Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Análise de Desempenho entre PostgreSQL e MonetDB para Ambientes OLAP Utilizando TPC-H

Letícia Torres

CASCAVEL
2017

LETÍCIA TORRES

ANÁLISE DE DESEMPENHO ENTRE POSTGRESQL E MONETDB PARA AMBIENTES OLAP UTILIZANDO TPC-H

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Dr. Clodis Boscarioli

CASCAVEL
2017

LETÍCIA TORRES

ANÁLISE DE DESEMPENHO ENTRE POSTGRESQL E MONETDB PARA AMBIENTES OLAP UTILIZANDO TPC-H

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Dr. Clodis Boscarioli (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE

Gustavo Rezende Krüger (Co-orientador)
Orbit Sistemas

Prof. Dr. Marcio Seiji Oyamada
Colegiado de Ciência da Computação,
UNIOESTE

Bruno Eduardo Soares
Origammi Soluções Digitais

Cascavel, 7 de junho de 2018

DEDICATÓRIA

AGRADECIMENTOS

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

TPC	*Transaction Processing Performance Council*
TPC-H	TPC *Benchmark* H
DSS	*Decision Support Systems*
DW	*Data Warehouse*
OLAP	*On-Line Analytical Processing*
OLTP	*On-Line Transaction Processing*
SGBD	Sistemas Gerenciadores de Banco de Dados
SGBDR	Sistemas Gerenciadores de Banco de Dados Relacional
SGBDC	Sistemas Gerenciadores de Banco de Dados Colunar
SF	*Scale Factor*
PK	*Primary Key*
FK	*Foreign Key*
RF	*Refresh Function*
GB	Gigabyte
TB	Terabyte
CAP	*Consistency, Availability, Partition tolerance*
BASE	*Basically Available, Soft state, Eventual consistency*
ACID	*Atomicity, Consistency, Isolation, Durability*
EM	*Early Materialization*
LM	*Late Materialization*

Lista de Símbolos

SF	Fator de escala do banco de dados
Q_i	Consulta SQL, onde $1 \leqslant i \leqslant 22$
S	Número de sessões de consulta do teste de vazão
s	Uma única sessão, onde $1 \leqslant s \leqslant S$
RF_j	Função de Atualização, onde $1 \leqslant i \leqslant 2$
T_s	Tempo em segundos da execução de todo o processo do teste de vazão

Sumário

Resumo

Data Warehouses se consolidaram nas organizações como tecnologia de apoio à tomada de decisão utilizando aplicações OLAP sobre os dados armazenados. Conforme o volume destes dados aumenta, tornam-se necessárias abordagens mais eficientes para seu processamento. Sistemas Gerenciadores de Bancos de Dados, os SGBD, relacionais são muito utilizados para este propósito, porém novas abordagens denominadas NoSQL têm ganhado destaque, em especial a classe colunar de banco de dados, cada qual com suas vantagens conforme a modelagem do *Data Warehouse*. Modelagens mais normalizadas são tradicionais entre os relacionais, enquanto que modelagens denormalizadas trazem desempenho superior em SGBD colunares. Um estudo comparativo entre os SGBD PostgreSQL e MonetDB utilizando o *benchmark* TPC-H é aqui apresentado, investigando qual é o mais indicado para gerenciar um *Data Warehouse* na recuperação de informações sob as modelagens *snowflake* e *star*. Este estudo foi feito considerando bases de dados de três tamanhos diferentes, 1Gb, 10Gb e 30Gb a fim de simular desde uma quantia menor de registros até um volume maior. Os resultados experimentais confirmam que, tomando apenas o SGBD, o PostgreSQL apresenta desempenho melhor sob ambientes normalizados, enquanto que o MonetDB se destaca nos denormalizados. Como um todo, o MonetDB se destacou tanto para o modelo normalizado quanto para o denormalizado em relação ao PostgreSQL, com ganhos superiores a 100% no ambiente *snowflake* e 600% no *star* considerando dados já em memória, e 80% e 200% sem considerar dados em memória.

Palavras-chave: OLAP, TPC-H, SGBD, NoSQL, MonetDB, PostgreSQL, Data Warehouse, Benchmark

Capítulo 1

Introdução

Vivencia-se atualmente uma economia caracterizada por uma rápida e constante mudança de mercado e oportunidades de negócio, tal que se tornou essencial às empresas a tomada de decisão correta e de forma rápida baseada em alguma decisão de negócio. Essas decisões são tomadas com base na análise de situações passadas e presentes de uma empresa, ou seja, com base nos dados armazenados por ela. Além disso, uma boa decisão também se utiliza da análise de mercado e predições.

Sob a ótica operacional, de acordo com Wremble e Koncilia [1], os dados de uma empresa são persistidos em sistemas de armazenamento de dados que podem ser heterogêneos, autônomos, e geograficamente distribuídos. Estas características diminuem a eficiência no acesso e processamento dos dados. A gestão de uma empresa requer, no entanto, uma visão abrangente de todos os seus aspectos, exigindo acesso eficiente a todos os dados de interesse. Por este motivo, a capacidade de integrar informações de várias fontes de dados é crucial para uma boa decisão de negócios [1].

Para se obter êxito em uma decisão de negócio é trivial (i) recuperar os dados mantidos por uma empresa; e sobretudo (ii) desenvolver um ambiente de análise cujo objetivo deve ser não apenas informar o significado destes dados, mas sim especular cenários sobre eles com questionamentos como *"e se"* ou *"por quê?"* [2]. Segundo Chaudhuri e Dayal [3] dois elementos são essenciais, dadas as condições anteriores, para uma boa decisão de negócio: *Data Warehouses* (DWs), responsáveis pelo armazenamento homogêneo de dados oriundos de sistemas heterogêneos, e recuperação destes dados; e aplicações OLAP (*On-Line Analytical Processing*).

De acordo com a literatura, existe uma série de princípios que devem ser seguidos ao projetar e implementar um ambiente OLAP dentro de um DW, destacando-se a rapidez com que os

dados devem ser recuperados e processados no DW. Este princípio é considerado fundamental na maior parte da literatura referente à construção de ambientes OLAP, a exemplo de Codd; Codd e Salley [2], Kimball e Ross [4] e Wremble e Koncilia [1]. Sendo assim, vários aspectos técnicos devem ser considerados sob diferentes pontos de vista. Políticas de *cache* específicas para um servidor de estruturas utilizadas pela aplicação OLAP para armazenar e recuperar dados em memória e o SGBD (Sistemas Gerenciadores de Banco de Dados) utilizado para o gerenciamento do DW são exemplos de aspectos técnicos que também fazem parte de um ambiente OLAP, e que devem ser considerados.

De acordo com Elmasri e Navathe [5] SGBD são importantes para a manutenção e proteção de um banco de dados por um longo período; e também atuam no processo de recuperação de dados de um DW. Neste contexto, a escolha do SGBD no processo de desenvolvimento de um ambiente de análise para organizações com grande quantidade de dados e que utilizam ferramentas OLAP como auxílio na tomada de decisões é importante.

Duas classes de SGBD podem ser utilizadas para realizar o gerenciamento de um DW: SGBD relacionais tradicionais orientados à linha e uma nova abordagem de SGBD NoSQL que são orientados à coluna, que fornecem um melhor desempenho à recuperação de dados [6]. Dada a importância na escolha do SGBD e as diferentes soluções apresentadas por cada um, torna-se útil o uso de um *benchmark*[1] para a realização de uma análise entre os SGBD. Existe uma organização sem fins lucrativos fundada com o objetivo de definir padrões para avaliar o desempenho de transações e de bancos de dados com o uso de *benchmarks*, o TPC (*Transactional Processing Performance Council*) [8]. Esta organização é responsável pela criação de um *benchmark* voltado para decisões de negócio, o TPC-H [9].

