de BI. Os DWs são tipicamente instalados e configurados (*deployed*) em bases de dados relacionais paralelas para que as *queries* SQL sejam executadas com latências baixas. O processamento paralelo reponde, de modo satisfatório, à necessidade de trabalhar com conjuntos de dados enormes e de forma eficiente.

O MR, assim como os SGBDs paralelos, utiliza uma arquitetura *shared-nothing* e proporciona um ambiente de programação de alto nível e paralelo. Ainda que possa parecer que o MR e os SGBDs paralelos se foquem em públicos diferentes, é possível escrever praticamente qualquer tarefa que se execute em paralelo através de um conjunto de *queries* ou trabalhos em MR (Pavlo et al., 2009). As plataformas baseadas em sistemas de ficheiros distribuídos e MR foram instaladas e configuradas – com sucesso – em *clusters*, com um número de nodos superior aos suportados por SGBD paralelos. Enquanto as bases de dados paralelas escalam, sem dificuldade, até às dezenas de nodos, o MR escala até aos milhares de nodos. A carga de trabalho realizado numa análise de dados caracteriza-se, essencialmente, por incorporar operações de *scan* gigantes, agregações multidimensionais ou *star-joins* que têm a possibilidade de serem executadas em paralelo, por arquiteturas *shared-nothing* (Chaudhuri et al., 2011).

Nesse sentido, este trabalho focou-se no estudo de uma estrutura multidimensional de dados em MR, abordando, principalmente, as questões de armazenamento e de acesso aos dados. A complexidade das análises e dos volumes de dados excessivos provocam sempre alguma apreensão relativamente ao tempo de resposta. Os SGBDs e o MR diferem em alguns aspetos. Relativamente ao armazenamento, todos os SGBDs exigem que os dados se encontrem em conformidade com um esquema bem definido, enquanto que o MR consente que os dados se encontrem num qualquer formato arbitrário. Outras disparidades entre os modelos referidos, compreendem a forma como estes sistemas indexam, comprimem e distribuem os dados, bem como as estratégias para a execução de *queries* (Pavlo et al., 2009). Os SDWs baseados em MR, em particular na *framework* Hadoop, desempenham diariamente um papel fundamental, por exemplo, na execução de análises *web click-stream* ou em aplicações de *data mining*, entre outros. Por outro lado, os dados não estruturados, a gestão de processos ETL e a evolução de esquemas constituem também sérios desafios que ainda permanecem em aberto (Abelló et al., 2011).

A quantidade de dados não estruturados existente é quarto ou cinco vezes superior à quantidade de dados estruturados (Inmon et al., 2010). Os DWs tradicionais não lidam com dados não estruturados, adotando a estratégia "*one size fits all*", o que não se apropria para a resolução de tarefas de processamento de dados em grande escala (Lee et al., 2012). (Abadi et al., 2008) demostram que o armazenamento orientado à coluna num SDW supera claramente os SGBD relacionais, nos quais o armazenamento linha a linha predomina. Já o Apache Hive (Thusoo et al., 2009) e o Apache Pig (Olston et al., 2008) têm a capacidade de suportar dados não estruturados. Estes utilizam o HDFS para o armazenamento de dados e linguagens de *scripting* para realizar as análises sobre os dados. Este sistema de ficheiros distribuídos foi produzido com o fim de que o padrão mais eficiente para processar dados é: uma escrita, muitas leituras. Após o conjunto de dados ser carregado da fonte, ou gerado, este é submetido a processos de análise que podem envolver uma grande fração (ou todos) os dados armazenados.

O MR e o HDFS formam a base para processar grandes quantidades de dados. Todavia, este processamento manifesta-se inútil em acessos aleatórios aos dados, especialmente nas consultas em tempo real. O Bigtable, do mesmo modo que o HBase, foi projetado para armazenar dados não estruturados. Esta abordagem armazena os dados em pedaços não formatados, tratados como *strings* sem significado (*uninterpreted*), que são indexados por dois valores, linha e coluna, que são *strings* arbitrárias. As colunas têm a possibilidade de serem agrupadas em famílias, o que é, de certa forma, análogo ao armazenamento orientado à coluna. O RCFile, utilizado pelo Apache Hive e o Apache Pig, adota também o tipo de armazenamento HDFS, em que a chave linha organiza fisicamente os dados (*clustered*). As bases de dados relacionais podem também servir como alicerce para o armazenamento dos dados, como o HadooDB representa uma abordagem hibrida – SGBD paralelo e Hadoop – para a análise de dados. Assim, é possível usufruir dos meios de otimização de *queries* que os SGBDs utilizam, como é o caso dos índices (Lee et al., 2012).