Para refletir a realidade de uma empresa, o TPC-H propõe um ambiente de análise normalizado. Segundo Bax e Souza [10] existe uma discussão sobre a normalização e a denormalização dos dados de um DW. Algumas empresas utilizam um modelo normalizado e têm como justificativa a flexibilidade e a facilidade em situações de manutenção do DW. Por outro lado, existem empresas que utilizam modelos denormalizados e têm como justificativa o ganho de desempenho nas consultas, visto que um modelo denormalizado tende a diminuir o número de tabelas e, por consequência, as operações de junção entre tabelas.

[1]Ferramentas utilizadas para medir e validar o desempenho de alguma tecnologia sob condições de avaliação [7]

O objetivo deste trabalho é a realização de um estudo comparativo entre os SGBD PostgreSQL e MonetDB, o primeiro relacional orientado à linha e o segundo orientado à coluna, como gerenciadores de DWs em ambientes OLAP. A comparação será realizada utilizando-se o *benchmark* TPC-H em sua proposta original, um modelo de banco normalizado; além de uma adaptação para um modelo DW denormalizado. Desta maneira, será possível avaliar os SGBD selecionados de acordo com o modelo do DW, não apenas no contexto original proposto pelo *benchmark*. Para alcançar tal objetivo, é necessária a execução de uma série de tarefas. A primeira parte engloba o desenvolvimento dos dois ambientes de análise, o modelo padrão proposto pelo TPC-H e o modelo denormalizado adaptado do TPC-H. Após, são gerados os dados para popular os SGBD e as consultas para as questões de negócio propostas pelo TPC-H; então é feita a população dos dados gerados nos SGBD. A segunda parte consiste na execução das consultas para cada um dos ambientes de análise de acordo com a metodologia proposta pelo TPC-H, sendo que para o ambiente adaptado as consultas devem ser escritas de acordo com as alterações efetuadas. E por fim a aplicação do cálculo de desempenho proposto pelo TPC-H para os dois ambientes.

A organização do documento segue da seguinte forma:

- **Capítulo 2**: apresenta detalhes sobre recuperação de informações, explicando os conceitos de *Data Warehouse*, ambientes OLAP e a conexão entre ambos.

- **Capítulo 3**: apresenta uma breve descrição da motivação no uso de SGBD como gerenciadores de DWs, bem como a descrição do modelo relacional, suas limitações e uma introdução a NoSQL.

- **Capítulo 4**: apresenta detalhes de como e porquê surgiram os SGBD NoSQL, assim como a descrição do modelo colunar.

- **Capítulo 5**: apresenta detalhes sobre o *benchmark* TPC-H, abrangendo informações sobre o ambiente normalizado proposto pelo próprio *benchmark*; bem como sobre o ambiente denormalizado proposto neste trabalho. É também apresentada a metodologia utilizada para medição e análise do desempenho dos SGBD.

- **Capítulo 6**: é apresentado o estudo comparativo entre os SGBD PostgreSQL e MonetDB,

dividido em cenários conforme o fator de escala, bem como a discussão dos resultados obtidos.

- **Capítulo 7**: são discorridas as conclusões obtidas a partir dos resultados obtidos no experimento, assim como trabalhos futuros.

Capítulo 2

Recuperação de Informação

Um ativo importante em qualquer organização é a informação. Segundo Kimball e Ross [4] essa informação é mantida sob duas formas: sistemas de banco de dados operacionais e *Data Warehouses*. Em sistemas operacionais geralmente os usuários lidam com o mesmo registro e realizam a mesma tarefa exaustivamente sob uma única informação, permanecendo no domínio das transações; enquanto que em um DW pode-se ver o progresso da organização utilizando dados armazenados continuamente, de forma otimizada para a recuperação de dados. Também, são formuladas perguntas com a finalidade de responder a alguma questão de negócio, como *"quantos pedidos foram recebidos pelo fornecedor X no período de tempo Y?"*, ou *"qual foi o impacto no número de vendas ao mudar o formato de envio de A para B?"*. Para responder questões dessa natureza não é viável lidar com dados de forma individual, mas sim recuperar um conjunto de dados a fim de formular uma resposta.

De acordo com Inmon [11], DWs são base de todos os Sistemas de Suporte à Decisão (do inglês *Decision Support Systems*, ou DSS). DSS são tecnologias utilizadas para decisões de negócio e solução de problemas, e incluem componentes que realizam gerenciamento de banco de dados e que permitem uma interação com o usuário de forma a simplificar consultas e geração de relatórios [12]. DWs foram as primeiras ferramentas a surgirem como solução para o suporte à decisão de negócio, integrando dados de diferentes bancos de dados operacionais [11, 4].

De forma geral, um DW é um repositório de dados capaz de fornecer rapidamente informações consistentes e cruciais para a tomada de decisão de uma organização, de tal forma que essa informação possa ser acessada de maneira intuitiva e legível pelo usuário, a fim de combinar diferentes informações entre os dados armazenados [4]. Deve também se adaptar a possíveis mudanças, sejam elas mudanças comerciais, mudanças nos dados ou na tecnologia. Inmon [11]

define um DW como: uma coleção de dados não-volátil, ou seja, que não muda após inserida no *warehouse*; orientado ao assunto principal da organização; integrado e variante no tempo para que seja mantido um histórico a fim de analisar situações passadas. Do ponto de vista estrutural, Wremble e Koncilia [1] definem um DW como uma base de dados homogênea, local e centralizada.

Para analisar os dados de um DW além de implementá-lo é necessário que alguma aplicação leia seu conteúdo e apresente-o de forma gráfica e intuitiva ao analisador. Aplicações OLTP (*On-Line Transactional Processing*) são utilizadas por bancos de dados operacionais e operam transações atômicas e isoladas de forma repetitiva, que correspondem ao dia-a-dia de uma organização [3]. DWs trabalham com suporte à decisão e são intensivos à consultas *ad hoc* complexas, que acessam milhões de registros. Sendo assim, os dados históricos, a taxa de vazão de uma consulta e o tempo de resposta são mais importantes que pequenas transações.

À aplicação aceita por um DW dá-se o nome de OLAP, cujo objetivo, de acordo com Codd; Codd e Salley [2], é identificar tendências, padrões de comportamento e anomalias, bem como relações em dados aparentemente não relacionados. Geralmente as consultas em uma aplicação analítica são longas, levam tempo para executar e estão interessadas em atributos específicos. Os resultados dessas análises servem de base para tomada de decisões de negócio. Ainda, o processo de inserção de dados em um ambiente OLAP é um pouco mais complexo que sistemas OLTP por seguir um processo fim-a-fim conhecido como ETL (extração, transformação e carga) [13]. Portanto, DWs e aplicações OLAP são componentes chave para a construção de um ambiente de análise.

O processo ETL faz parte da arquitetura de um DW. Neste domínio existem diversos componentes que realizam funções específicas a fim de construir um ambiente de *warehouse* desde a obtenção dos dados de fontes externas e sistemas operacionais de bancos de dados, até o acesso a esses dados por meio do DW por alguma consulta analítica definida sob uma aplicação OLAP. Para entender esta arquitetura fim-a-fim, ilustrada na Figura 2.1 adaptada de Kimball e Ross [4], é necessário compreender alguns componentes e conceitos que formam um DW [4]:

- **Sistemas de Fonte Operacional**: possuem detalhes sobre as transações do negócio, correspondendo a ambientes OLTP. Engloba os dados que irão estruturar as informações do DW, portanto se encontram externos ao *warehouse*. Podem ser tanto sistemas de banco

de dados ou alguma outra fonte de dados, como um documento no formato XLS, CSV, TXT; e sistemas CRM.

- ***Staging Area***: compreende tanto uma área de armazenamento temporária quanto um conjunto de processos denominado ETL (*extract-transformation-load*). De forma geral é uma área a qual os usuários não têm acesso, onde os dados são traduzidos para algo que possa ser enviado de maneira compatível ao *warehouse* e não se trabalha diretamente sobre os dados transacionais. Quanto aos processos ETL, a fase de Extração (*Extraction*) consiste na leitura da fonte de dados, transferindo o conteúdo necessário para a *staging area*; após essa extração pode ser necessário realizar uma "limpeza"nos dados; unir dados de diferentes fontes; tratar duplicatas e atribuir chaves do *warehouse*. A isto dá-se o nome de Transformação (*Transformation*). A última fase, fase de Carregamento (*Load*), é responsável por carregar, ou popular, os dados na área de estruturação de dados do DW.