As estratégias para a execução de *queries* através do modelo MR influenciou a estrutura de dados final, especialmente ao nível das junções. Esta operação, bastante comum em ambientes de análise, não é bem tratada pelas funções *map* e *reduce*. A partir do momento em que o MR foi projetado para processar um único *input*, as junções que exigem mais de dois *inputs* tem sido uma questão em aberto. As dependências funcionais entre os atributos que compõem as hierarquias determinam as partições da tabela de factos utilizadas no cálculo das *queries*. A técnica

materialized path expression tem um papel crucial em operações de projeção nas *queries* em MR, uma vez que filtram subconjuntos da tabela de factos relacionados com os predicados das *queries* e, por vezes, evita o cálculo de junções com as tabelas dimensão.

Uma outra questão prende-se com o particionamento vertical da tabela de factos, ou seja, a distribuição das medidas por colunas família. A coluna família constitui a parte menos dinâmica do modelo de dados do HBase. O acréscimo de colunas família traduz-se num aumento do espaço em disco ocupado pela tabela de factos e de manutenção uma vez que exige um maior número de operações. Todavia, a quantidade de dados a processar é mais reduzida. A divisão das medidas em três colunas família não se revelou proveitosa, quando comparado com duas colunas família, devido à pequena redução da quantidade de dados a processar. Contudo, num outro cenário, uma repartição maior das medidas da tabela de factos pode ser uma mais-valia para a redução dos tempos de resposta às *queries*, principalmente se a diferença da quantidade de dados a processar for bastante acentuada (fator de escala superior). A distribuição das medidas pelas colunas família precaveu o *overhead* de reconstrução de registos da tabela de factos durante o processamento das *queries*. Isto deve-se ao facto das medidas foram agrupadas com o conhecimento prévio das *queries*. As colunas família, porém, não se adaptam rapidamente a cargas de trabalho dinâmicas. Caso uma *query* exija medidas de diferentes colunas família, o processamento da *query* será mais complexo, já que impõe a reconstrução dos registos "*on the fly*". A replicação de medidas utilizadas frequentemente – a sobreposição de atributos – pode ultrapassar a falta de adaptabilidade rápida do HBase. Todavia precisa de mais espaço em disco (He et al., 2011).

Tanto para o SGBD relacional como para o MR, o tempo e o espaço consumido pelo povoamento foram discordantes. Na literatura é sugerido que o povoamento de dados em MR é claramente mais rápido do que o dos SGBDs, chegando a ser sugerido que o MR é preferível aos SGBDs para analises rápidas ou para um conjunto de dados alvo de poucas análises. Porém, estes povoamentos consistem na cópia dos ficheiros que contêm o conjunto de dados do disco local para o HDFS, seguido da replicação dos dados pelos nodos. Esta operação constitui apenas o primeiro passo do povoamento apresentado neste trabalho (Secção 4.2.1.). O processo de povoamento da tabela de factos "LINEORDER" organiza os registos pelas chaves linha. Esta organização aliada à compressão dos dados no final do processo aumentou o tempo deste povoamento. Quanto ao espaço consumido pelos DWs, os espaços ocupados pela tabela de factos "LINEORDER" e as estruturas dedicadas ao armazenamento das hierarquias, em HBase, é inferior ao espaço

despendido pelo SGBD. Isto deve-se ao facto de o HBase empregar compressão, enquanto que o SGBD não aplica qualquer tipo de compressão.

Os resultados obtidos dos testes de desempenho (Secção 4.3.) permitem retirar alguns dados interessantes. Para o DW de menor dimensão, repare-se que o tempo de arranque dos trabalhos MR – uma duração média de 13 segundos – desmotiva a utilização do Hadoop. O "*cold start*" é sintomático da implementação do Hadoop, não sendo contudo uma característica intrínseca do modelo de programação. Para o SGBD relacional, as *queries* com a duração de poucos segundos demoram significativamente mais tempo, quando executadas em MR, mesmo para conjuntos de dados de pequena dimensão. Os SGBDs relacionais são considerados ininterruptamente "*warm*", a aguardar a execução de *queries*. A redução do tempo *startup* foi uma das primeiras prioridades dos SGBDs. De facto, o MR está a dar os primeiros passos, relativamente aos SGBDs relacionais. Algum trabalho no futuro passará por reduzir este tempo no processamento. Tal como sugerido por (Dean and Ghemawat, 2010), o *overhead* do tempo *startup* pode ser tratado através da preservação dos processos *worker* permanentemente ativos, que estão a aguardar as invocações do MR. No entanto, para o DW de maior volume, é possível verificar que este *overhead* é irrelevante, relativamente ao tempo de processamento total. Desta forma, deduz-se que um volume da dados pequeno deve ser suportado por um SGBD relacional, devido à sua flexibilidade e maturidade. Já para um volume da dados grande, o MR + HBase representa uma solução viável, devido à sua capacidade de escalar (White, 2012).