- **Estruturação de Dados**: trata de como os dados serão organizados, armazenados e disponibilizados para consultas de usuários, relatórios e outras aplicações. No que tange à comunidade empresarial, a fase de apresentação de dados *é* o DW em si, pois corresponde ao que pode ser acessado via ferramentas de acesso a dados. A etapa de estruturação é comumente definida como sendo um conjunto de *data marts*. *Data marts* são subconjuntos do total de informações de um DW, cada qual representando os dados de um determinado assunto, departamento, ou processo de negócio. Nesta fase é definida a modelagem conceitual do ambiente de análise do DW.

- **Ferramentas de Acesso aos Dados**: são formas de aplicar uma consulta, dentro de aplicações OLAP, aos dados organizados na fase de estruturação. Pode ser uma consulta *ad hoc* ou algo mais complexo, como consultas aplicadas à mineração de dados.

Existe uma discussão acerca da modelagem conceitual da área de apresentação de dados de um ambiente de análise nos DWs [14]. Segundo Sen e Sinha [14] as duas técnicas de modelagem mais utilizadas são a Entidade-Relacional (ER) e a Dimensional. A primeira segue o padrão de modelagem para ambientes OLTP, que traduz a modelagem ER para um esquema relacional em seguida normalizando-o geralmente até a Terceira Forma Normal (3NF) [4], normalmente utilizadas por bancos relacionais. O modelo Dimensional, ou multidimensional, por

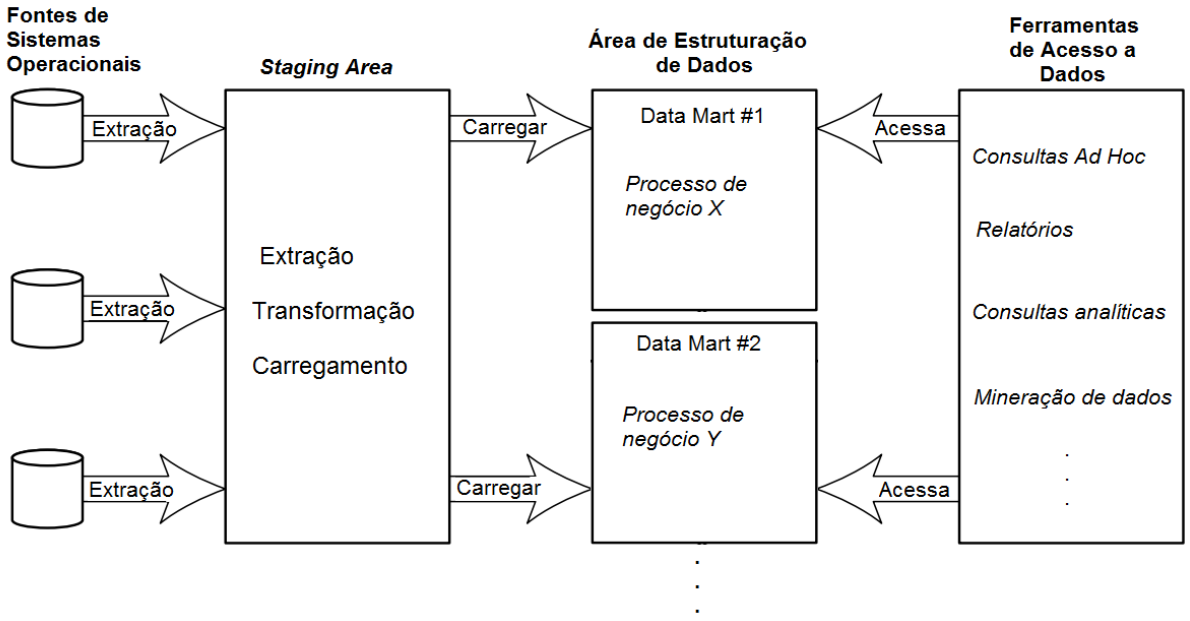


Figura 2.1: Arquitetura de um *Data Warehouse*.

sua vez, evita atingir o mesmo nível de normalização da modelagem ER, e faz analogia a um cubo para representação de dados, uma vez que uma informação pode ser vista através de n dimensões.

O modelo dimensional é composto por tabelas denominadas Tabelas Fato e Tabelas Dimensão [4], e é conhecido comumente como modelo *star join*, ou apenas *star* [14], pelo seu formato lembrar o de uma estrela. A Tabela Fato é a principal tabela do modelo Dimensional, contemplando atributos responsáveis por determinar as regras e métricas de negócio, ou um fato. Em sua maioria são atributos numéricos, relacionados a quantidade, e aditivos, visto que uma consulta em um DW pode retornar até milhares de tuplas, tornando interessante o conhecimento de informações como o total de um atributo dada alguma questão de negócio. Atributos textuais não são definidos como um fato, na maioria das vezes descrevem algo e devem estar inseridos em Tabelas Dimensão pois há maior chance de estarem relacionados com os atributos dimensão. Outro ponto a se ater é a importância de se evitar incluir dados com zeros que não representam dados úteis, ou representando nada, pois a inclusão de dados sem significado sobrecarregariam o DW sem motivo.

As Tabelas Fato são auxiliadas pelas Tabelas Dimensão no que concerne à descrição textual das questões de negócio. É comum essas tabelas terem de 50 a 100 atributos, ou mais, pois a

intenção das dimensões é descrever as regras de negócio. É nestas tabelas que são realizados os filtros de consulta. Por filtros entende-se agrupamentos, padrões e ordenações por exemplo. De forma a exemplificar, se fosse desejado buscar as vendas por nome de fornecedor em um determinado intervalo de data, os atributos "nome de fornecedor"e "data"deveriam estar armazenados em tabelas dimensão. Segundo Kimball [4], quanto mais bem descritos os atributos dimensão, melhor o DW é, tornando a qualidade do *warehouse* dependente das entidades de dimensão.

Todas as Tabelas Fato tem duas ou mais chaves relacionando-as às Tabelas Dimensão, como mostra o exemplo da Figura 2.2, onde a tabela *Vendas* corresponde à uma Tabela Fato e as demais à Tabelas Dimensão. Note que esta figura também faz referência a um modelo *star*.

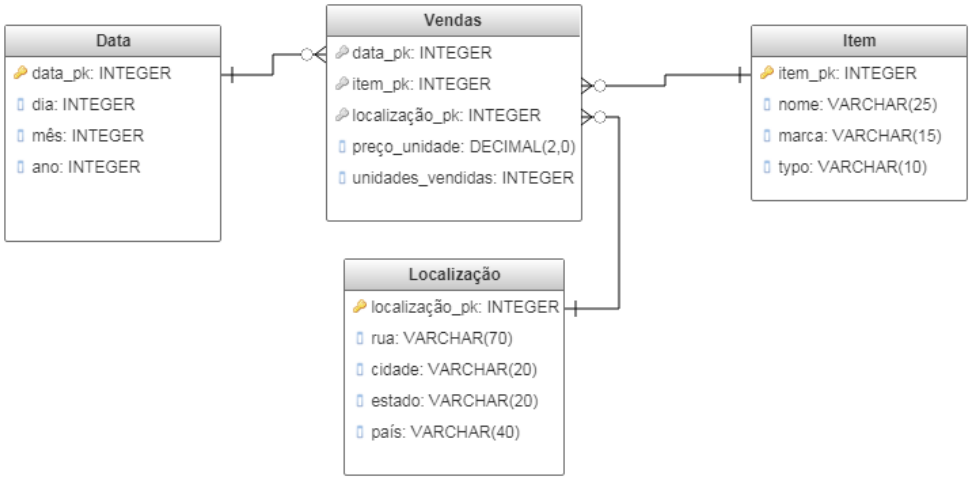


Figura 2.2: Exemplo de esquema *star* com tabelas Fato e Dimensão

Mesmo que a modelagem Dimensional não atinja a normalização 3NF, modelos *star* podem ser trabalhados de forma a oferecer suporte à hierarquia de atributos às Tabelas Dimensão, permitindo que estas tenham Tabelas "Subdimensão". A esse refinamento se dá o nome de *snowflake* [5]. Embora tenham uma estrutura mais simplificada, segundo Levene e Loizou [15] a escolha do uso de esquemas *snowflake* se dá por serem um esquema intuitivo, de fácil entendimento, passíveis à otimização de consultas, e de fácil extensão – uma vez que pode-se adicionar atributos às tabelas sem interferir em programas já existentes. Uma possível adaptação de um modelo *star* para *snowflake* é como mostrado na Figura 2.3, no qual foi adaptado o modelo da Figura 2.2, adicionando a Tabela Subdimensão Cidade.

Para que possa ser construído um DW e aplicadas as modelagens acima descritas, é necessá-

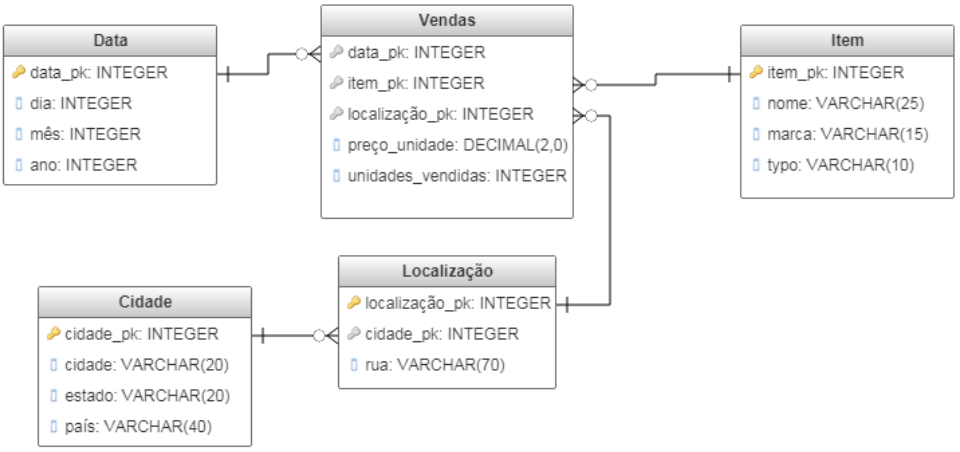


Figura 2.3: Exemplo de esquema *snowflake* adaptado da Figura 2.2

rio alguma ferramenta que possa fazer o gerenciamento dele. De acordo com Elmasri e Navathe [5] um banco de dados pode ser gerenciado por sistemas que facilitem este processo de gerenciamento no banco de dados. Estes sistemas são conhecidos como Sistemas Gerenciadores de Banco de Dados, ou SGBD.

Capítulo 3

Sistemas Gerenciadores de Banco de Dados

Antes de se ter um sistema que visava gerenciamento de um banco de dados, era utilizado para persistência de dados o sistema de arquivos. Apesar de simples, esta abordagem apresentava alguns problemas por não apresentar suporte à redundância de informações; não garantir integridade de dados; falta de segurança; e o acesso e gerenciamento dos dados dependia de programas e aplicativos, fazendo com que seja necessário criar um novo aplicativo, ou adaptá-lo, a cada requisição de dados diferente. Outro problema crítico é não se ter informação de relacionamento entre arquivos diferentes.

Como forma de manter o gerenciamento de dados independente de aplicações e programas bem como solucionar os demais falhas do sistema de arquivos foram criados os Sistemas Gerenciadores de Bancos de Dados, os SGBD. De acordo com Elmasri e Navathe [5], SGBD são uma coleção de programas para criação e manutenção de um banco de dados, que facilita a definição, construção, manipulação e compartilhamento de dados entre usuários e aplicações.

Dentre as vantagens que os SGBD trouxeram em detrimento ao sistema de arquivos estão:

- **Controle de redundância**, para que não seja permitido duplicação de dados, pois isto causaria desperdício na capacidade de armazenamento.
- **Restrição de acesso aos usuários**, pois não será permitida manipulação do banco a todos os usuários, ou funcionários de uma empresa por exemplo.
- **Execução de consultas eficiente** através do uso de índices, normalmente implementadas utilizando hash ou árvores, para que o acesso ao disco seja mais rápido.

- **Restrições de integridade**, a fim de garantir que (i) dados não sejam inseridos de forma inconsistente de acordo com o tipo de atributo definido; (ii) as relações entre entidades sejam efetuadas e (iii) restrições de chave sejam mantidas.
- **Persistência de dados**, para garantir que os dados serão inseridos e de fato armazenados no modelo.
- ***Backup*** de dados periodicamente para evitar problemas caso aconteça alguma perda no banco e posterior **restauração**, para recuperar uma imagem do último backup feito no banco.

Existem várias classes de SGBD, conforme sua estruturação e a forma como manipulam os dados, entre as mais conhecidas estão o modelo relacional, objeto-relacional, orientado a objetos e a abordagem mais recente NoSQL.

3.1 Modelo de SGBD Relacional

O modelo mais utilizado de SGBD é o relacional, ou SGBDR (SGBD Relacional). Ele foi conceituado por Codd [16] em 1970 em um artigo no qual são expostas as vantagens de um modelo relacional em detrimento a um modelo de redes.

Um modelo relacional define através de um conjunto de tabelas entidades que representam objetos do mundo real, cada qual com seus atributos. Esse conjunto de atributos é denominado registro do banco de dados, representando uma tupla, ou linha, na tabela. Assim como no mundo real, objetos devem estar também relacionados, e para tal SGBDR utilizam-se de chaves.

Em um SGBDR além de definir relacionamentos, as chaves também garantem integridade entre os dados. Existem dois tipos de chaves, uma que garante a unicidade de um atributo, ou seja, assegura que não haverão registros repetidos no banco tomando como referência aquele valor de chave, chamada de chave primária (do inglês *Primary Key*, ou PK). A outra chave garante integridade no relacionamento entre duas entidades referenciando uma tabela na outra, que é a chave estrangeira (do inglês *Foreign Key*, ou FK).

Tomando como exemplo uma empresa fictícia, são criados dois objetos reais que precisam ser representados sob a forma relacional, *funcionário* e *projeto*. Nesta empresa funcionários possuem *nome, CPF, sexo, salário,* um *supervisor* e trabalham em um ou mais *projetos*. Esses

projetos também possuem atributos: *nome, código* e o número do *departamento* no qual foram criados. Apenas com estas informações duas tabelas já são definidas no banco, *funcionário* e *projeto*, bem como seus atributos, como ilustra a Figura 3.1.

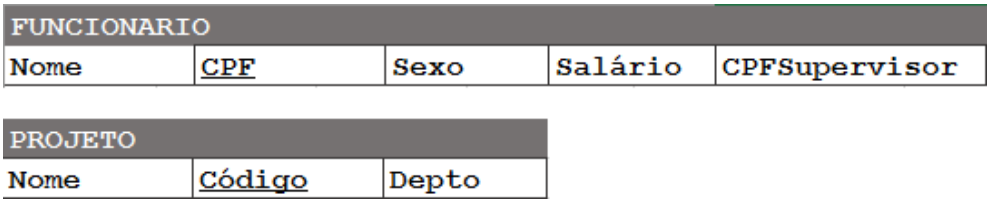


Figura 3.1: Atributos das entidades *funcionário* e *projeto*

Como descrito, um funcionário trabalha em um ou mais projetos e ele precisa ainda registrar o número de horas trabalhadas nestes projetos. Desta forma é preciso de alguma forma relacionar funcionário com os projetos. Neste exemplo cabe a criação de uma nova entidade responsável unicamente por relacionar estes dois objetos e ainda armazenar o número de horas – veja que o número de horas não cabe na tabela de funcionários nem na de projetos.