Na maioria dos casos, um trabalho MR resume-se ao processamento de uma agregação GROUP BY por uma estrutura de dados par do género chave/valor. A função *map* seleciona, projeta e determina os agrupamentos, enquanto que a função *reduce* executa as agregações. O próprio *framework* ordena os pares chave/valor lexicograficamente por chave. Estas operações são tipicamente executadas por sistemas de processamento analítico. Assim, o MR enquadra-se incontestavelmente em ambientes de análise. No entanto, o MR pode parecer um modelo de programação rígido, uma vez que se cinge a processar pares chave/valor, através de um *input*. Um outro fator que também prejudica o tempo de execução das *queries* é o *parsing* dos dados "*on the fly*" armazenados em HBase. Tanto o valor do *qualifier* como o valor das medidas têm de ser interpretados (*deserialize*) para que estes tenham a possibilidade de ser processados. Contudo, isto não acontece nos SGBDs, já que, no momento do carregamento, os dados são analisados

(*parsing*) podendo-se extrair rapidamente as medidas dos registos da tabela de factos, essencialmente sem qualquer custo adicional.

Por definição, o processamento intensivo de dados (*data-intensive processing*) denota que o tamanho do conjunto de dados relevante excede a capacidade de memória e, deste modo, os dados são, forçosamente, mantidos em disco. Para os acessos aleatórios, os tempos de busca são essencialmente limitados pela natureza mecânica dos dispositivos. As computações devem ser organizadas de forma processar os dados sequencialmente, em lugar dos acessos aleatórios aos dados. Os acessos sequenciais aos dados são mais rápidos do que os acessos aleatórios (Lin and Dyer, 2010). De facto, a inclusão de índices do tipo árvore B+ por parte do SGBD revelou-se positiva unicamente quando o DW cabe em memória. Caso contrário, os índices deterioram o desempenho conseguido sem o recurso a otimizações, ou seja, os *full table scans* frequentes. Quanto ao MR, a sua organização dos dados em disco permite os acessos sequenciais a grupos de registos indexados pela chave-linha.

Existem algumas considerações adicionais acerca da instalação, configuração e declaração de *queries* para ambos os sistemas que convém referir. O Hadoop foi instalado sem grande esforço. A instalação do sistema consiste apenas numa configuração de ficheiros XML na qual se definem alguns parâmetros, como a localização dos dados ou a memória disponível para cada processo. Quanto à integração do HBase, esta revelou-se um processo difícil, principalmente naquilo que diz respeito à manutenção do servidor como disponível. Com efeito, o consumo excessivo de memória por parte do Hadoop em conjunção com o HBase conduziu a erros por falta de memória. A configuração dos parâmetros foi obtida através do método de tentativa/erro. Já a instalação e configuração do SGBD foi relativamente simples. Em relação à declaração das *queries*, o código SQL foi fornecido pelo teste que se escolheu, bem com as instruções necessárias para a criação do esquema em estrela. Por outro lado, o processo de adaptação das *queries* para a linguagem Java levantou algumas dificuldades. A primeira foi a de compatibilizar os pares chave/valor emitidos pelos *mappers* com o *input* dos *reducers*, isto é, a declaração do tipo de dados para a chave e para o valor. Um outro obstáculo surgiu na necessidade de recorrer a funções extra (ex. *setup*) - a literatura sugeria a implementação de apenas duas funções. A elaboração das *queries* em Java ficou também marcada pelas constantes alterações do código. De facto, um paradigma *"low-level"* complica a declaração de consultas a uma estrutura de dados, quando comparado com o SQL.

5.2 Uma Possível Orientação para Trabalho Futuro

O trabalho futuro a desenvolver no âmbito desta dissertação será, seguramente, alargar o estudo a ambientes com *clusters*, de forma a avaliar a estrutura de dados produzida, por exemplo: a variação do número de nodos, a variação do número de utilizadores e a alteração dos parâmetros de configuração, tais como o tamanho dos blocos em HDFS ou o número de *mappers*. A chave linha em HBase deverá ser objeto de estudo relativamente ao *data skew*, isto é, à variação da dimensão das partições - a forma como e onde os dados são armazenados é controlada por esta chave. As chaves compostas têm as propriedades dos índices *left-edge*. Um desenho apropriado destas chaves permite o crescimento dos dados de 10 até 10 milhões de entradas, conservando o mesmo desempenho de leitura e de escrita (Lars, 2011). O Apache Pig e o Apache Hive têm a capacidade de suportar, implicitamente, esquemas em estrela, ou mesmo, esquemas em floco de neve (Liu et al., 2011). Desta forma, o desempenho destas plataformas devem ser testadas. A inclusão das linguagens declarativas destas plataformas poderia também ser uma mais-valia para a expressão das *queries*. De facto, tanto o Hive como o Pig incorporam o HBase para o armazenamento dos dados.Este trabalho focou-se especialmente nos cálculos "*on the fly*" sobre a tabela de factos. Os dados armazenados em HBase respeitam a granularidade definida pelo esquema do DW. No entanto, a possibilidade de partir de um nível de detalhe superior – vistas materializadas – não deve ser descartada no futuro.

Também foi aqui realizada uma avaliação sobre a capacidade do modelo de programação MR suportar estruturas multidimensionais de dados. Para isso, este estudo apresentou o modelo de programação MR (capítulo 2), as estruturas multidimensionais de dados típicas de um ambiente de um DW, algumas técnicas utilizadas na satisfação de *queries* e uma análise da integração de MR em DW (capítulo 3). Relativamente ao estudo da inclusão de MR em SDW, foi exposta a sua aplicação em ambientes de processamento analítico face a um esquema em estrela, que, usualmente, é o mais utilizado para representar o modelo de dados multidimensional em ROLAP. O estudo de desenvolvido e analisado permitiu organizar de forma coerente as características de ambos os sistemas – o SGBD e o *framework* Hadoop – e conceber uma base de conhecimento mais sólida neste domínio. Este alicerce poderá ser aproveitado como um princípio para outros trabalhos que tenham o propósito de progredir o estudo neste campo.

A literatura da área salienta a incapacidade do MR utilizar índices para o processamento de dados. Neste panorama, a contribuição deste trabalho traduz-se num algoritmo de indexação e

povoamento de uma tabela de factos, beneficiando das tecnologias MR e Bigtable e uma análise experimental das características das consultas em MR e Bigtable, de forma a selecionar o algoritmo mais conveniente em relação à estrutura de dados que se pretende construir. De facto, é relativamente difícil planear uma estrutura de dados, sem o conhecimento prévio do conjunto de *queries* esta vai sustentar eficientemente. Uma estrutura de dados flexível, capaz de se adaptar dinamicamente a uma nova *query* atípica à estrutura seria uma vertente interessante a explorar numa linha de trabalho futuro. A plataforma MR possibilita a escalabilidade sustentada e simplificada, a preços mais reduzidos do que as soluções tipicamente fornecidas.

Bibliografia

ABADI, D., MADDEN, S. & FERREIRA, M. 2006. Integrating compression and execution in column-oriented database systems. Proceedings of the 2006 ACM SIGMOD international conference on Management of data. Chicago, IL, USA: ACM.

ABADI, D. J. 2008. Query execution in column-oriented database systems. Massachusetts Institute of Technology.

ABADI, D. J., MADDEN, S. R. & HACHEM, N. 2008. Column-stores vs. row-stores: how different are they really? Proceedings of the 2008 ACM SIGMOD international conference on Management of data. Vancouver, Canada: ACM.

ABADI, D. J., MARCUS, A., MADDEN, S. R. & HOLLENBACH, K. Scalable semantic web data management using vertical partitioning. Proceedings of the 33rd international conference on Very large data bases, 2007. VLDB Endowment, 411-422.

ABELLÓ, A., FERRARONS, J. & ROMERO, O. 2011. Building cubes with MapReduce. Proceedings of the ACM 14th international workshop on Data Warehousing and OLAP. Glasgow, Scotland, UK: ACM.

ABOUZEID, A., BAJDA-PAWLIKOWSKI, K., ABADI, D., SILBERSCHATZ, A. & RASIN, A. 2009. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proceedings of the VLDB Endowment, 2, 922-933.

AFRATI, F. N. & ULLMAN, J. D. Optimizing joins in a map-reduce environment. Proceedings of the 13th International Conference on Extending Database Technology, 2010. ACM, 99-110.

AILAMAKI, A., DEWITT, D. J., HILL, M. D. & SKOUNAKIS, M. Weaving Relations for Cache Performance. VLDB, 2001. 169-180.

BAYER, R. & MCCREIGHT, E. M. 1972. Organization and maintenance of large ordered indexes. Acta informatica, 1, 173-189.

BLANAS, S., PATEL, J. M., ERCEGOVAC, V., RAO, J., SHEKITA, E. J. & TIAN, Y. 2010. A comparison of join algorithms for log processing in MaPreduce. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. Indianapolis, Indiana, USA: ACM.

BONCZ, P. A., MANEGOLD, S. & KERSTEN, M. L. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. Proceedings of the 25th International Conference on Very Large Data Bases. Morgan Kaufmann Publishers Inc.

BONDI, A. B. 2000. Characteristics of scalability and their impact on performance. Proceedings of the 2nd international workshop on Software and performance. Ottawa, Ontario, Canada: ACM.

BORTHAKUR, D. 2007. The Hadoop Distributed File System: Architecture and Design.

CAPRIOLO, E., WAMPLER, D. & RUTHERGLEN, J. 2012. Programming Hive, O'Reilly Media.

CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A. & GRUBER, R. E. 2006. Bigtable: a distributed storage system for structured data. Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7. Seattle, WA: USENIX Association.

CHAUDHURI, S. & DAYAL, U. 1997. An overview of data warehousing and OLAP technology. SIGMOD Rec., 26, 65-74.