É nesta relação que são explorados os conceitos de chaves em um banco de dados. Para que a entidade que relaciona funcionário e projeto tenha informações de que funcionário trabalha em qual projeto é necessário acessar um número, ou código, definido para diferenciar todos os funcionários da empresa bem como o código do projeto e relacioná-los. Para o funcionário uma PK válida é o número do CPF e para o projeto o próprio atributo código. Essa relação pode ser nomeada unindo as entidades que relaciona, nesse caso *funcionário_projeto*, ou quando faz sentido pode ser nomeada de acordo com a função que realiza, neste caso como um funcionário trabalha em um projeto a relação pode ser *trabalha_em*.

O esquema que representa estas relações é como mostrado na Figura 3.2. Nela podemos ver as chaves primárias, em destaque, nas relações *funcionário* e *projeto*, e seus valores como chave estrangeira na relação *trabalha_em*.

Modelos de bancos de dados prezam, sempre que possível, pela consistência de dados, principalmente se tomar operações transacionais como transações bancárias, que necessitam de um alto grau de consistência após operações de inserção e atualização, para que não haja perda de valores ou que esses ainda façam sentido no sistema. Por essa razão SGBDR seguem o conceito ACID (do inglês *Atomicity, Consistency, Isolation, Durability*), o qual garante Atomicidade, Consistência, Isolamento e Durabilidade, ou Persistência, de dados.

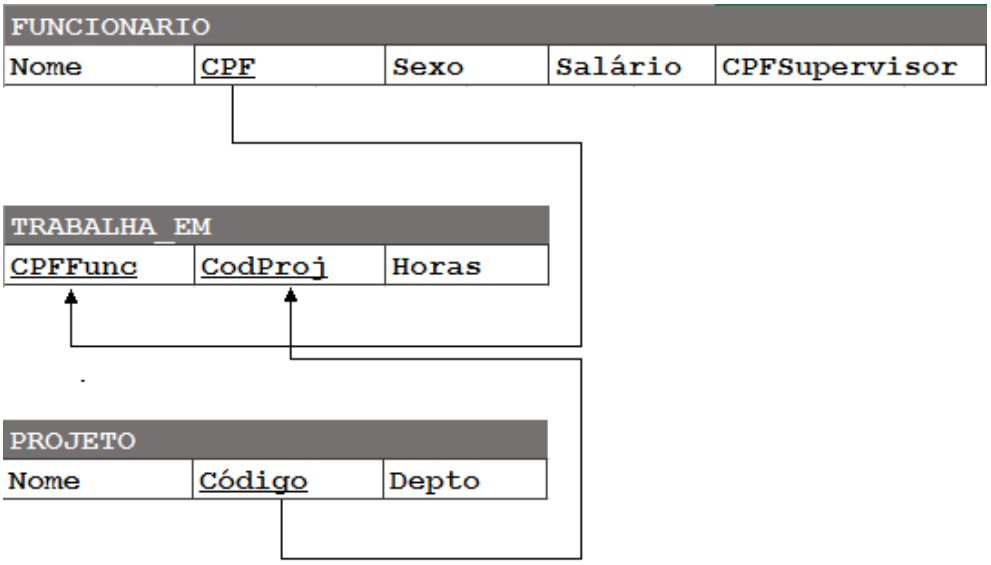


Figura 3.2: Relacionamento entre *funcionário* e *projeto*

Modelos relacionais também seguem um alto grau de normalização nos dados a fim de evitar redundância e confusão entre atributos que podem ser desmembrados em outras entidades. Essa normalização é realizada normalmente até a Terceira Forma Normal, e o esquema final conterá um número maior de tabelas que o original devido ao desmembramento de atributos. Essa quantidade maior de tabelas acarreta no aumento de junções necessárias para recuperar atributos entre entidades relacionadas, ou seja, no aumento do acesso a entidades e no processamento de tuplas.

Além de diminuir o desempenho na recuperação de dados, um esquema normalizado causa um forte acoplamento entre tabelas, o que tornaria inviável o particionamento dos dados do banco em *n* servidores como forma de escalar estes dados conforme seu volume aumenta (mais detalhes no Capítulo 4), como é o caso de um ambiente analítico. Uma solução trivial para este problema é a denormalização de dados, porém para um SGBDR essa solução pode não ser viável, visto que sua implementação se dá seguindo a modelagem relacional.

Exemplos de SGBDR são PostgreSQL [17], MySQL [18], Microsoft SQL Server [19], SQ-Lite [20], MariaDB [21] e Oracle [22].

3.2 Modelo de SGBD NoSQL

Bancos com o objetivo de suprir problemas apresentados por SGBDR começaram a emergir entre 2004 e 2007 visando alta escalabilidade, recuperação e armazenamento de grandes volumes de dados de forma eficiente, e ao mesmo tempo diminuir custos com hardware [23]. Esses bancos levaram o nome de NoSQL, porém isso não significa que não utilizam SQL para manipulação de dados.

Detalhes sobre bancos de dados NoSQL são encontrados no Capítulo 4, no qual também é descrito o modelo colunar, utilizado por satisfazer os requerimentos de um ambiente de DW.

Capítulo 4

Banco de Dados NoSQL

O desenvolvimento de novas aplicações e surgimento de novas soluções para sistemas gerou crescimento no volume de dados de maneira acelerada. Com isso cresce também o número de usuários, necessitando portanto escalar o banco de dados. Existem duas soluções para escalar um sistema: o escalonamento vertical, que consiste em um *upgrade* do servidor no qual o banco está hospedado; e o horizontal, aumentando o número de servidores e distribuindo o banco [24, 25].

Existem algumas desvantagens no *upgrade* de um sistema. O banco de dados pode superar a capacidade da melhor configuração disponível no mercado, e ainda é caro por requerer a aquisição de uma configuração melhor. Mesmo considerado mais complexo, o escalonamento horizontal é mais viável. Entre as soluções apresentadas no particionamento horizontal em banco de dados estão o particionamento funcional e o *sharding* [24].

O particionamento funcional consiste em fragmentar os dados de acordo com a forma como são utilizados dado um contexto. Um esquema com quatro entidades, *usuários*, *produtos*, *clientes* e *endereço* pode ser distribuído em quatro servidores, um para cada entidade. Entidades que são utilizadas somente para leitura podem ser separadas de entidades onde dados são escritos, caracterizando outro exemplo de particionamento funcional. O problema com esta estratégia está no acoplamento entre entidades: se duas ou mais entidades estiverem relacionadas elas deverão estar no mesmo servidor, caso contrário não será possível atribuir as restrições de chave àquele relacionamento [24]. Tomando o exemplo acima com as quatro entidades, supondo que *cliente* e *endereço* estejam relacionadas estas devem ser armazenados no mesmo servidor.

A segunda estratégia, o *sharding*, consiste em dividir os dados do banco utilizando algum critério de separação que não seja limitado à funcionalidade das entidades. Soluções como par-

ticionamento com base em *hash* ou listas podem ser aplicadas. Considerando dez servidores e uma chave primária auto-incremental, a função de *hash* pode tomar o módulo da chave primária pelo total de servidores como critério de seleção da partição na qual o dado será inserido. Utilizando uma lista é possível definir valores para os servidores e distribuir os dados de acordo com esses valores. Por exemplo, as linguagens C++, Java e C# poderiam ser inseridas em uma partição destinada à linguagens orientadas a objeto.

O *sharding* enfrenta problemas com junções entre dados, visto que os dados devem ser recuperados de diferentes partições e a maioria dos SGBDR não oferece suporte a chaves estrangeiras sob diferentes servidores, restando ao desenvolvedor tratar isso no código da aplicação [24]. Uma solução para tais problemas é a denormalização de dados para que o número de junções diminua ou, no melhor dos casos e quando possível, seja zero. Contudo, isso é um problema para modelos relacionais, visto que trabalham sobre uma modelagem normalizada de dados. Além disso e de não se adaptarem à escalabilidade horizontal, são complexos e o fato de trazerem todas as informações de uma entidade sob a forma de tuplas causa lentidão em um ambiente analítico na recuperação de dados, cujas consultas percorrem o banco visando atributos específicos, processando somente o necessário.

Outro problema crítico ao não utilizar a escalabilidade horizontal está na disponibilidade dos dados. Enquanto um banco se apoiar na escalabilidade vertical, qualquer queda no servidor acarretará na queda total no sistema de armazenamento, ao passo que quando se trabalha com servidores em paralelo a queda em uma máquina não trará prejuízos no conjunto todo. Assim, soluções como melhorar o hardware do servidor continuam sendo desvantajosas. Além da escalabilidade, disponibilidade de dados foi um dos argumentos utilizados por um dos engenheiros da rede social Twitter ao migrar do MySQL para o Cassandra, um SGBD NoSQL. Em 2008 a rede ficou fora do ar por 84h [26].

Com o intuito de suprir tais problemas de escalabilidade, disponibilidade e recuperação rápida de dados, entre 2004 e 2007 os SGBD NoSQL começaram a ganhar destaque com o surgimento das bases de dados BigTable da Google [27], e Dynamo da Amazon [28]. O termo NoSQL, embora a princípio pareça indicar total independência de SQL, significa *Not Only SQL*, "Não Apenas SQL". Também, ele não descreve um único tipo de SGBD, mas sim uma classe de modelos, cada qual com suas propriedades. As mais conhecidas são quatro:

- **Orientado a Grafos**, que se utiliza da Teoria dos Grafos para estruturar seus dados. Um exemplo desta classe é o Neo4j [29].

- **Orientado a Chave-Valor**, que armazena os dados de forma similar a uma tabela *hash*, com uma chave referenciando um valor, ou tipo de dado. Exemplos são o Project Voldemort [30], DyanmoDB [31], o Riak [32] e o Redis [33].

- **Orientado a Documento**, uma versão melhorada do Chave-Valor, no qual os valores são armazenados como documentos através de estruturas complexas como JSON e XML. Exemplos são o MongoDB [34] e o CouchDB [35].

- **Orientado a Colunas**, ou modelo colunar, que utiliza tabelas como armazenamento de entidades, porém não agrupa os dados sob forma de tuplas, e sim colunas. Exemplos são o MonetDB [36], C-Store [37], BigTable [38] e Cassandra [39].

De maneira geral, as vantagens no uso de um SGBD NoSQL estão no rápido processamento de um grande volume de dados, flexibilidade para expansão e baixo custo de escalabilidade. Devido ao vasto número de SGBD dentro de cada classe de NoSQL, esses bancos também podem ser classificados de acordo com o Teorema CAP (do inglês *Consistency, Availability, Partition tolerance*). Segundo Eric Brewer [40, 41], um sistema distribuído não é capaz de conciliar consistência, disponibilidade e tolerância a partição de dados simultaneamente, tendo que optar por apenas dois destes. SGBDR prezam por disponibilidade e consistência de dados seguindo o ACID, porém a preocupação com consistência pode tornar a manipulação de dados lenta. Visto que o movimento NoSQL surgiu com a intenção de melhorar a escalabilidade e disponibilidade de dados, e tornar a manipulação destes mais rápida, a maioria deles trabalha com os atributos de disponibilidade e tolerância à partição. Essa configuração assume um conceito diferente do ACID para bancos NoSQL, e Pritchett [24] propôs um teorema diferente deste, mais otimista, onde a consistência é relaxada, o Teorema BASE (*Basically Available, Soft State, Eventual Consistency*).

O Teorema BASE assume que a consistência em um banco de dados está em estado de fluxo, ao contrário do ACID que força a consistência a cada operação, sendo este considerado pelo autor um método pessimista. Neste cenário a disponibilidade é garantida devido à tolerância a partição, fazendo com que a falha de um servidor não cesse o funcionamento de todo o sistema

– por exemplo, caso uma partição falhe em um sistema rodando sobre dez servidores, apenas 10% dos dados estarão indisponíveis e somente os usuários daquela partição serão afetados.

4.1 Modelo de SGBD Colunar

Dentre as categorias de bancos de dados NoSQL a que melhor se adequa aos propósitos analíticos é a colunar. Sistemas colunares armazenam seus dados em colunas que representam atributos das entidades, fazendo com que apenas os atributos necessários sejam lidos [42], o que diminui o tempo de acesso ao disco [43, 44]. Cada uma destas colunas pode armazenar seus valores utilizando o par *chave, valor* [45, 42], sendo esta uma das formas de implementação de um sistema colunar (descrita na Seção 4.2 deste capítulo). A Figura 4.2 ilustra de forma simples, embora apenas visual, a diferença principal entre um armazenamento utilizando linhas, conforme a Figura 4.2(a), e outro utilizando colunas, Figura 4.2(b), a partir dos registros da Figura 4.1, baseada na entidade *funcionário* apresentada no Capítulo 3.

Nome	CPF	Sexo	Salario	CPFSupervisor
Aliyah	61141093090	F	2300	45074294711
Lana	89212829040	F	989	38914097102
Brenno	63791658352	M	900	54233286830
Ali	1439498105	M	1923	6644601006
Jett	80360617832	M	3452	6958676360
Moly	56117478178	F	3400	34807186043

Figura 4.1: Exemplos de registros para a entidade *funcionário* apresentada no Capítulo 3

Tupla#1	Aliyah	61141093090	F	2300	45074294711
Tupla#2	Lana	89212829040	F	989	38914097102
Tupla#3	Brenno	63791658352	M	900	54233286830

(a)

Col#1	Aliyah	Lana	Brenno
Col#2	61141093090	89212829040	63791658352
Col#3	F	F	M
Col#4	2300	989	900
Col#5	45074294711	38914097102	54233286830

(b)

Figura 4.2: Diferença visual entre registros armazenados em um banco relacional e colunar

Matei [43] cita algumas vantagens de sistemas colunares:

- **Melhor desempenho**, pela forma com que os dados são armazenados e o uso de índices visando o armazenamento ao invés de localização de registros, resultando em menos

operações de entrada e saída.

- **Rápidas operações de agrupamento**, bastante utilizadas em ambientes OLAP, visto que valores de um mesmo atributo são armazenados consecutivamente. Bem como operações matemáticas, como recuperar o maior ou menor valor, soma e média.

- **Alta compressão de dados** pode ser alcançada, devido às chances de se ter valores repetidos para um mesmo atributo serem maiores.

Existem diferentes abordagens para se implementar um SGBD colunar. Essas abordagens levam em conta mudanças na codificação de um banco, apenas na modelagem dele, bem como se será possível atingir níveis altos de compressão. A primeira técnica consiste no **particionamento vertical** dos dados, a segunda em **modificar a camada de armazenamento** e a terceira corresponde a junção de ambas.

4.2 Implementação do Sistema Colunar

Antes de detalhar os métodos de implementação de um banco de dados colunar, é fundamental discutir sobre a reconstrução de tuplas nesse tipo de SGBD. Segundo Abadi et al. [46], a parte lógica e visual de um banco de dados colunar se apresenta da mesma forma que um relacional. Isso faz com que a maioria dos SGBD colunares ofereçam uma interface relacional de comunicação, compatível com os padrões.

Como os atributos de um modelo colunar acabam ficando separados em disco um do outro, eles precisam ser unidos novamente em tuplas para que o resultado de uma consulta seja exibido. Existem duas técnicas para realizar essa reconstrução, ou materialização: *Early Materialization* (EM) e *Late Materialization* (LM) [46, 47].

A técnica de *Early Materialization*, melhor traduzida como materialização prévia nesse contexto, é a política adotada por banco de dados relacionais, e consiste em adicionar as colunas à tupla conforme a coluna é requisitada na consulta. Essa técnica não leva em conta o predicado da consulta, isto é, considere a consulta a seguir:

```
SELECT nome FROM funcionario
WHERE salario >= 1000 AND sexo='F'
```

A EM irá processar essa consulta e construir uma tupla com os atributos *nome, salário* e *sexo*, com todos os registros armazenados em banco. Considere que os registros são como mostrados na Figura 4.1, o construtor de tuplas retornará os registros conforme a Figura 4.3. Esse método só analisa o predicado após a construção das tuplas, o que não acontece na LM.