CHAUDHURI, S., DAYAL, U. & NARASAYYA, V. 2011. An overview of business intelligence technology. Commun. ACM, 54, 88-98.

CHEN, S. & SCHLOSSER, S. W. 2008. Map-Reduce Meets Wider Varieties of Applications. Pittsburgh, USA.

CODD, E. F. 1970. A relational model of data for large shared data banks. Commun. ACM, 13, 377-387.

CODD, E. F., CODD, S. B. & SALLEY, C. T. 1993. Providing OLAP (On-line Analytical Processing) to User-analysts: An IT Mandate, Codd & Associates.

COMER, D. 1979. Ubiquitous B-Tree. ACM Comput. Surv., 11, 121-137.

COMMUNITY, A. 2013. PoweredBy - Hadoop Wiki [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy 2013].

COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D. & YERNENI, R. 2008. PNUTS: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment, 1, 1277-1288.

COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R. & SEARS, R. Benchmarking cloud serving systems with YCSB. Proceedings of the 1st ACM symposium on Cloud computing, 2010. ACM, 143-154.

CORPORATION, O. & AFFILIATES. 2013. MySQL 5.5 Reference Manual [Online]. Available: https://dev.mysql.com/doc/refman/5.5/en/ 2013].

DEAN, J. & GHEMAWAT, S. 2004. MapReduce: simplified data processing on large clusters. Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6. San Francisco, CA: USENIX Association.

DEAN, J. & GHEMAWAT, S. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM, 51, 107-113.

DEAN, J. & GHEMAWAT, S. 2010. MapReduce: a flexible data processing tool. Commun. ACM, 53, 72-77.

DEWITT, D. & GRAY, J. 1992. Parallel database systems: the future of high performance database systems. Commun. ACM, 35, 85-98.

EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S.-H., QIU, J. & FOX, G. 2010. Twister: a runtime for iterative MapReduce. Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. Chicago, Illinois: ACM.

FURTADO, P. 2011. A survey of parallel and distributed data warehouses. Integrations of Data Warehousing, Data Mining and Database Technologies: Innovative Approaches, 148.

GATES, A. 2011. Programming Pig, O'Reilly.

GHEMAWAT, S., GOBIOFF, H. & LEUNG, S.-T. 2003. The Google file system. Proceedings of the nineteenth ACM symposium on Operating systems principles. Bolton Landing, NY, USA: ACM.

GOLFARELLI, M. & RIZZI, S. 2009. Data Warehouse Design: Modern Principles and Methodologies, McGraw-Hill, Inc.

GRAEFE, G. & SHAPIRO, L. D. Data Compression and Database Performance. 1991 In Proc. ACM/IEEE-CS Symp. On Applied Computing.

GRAY, J. 1981. The transaction concept: virtues and limitations (invited paper). Proceedings of the seventh international conference on Very Large Data Bases - Volume 7. Cannes, France: VLDB Endowment.

GUPTA, A. & MUMICK, I. S. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. IEEE Data Engineering Bulletin, 18, 3--18.

GUPTA, H., HARINARAYAN, V., RAJARAMAN, A. & ULLMAN, J. D. 1997. Index Selection for OLAP. Proceedings of the Thirteenth International Conference on Data Engineering. IEEE Computer Society.

HE, Y., LEE, R., HUAI, Y., SHAO, Z., JAIN, N., ZHANG, X. & XU, Z. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. Data Engineering (ICDE), 2011 IEEE 27th International Conference on, 2011. IEEE, 1199-1208.

HUFFMAN, D. A. 1952. A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE, 40, 1098-1101.

HUYN, N. 1997. Multiple-View Self-Maintenance in Data Warehousing Environments. Proceedings of the 23rd International Conference on Very Large Data Bases. Morgan Kaufmann Publishers Inc.

INMON, W. H. 2002. Building the Data Warehouse, John Wiley & Sons, Inc.

INMON, W. H., STRAUSS, D. & NEUSHLOSS, G. 2010. DW 2.0: The Architecture for the Next Generation of Data Warehousing: The Architecture for the Next Generation of Data Warehousing, Elsevier Science.

KIMBALL, R. 1996. The data warehouse toolkit: practical techniques for building dimensional data warehouses, John Wiley \\& Sons, Inc.

KIMBALL, R., REEVES, L., THORNTHWAITE, W., ROSS, M. & THORNWAITE, W. 1998. The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing and Deploying Data Warehouses, John Wiley \& Sons, Inc.

KIMBALL, R. & ROSS, M. 2002. The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling, John Wiley \& Sons, Inc.

LAKSHMAN, A. & MALIK, P. Cassandra: structured storage system on a p2p network. Proceedings of the 28th ACM symposium on Principles of distributed computing, 2009. ACM, 5-5.

LARS, G. 2011. HBase: The Definitive Guide, O'Reilly Media, Incorporated.