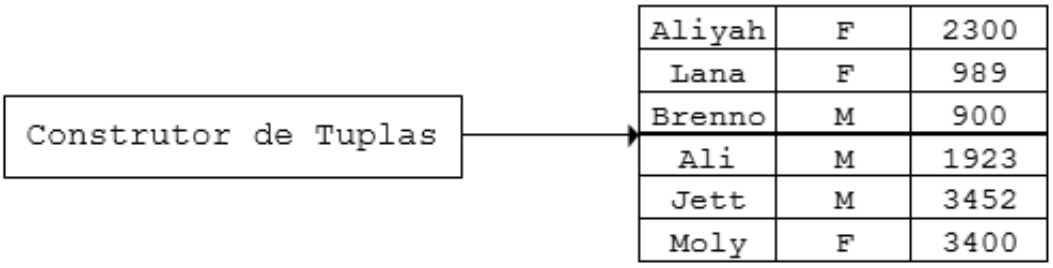


Figura 4.3: Tuplas reconstruídas a partir da *Early Materialization*

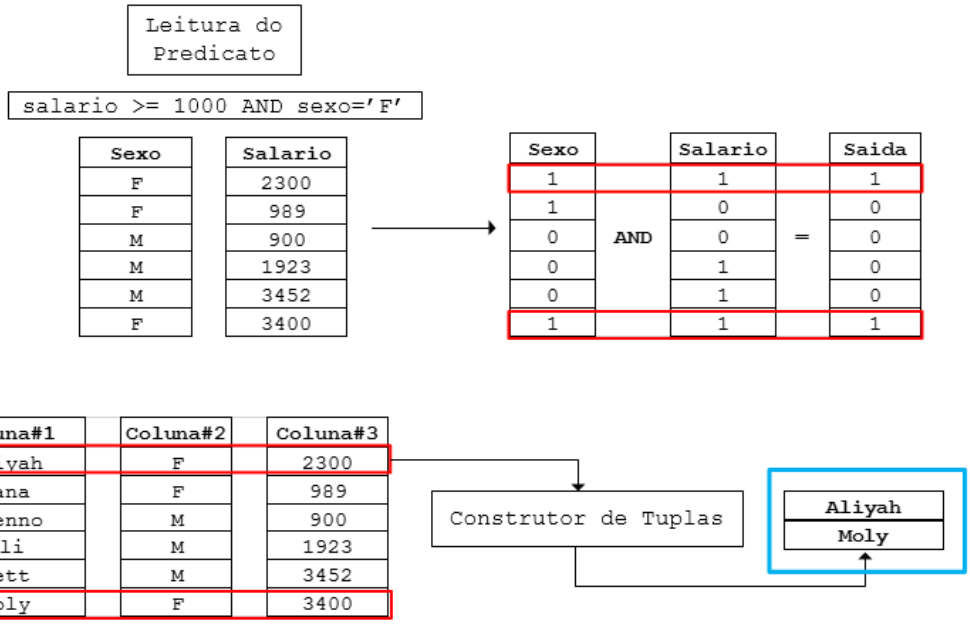


Figura 4.4: Tuplas reconstruídas a partir da *Late Materialization*

Na LM, ou materialização tardia, é analisado primeiro o predicado e verificado quais valores atendem a esse predicado em cada coluna. No caso da SQL acima primeiro seria analisado o predicado de seleção, nesse caso `WHERE salario >= 1000 AND sexo='F'`, e então através do operador lógico AND seria possível retornar a posição dos atributos que satisfazem essa seleção, para então construir a tupla a partir do predicado de projeção somente com o atributo *nome*. Em suma, apenas após ter a posição de cada atributo é que a tupla é construída,

descartando a reconstrução de tuplas que seriam posteriormente descartadas. A Figura 4.4 apresenta o resultado após a construção das tuplas na LM.

4.2.1 Particionamento Vertical

Considerada a técnica mais simples para implementar um banco colunar, esta técnica realiza a partição dos dados de forma que cada atributo de uma entidade seja armazenado em uma tabela de duas colunas contendo o par <chave/índice, valor do atributo> [42].

A Figura 4.5 mostra que é possível desmembrar o esquema relacional da Figura 4.1 em colunas de atributos – foram desmembrados os atributos *nome, sexo* e *salario*. No momento em que uma consulta analítica é realizada, não serão processados dados além dos de fato requeridos e nenhum dado é perdido pois ainda haverá referência entre os atributos através das chaves.

Coluna#1	
Índice	Nome
1	Aliyah
2	Lana
3	Brenno
4	Ali
5	Jett
6	Moly

Coluna#2	
Índice	Sexo
1	F
2	F
3	M
4	M
5	M
6	F

Coluna#3	
Índice	Salario
1	2300
2	989
3	900
4	1923
5	3452
6	3400

Figura 4.5: Implementação do modelo colunar utilizando particionamento vertical

Essa abordagem é a mais simples de se implementar, pois nenhuma mudança é feita na codificação do banco de dados, sendo que a única alteração necessária está a nível da aplicação responsável por executar as consultas, mudando a lógica de acesso aos dados. Isso torna prático para empresas a mudança entre a estratégia relacional e a colunar, pois torna possível o uso de um mesmo SGBD como DW.

Ao mesmo tempo que essa implementação traz praticidade na migração de dados, ela não difere muito da abordagem relacional no que concerne ao desempenho. Em Abadi [47] é mostrado que adaptar dados de um esquema relacional utilizando particionamento vertical não trouxe melhoras no desempenho das consultas. Foi verificado que o custo de junções para unir as colunas é alto conforme a quantidade de atributos selecionados aumenta e não foi possível usufruir eficientemente da compressão de dados.

4.2.2 Modificação na Camada de Armazenamento

A segunda abordagem para implementar um sistema colunar consiste em já modificar o armazenamento do banco. Essa técnica não altera a parte lógica da modelagem do banco, mas a camada física alterando o armazenamento de linha por linha para coluna por coluna. Nessa implementação as chaves não precisam ser repetidas como na abordagem anterior, na qual a primeira coluna de cada tabela consistia no valor de chave da cada atributo. O *i-ésimo* valor de atributo de uma coluna estará associado ao *i-ésimo* atributo das demais colunas e não é desperdiçado espaço em disco com as chaves.

Neste método a reconstrução de tuplas é realizada antes da execução da consulta de fato. Como apenas um determinado número de atributos precisa ser acessado, estes são combinados seguindo a lógica que o *i-ésimo* atributo de uma coluna está relacionado com o *i-ésimo* atributo das demais. Aqui tem-se uma vantagem no modelo orientado a linhas, por este já armazenar as sob o formato de linhas e não precisar armazenar os dados em ordem.

4.2.3 Modificação na Camada de Armazenamento e Particionamento Vertical

A última, e melhor, abordagem consiste em modificar a camada de armazenamento do banco de dados e a aplicação que processa as consultas [48, 49], assim o processador pode manter os dados sob a forma de colunas sem precisar unir as colunas e construir tuplas. Uma vantagem óbvia desta técnica é a eliminação da reconstrução de tuplas e com isso menos dados precisam ser previamente movimentados, diminuindo o tempo de processamento.

4.3 Compressão de Dados

Uma característica marcante de um SGBD colunar é a alta possibilidade de compressão de dados. Existem diversas formas de comprimir os dados, porém deve-se ater ao fato de que o tempo de compressão e descompressão não deve afetar significativamente o processamento dos dados no SGBD. Para tal, uma solução é utilizar métodos que consigam operar sob os dados ainda comprimidos – ainda nesta seção são apresentados alguns dos algoritmos utilizados para compressão em banco de dados.