LEE, K.-H., LEE, Y.-J., CHOI, H., CHUNG, Y. D. & MOON, B. 2012. Parallel data processing with MapReduce: a survey. ACM SIGMOD Record, 40, 11-20.

LIN, J. & DYER, C. 2010. Data-Intensive Text Processing with MapReduce, Morgan and Claypool Publishers.

LIU, X., THOMSEN, C. & PEDERSEN, T. B. 2011. The ETLMR MapReduce-based ETL framework. Proceedings of the 23rd international conference on Scientific and statistical database management. Portland, OR: Springer-Verlag.

MOODY, D. L. & KORTINK, M. A. R. 2000. From Enterprise Models to Dimensional Models: A Methodology for Data Warehouse and Data Mart Design.

O'NEIL, P. & QUASS, D. 1997. Improved query performance with variant indexes. SIGMOD Rec., 26, 38-49.

O'NEIL, P. E. 1989. Model 204 Architecture and Performance. Proceedings of the 2nd International Workshop on High Performance Transaction Systems. Springer-Verlag.

O'NEIL, P. E., O'NEIL, E. J. & CHEN, X. 2007. The Star Schema Benchmark (SSB).

OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R. & TOMKINS, A. 2008. Pig latin: a not-so-foreign language for data processing. Proceedings of the 2008 ACM SIGMOD international conference on Management of data. Vancouver, Canada: ACM.

PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S. & STONEBRAKER, M. 2009. A comparison of approaches to large-scale data analysis. Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. Providence, Rhode Island, USA: ACM.

POESS, M. & FLOYD, C. 2000. New TPC benchmarks for decision support and web commerce. SIGMOD Rec., 29, 64-71.

PONNIAH, P. 2004. Data Warehousing Fundamentals: A Comprehensive Guide for IT Professionals, Wiley.

SCHWARTZ, B., ZAITSEV, P. & TKACHENKO, V. 2012. High Performance MySQL: Optimization, Backups, and Replication, O'Reilly Media, Inc.

SRIRAMA, S. N., JAKOVITS, P. & VAINIKKO, E. 2012. Adapting scientific computing problems to clouds using MapReduce. Future Gener. Comput. Syst., 28, 184-192.

STONEBRAKER, M. 1986. The case for shared nothing. Database Engineering Bulletin, 9, 4-9.

STONEBRAKER, M., ABADI, D., DEWITT, D. J., MADDEN, S., PAULSON, E., PAVLO, A. & RASIN, A. 2010. MapReduce and parallel DBMSs: friends or foes? Communications of the ACM, 53, 64-71.

STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S. & O'NEIL, E. C-store: a column-oriented DBMS. Proceedings of the 31st international conference on Very large data bases, 2005. VLDB Endowment, 553-564.

THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P. & MURTHY, R. 2009. Hive: a warehousing solution over a map-reduce framework. Proc. VLDB Endow., 2, 1626-1629.

THUSOO, A., SHAO, Z., ANTHONY, S., BORTHAKUR, D., JAIN, N., SARMA, J. S., MURTHY, R. & LIU, H. 2010. Data warehousing and analytics infrastructure at facebook. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. Indianapolis, Indiana, USA: ACM.

WHITE, T. 2012. Hadoop: the definitive guide, O'Reilly Media, Inc.

YANG, H.-C., DASDAN, A., HSIAO, R.-L. & PARKER, D. S. 2007. Map-reduce-merge: simplified relational data processing on large clusters. Proceedings of the 2007 ACM SIGMOD international conference on Management of data. Beijing, China: ACM.

YANG, J., KARLAPALEM, K. & LI, Q. 1997. Algorithms for Materialized View Design in Data Warehousing Environment. Proceedings of the 23rd International Conference on Very Large Data Bases. Morgan Kaufmann Publishers Inc.

ZIV, J. & LEMPEL, A. 1978. Compression of individual sequences via variable-rate coding. Information Theory, IEEE Transactions on, 24, 530-536.

Apêndice A. Star schema benchmark (SSB)

Tabela dimensão "DATE":

```
CREATE TABLE dat (
    D_DATEKEY           BIGINT      NOT NULL    PRIMARY KEY,
    D_DATE              CHAR(18)    NULL,
    D_DAYOFWEEK         CHAR(8)     NULL,
    D_MONTH             CHAR(9)     NULL,
    D_YEAR              SMALLINT    NULL,
    D_YEARMONTHNUM      INTEGER     NULL,
    D_YEARMONTH         CHAR(7)     NULL,
    D_DAYNUMINWEEK      TINYINT     NULL,
    D_DAYNUMINMONTH     TINYINT     NULL,
    D_DAYNUMINYEAR      SMALLINT    NULL,
    D_MONTHNUMINYEAR    TINYINT     NULL,
    D_WEEKNUMINYEAR     TINYINT     NULL,
    D_SELLINGSEASON     CHAR(12)    NULL,
    D_LASTDAYINWEEKFL   TINYINT(1)  NULL,
    D_LASTDAYINMONTHFL  TINYINT(1)  NULL,
    D_HOLIDAYFL         TINYINT(1)  NULL,
    D_WEEKDAYFL         TINYINT(1)  NULL);
```

Tabela dimensão "PART":

```
CREATE TABLE part (
    P_PARTKEY   BIGINT      NOT NULL    PRIMARY KEY,
    P_NAME      VARCHAR(22) NULL,
    P_MFGR      CHAR(6)     NULL,
    P_CATEGORY  CHAR(7)     NULL,
    P_BRAND1    CHAR(9)     NULL,
    P_COLOR     VARCHAR(11) NULL,
    P_TYPE      VARCHAR(25) NULL,
    P_SIZE      TINYINT     NULL,
    P_CONTAINER CHAR(10)    NULL);
```

Tabela dimensão "SUPPLIER":

```
CREATE TABLE supplier (
    S_SUPPKEY   INTEGER     NOT NULL    PRIMARY KEY,
```

```
    S_NAME      CHAR(25)    NULL,
    S_ADDRESS   VARCHAR(25) NULL,
    S_CITY      CHAR(10)    NULL,
    S_NATION    CHAR(15)    NULL,
    S_REGION    CHAR(12)    NULL,
    S_PHONE     CHAR(15)    NULL);
```

Tabela dimensão "CUSTOMER":

```
CREATE TABLE customer (
    C_CUSTKEY       BIGINT      NOT NULL    PRIMARY KEY,
    C_NAME          VARCHAR(25) NULL,
    C_ADDRESS       VARCHAR(25) NULL,
    C_CITY          CHAR(10)    NULL,
    C_NATION        CHAR(15)    NULL,
    C_REGION        CHAR(12)    NULL,
    C_PHONE         CHAR(15)    NULL,
    C_MKTSEGMENT    CHAR(10)    NULL);
```

Tabela de factos "LINEORDER":

```
CREATE TABLE lineorder (
    LO_ORDERKEY         BIGINT    NOT NULL,
    LO_LINENUMBER       TINYINT   NOT NULL,
    LO_CUSTKEY          BIGINT    NOT NULL   REFERENCES customer (C_CUSTKEY),
    LO_PARTKEY          BIGINT    NOT NULL   REFERENCES part (P_PARTKEY),
    LO_SUPPKEY          INTEGER   NOT NULL   REFERENCES supplier (S_SUPPKEY),
    LO_ORDERDATE        BIGINT    NOT NULL   REFERENCES dat (D_DATEKEY),
    LO_ORDERPRIORITY    CHAR(15)  NULL,
    LO_SHIPPRIORITY     CHAR(1)   NULL,
    LO_QUANTITY         TINYINT   NULL,
    LO_EXTENDEDPRICE    DECIMAL   NULL,
    LO_ORDTOTALPRICE    DECIMAL   NULL,
    LO_DISCOUNT         DECIMAL   NULL,
    LO_REVENUE          DECIMAL   NULL,
    LO_SUPPLYCOST       DECIMAL   NULL,
    LO_TAX              TINYINT   NULL,
    LO_COMMITDATE       BIGINT    NOT NULL REFERENCES dat (D_DATEKEY),
    LO_SHIPMODE         CHAR(10)  NULL,
    PRIMARY KEY (LO_ORDERKEY, LO_LINENUMBER));
```

Apêndice B. Queries SSB (SQL)

Query 1.1:

```
SELECT SUM(lo_extendedprice*lo_discount) AS revenue
FROM lineorder, dat
WHERE lo_orderdate = d_datekey
    AND d_year = 1993
    AND lo_discount BETWEEN 1 AND 3
    AND lo_quantity < 25;
```

Query 1.2:

```
SELECT SUM(lo_extendedprice*lo_discount) AS revenue
FROM lineorder, dat
WHERE lo_orderdate = d_datekey
    AND d_yearmonthnum = 199401
    AND lo_discount BETWEEN 4 AND 6
    AND lo_quantity BETWEEN 26 AND 35;
```

Query 1.3:

```
SELECT SUM(lo_extendedprice*lo_discount) AS revenue
FROM lineorder, dat
WHERE lo_orderdate = d_datekey
    AND d_weeknuminyear = 6
    AND d_year = 1994
    AND lo_discount BETWEEN 5 AND 7
    AND lo_quantity BETWEEN 26 AND 35;
```

Query 2.1:

```
SELECT SUM(lo_revenue), d_year, p_brand1
FROM lineorder, dat, part, supplier
WHERE lo_orderdate = d_datekey
    AND lo_partkey = p_partkey
    AND lo_suppkey = s_suppkey
    AND p_category = 'MFGR#12'
    AND s_region = 'AMERICA'
GROUP BY d_year, p_brand1
```

```
ORDER BY d_year, p_brand1;
```

Query 2.2:

```
SELECT SUM(lo_revenue), d_year, p_brand1
FROM lineorder, dat, part, supplier
WHERE lo_orderdate = d_datekey
    AND lo_partkey = p_partkey
    AND lo_suppkey = s_suppkey
    AND p_brand1 BETWEEN 'MFGR#2221' AND 'MFGR#2228'
    AND s_region = 'ASIA'
GROUP BY d_year, p_brand1
ORDER BY d_year, p_brand1;
```

Query 2.3:

```
SELECT SUM(lo_revenue), d_year, p_brand1
FROM lineorder, dat, part, supplier
WHERE lo_orderdate = d_datekey
    AND lo_partkey = p_partkey
    AND lo_suppkey = s_suppkey
    AND p_brand1 = 'MFGR#2221'
    AND s_region = 'EUROPE'
GROUP BY d_year, p_brand1
ORDER BY d_year, p_brand1;
```

Query 3.1:

```
SELECT c_nation, s_nation, d_year, SUM(lo_revenue) AS revenue
FROM customer, lineorder, supplier, dat
WHERE lo_custkey = c_custkey
    AND lo_suppkey = s_suppkey
    AND lo_orderdate = d_datekey
    AND c_region = 'ASIA'
    AND s_region = 'ASIA'
    AND d_year >= 1992 AND d_year <= 1997
GROUP BY c_nation, s_nation, d_year
ORDER BY d_year ASC, revenue DESC;
```

Query 3.2:

```
SELECT c_city, s_city, d_year, SUM(lo_revenue) AS revenue
FROM customer, lineorder, supplier, dat
WHERE lo_custkey = c_custkey
    AND lo_suppkey = s_suppkey
    AND lo_orderdate = d_datekey
    AND c_nation = 'UNITED STATES'
    AND s_nation = 'UNITED STATES'
    AND d_year >= 1992 AND d_year <= 1997
GROUP BY c_city, s_city, d_year
ORDER BY d_year ASC, revenue DESC;
```

Query 3.3:

```
SELECT c_city, s_city, d_year, SUM(lo_revenue) AS revenue
FROM customer, lineorder, supplier, dat
WHERE lo_custkey = c_custkey
    AND lo_suppkey = s_suppkey
    AND lo_orderdate = d_datekey
    AND (c_city='UNITED KI1' OR c_city='UNITED KI5')
    AND (s_city='UNITED KI1' OR s_city='UNITED KI5')
    AND d_year >= 1992 AND d_year <= 1997
GROUP BY c_city, s_city, d_year
ORDER BY d_year ASC, revenue DESC;
```

Query 3.4:

```
SELECT c_city, s_city, d_year, SUM(lo_revenue) AS revenue
FROM customer, lineorder, supplier, dat
WHERE lo_custkey = c_custkey
    AND lo_suppkey = s_suppkey
    AND lo_orderdate = d_datekey
    AND (c_city='UNITED KI1' OR c_city='UNITED KI5')
    AND (s_city='UNITED KI1' OR s_city='UNITED KI5')
    AND d_yearmonth = 'Dec1997'
GROUP BY c_city, s_city, d_year
ORDER BY d_year ASC, revenue DESC;
```

Query 4.1:

```
SELECT d_year, c_nation,  SUM(lo_revenue-lo_supplycost) AS profit
FROM dat, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
    AND lo_suppkey = s_suppkey
    AND lo_partkey = p_partkey
    AND lo_orderdate = d_datekey
    AND c_region = 'AMERICA'
    AND s_region = 'AMERICA'
    AND (p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2')
GROUP BY d_year, c_nation
ORDER BY d_year, c_nation;
```

Query 4.2:

```
SELECT d_year, s_nation, p_category, SUM(lo_revenue-lo_supplycost) AS profit
FROM dat, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
    AND lo_suppkey = s_suppkey
    AND lo_partkey = p_partkey
    AND lo_orderdate = d_datekey
    AND c_region = 'AMERICA'
    AND s_region = 'AMERICA'
```

```
    AND (d_year = 1997 OR d_year = 1998)
    AND (p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2')
GROUP BY d_year, s_nation, p_category
ORDER BY d_year, s_nation, p_category;
```

Query 4.3:

```
SELECT d_year, s_city, p_brand1, SUM(lo_revenue-lo_supplycost) AS profit
FROM dat, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
    AND lo_suppkey = s_suppkey
    AND lo_partkey = p_partkey
    AND lo_orderdate = d_datekey
    AND c_region = 'AMERICA'
    AND s_nation = 'UNITED STATES'
    AND (d_year = 1997 OR d_year = 1998)
    AND p_category = 'MFGR#14'
GROUP BY d_year, s_city, p_brand1
ORDER BY d_year, s_city, p_brand1;
```