Como exemplo, apenas ilustrativo, de como os dados em um sistema colunar podem ser

comprimidos de forma mais eficiente que em um relacional, considere a entidade *usuário* a seguir, com os atributos *id, nome, idade* e *sexo*. A primeira coluna corresponde aos índices utilizados no banco.

```
1: 0, Bruno, 50, M
2: 10, João, 23, M
3: 20, Maria, 30, F
4: 90, Ana, 23, F
5: 78, Gabriel, 23, M
```

Um banco relacional armazena estes atributos da forma como foram apresentados acima. Considerando o armazenamento colunar, os mesmos dados seriam armazenados da seguinte forma:

```
1: 0, 2: 10, 3: 20, 4: 90, 5: 78
1: Bruno, 2: João, 3: Maria, 4: Ana, 5: Gabriel
1: 50, 2: 23, 3: 30, 4: 23, 5: 23
1: M, 2: M, 3: F, 4: F, 5: M
```

Note que fica mais claro que como os atributos são armazenados em uma coluna com chave e valor existirão mais chances de repetição de valores em uma mesma coluna. Os atributos *idade* e *sexo* se repetem para alguns registros, possibilitando assim compressão destes dados. Comprimindo estas repetições o nosso modelo é armazenado da forma:

```
1: 0, 2: 10, 3: 20, 4: 90, 5: 78
1: Bruno, 2: João, 3: Maria, 4: Ana, 5: Gabriel
1: 50, 2, 4, 5: 23, 3: 30
1, 2, 5: M, 3, 4: F
```

Westmann et al. [50] consideram que os métodos de compressão em um banco de dados devem ser capazes de se aplicar no banco como um todo, bem como em uma tupla ou a cada atributo de tupla, e serem rápidos em processamento quando há casos em que seja necessário realizar descompressão.

O primeiro método consiste em comprimir a representação de valores inteiros, baseado no algoritmo *null supression* [50, 51]. A ideia é descartar zeros na representação de um inteiro de quatro bytes, por exemplo, ao invés de se representar o número 10 da forma '00000000000000000000000000001010', seriam utilizados apenas quatro bits, '1010'. É necessário entretanto guardar a informação de quantos bits são utilizados para armazenar um número e em alguns casos também é preciso armazenar um bit a mais para o sinal do inteiro.

Esse algoritmo pode ser utilizado para comprimir o tamanho de strings no banco quando estas são representadas pelo tipo VARCHAR, visto que para esse tipo de dado o tamanho também é armazenado devido a possibilidade deste variar.

Um método bastante empregado é a Codificação por Dicionário. Nele atributos que possuem um padrão fixo, por exemplo, os números primos de 3 a 19 (3, 5, 7, 11, 13, 17, 19), podem ser representados por um dicionário de 3 bits. Este algoritmo pode ser utilizado para compressão de valores NULL, considerando-o como um possível valor no padrão.

Abadi, Madden e Ferreira [52] utilizam a compressão por dicionário e uma variação do algoritmo *null supression* para implementar o banco de dados C-Store e ainda apresentam dois outros métodos. O primeiro é o *run-length*, que pode ser bem aproveitado em bancos colunares quando atributos são repetidos sequencialmente e possuem pouca variação. Ele consiste em contar o número de vezes no qual um valor é repetido utilizando a tripla (valor, posição inicial, tamanho).

O último método apresentado pelos autores é a Codificação por Vetor de Bit, bastante útil quando uma coluna possui número limitado de possibilidades, como linguagens de programação dentro de um dado paradigma ou estados do Brasil. Uma string de bits representa a ocorrência do valor de atributo, onde '1' representa a ocorrência naquela posição e '0' a não ocorrência. Como exemplo, uma coluna com os valores M M F M F F M podem ser representados pela cadeia de bits 1 1 0 1 0 0 1 para o valor M, e 0 0 1 0 1 1 0 para F.

Em suma, ao implementar um banco colunar tanto a aplicação deve ser capaz de entender e conseguir recuperar informações mesmo comprimidas, como a modificação da camada de armazenamento deve ser implementada tal que se possa usufruir de algum método de compressão. Caso contrário um dos principais diferenciais de modelos colunares não será explorado e poderá não haver impacto no desempenho quando trabalhado com uma grande massa de dados, como é a realidade de empresas.

Capítulo 5

TPC *Benchmark* H

A essência de um *benchmark* é identificar e definir os melhores padrões de excelência para produtos, serviços e/ou processos, e então realizar aperfeiçoamentos de forma que esses padrões sejam alcançados [53]. De maneira geral, *benchmarks* se consolidaram como uma ferramenta para melhorar o desempenho de organizações e a competitividade nos negócios [54].

O TPC *Benchmark* H é um padrão de decisão de negócio internacionalmente aceito para comparações de desempenho entre SGBD utilizados em ambientes OLAP, criado pela organização TPC. De acordo com a página oficial do TPC [8], esta organização sem fins lucrativos foi fundada com o objetivo de definir padrões para *benchmarks* no processamento de transações e bancos de dados em geral. Ela é responsável pelo desenvolvimento e atualização destes *benchmarks*, bem como pela divulgação dos resultados apresentados por eles. Entre os sócios[1] responsáveis por isto estão as empresas Dell, Hewlet Packard, IBM, Microsoft, Oracle, Intel e Cisco. A lista de todos os sócios pode ser acessada na página oficial da organização.

Alguns trabalhos já foram realizados utilizando o TPC-H [55, 56, 57, 58, 59, 60]. Ngamsuriyaroj e Pornpattana [55] aplicam as consultas do TPC-H no MySQL Cluster a fim de avaliar seu desempenho. O trabalho de Nadee e Ngamsuriyaroj [56] tem uma proposta parecida com o primeiro [55], porém o intuito é avaliar o desempenho de um *cluster* Java EE baseado nas características das consultas do TPC-H. Thanopoulou et al. [57] foca em aplicar o TPC-H em bancos de dados de pequenas empresas que não podem arcar com configurações de *hardware* melhores para administrar um banco de dados. Barata, Bernardino e Furtado [58] fazem uma descrição teórica de dois *benchmarks*, o YCSB (*Yahoo Cloud Serving Benchmark*), voltado para a área de *Big Data*, e o TPC-H. Rutishauser e Noureldin [59] realizam uma análise comparativa

[1] Lista de sócios acessada em Junho de 2017

entre PostgreSQL e MongoDB aplicando o TPC-H. O primeiro é um SGBD relacional clássico, e o outro um SGBD NoSQL com algumas consultas do TPC-H adaptadas. Por fim, Soares [60] compara SGBD relacionais e colunares executando um teste de força sob o *Star Schema Benchmark* simulando bases de dados de 1GB, 2GB, 5GB e 10GB.

De acordo com o manual de especificação fornecido pelo TPC [61], o TPC-H é um *benchmark* de suporte à decisão de negócios, constituído de uma série de consultas comerciais *ad-hoc* e modificações simultâneas de dados com finalidade de retratar a realidade das empresas. Ele representa sistemas de suporte à decisão que examinam grandes volumes de dados, executam consultas com um alto grau de complexidade, e respondem questões críticas de negócio. Como o *benchmark* trata de grandes volumes de dados, o tamanho mínimo de banco de dados proposto pelo TPC-H é de 1GB, seguido por 10GB, 30GB, 100GB, 300GB, 1TB, 3TB, 10TB, 30TB e 100TB. Estes valores correspondem ao Fator de Escala (do inglês *Scale Factor*, ou SF) do banco de dados.

Com o intuito de avaliar o resultado do desempenho de SGBD como DW, é necessário ter conhecimento de como os dados que irão popular o DW estão distribuídos. Para tal, o TPC-H propõe um ambiente normalizado *snowflake*. Ele é composto por oito tabelas e é descrito na Seção 5.2. Do mesmo modo, a fim de obedecer as regras de modelagem deste ambiente, é preciso ter dados conhecidos para popular os DWs. Estes dados são fornecidos pelo *software* DBGen.

O DBGen foi implementado pelo TPC-H, com o objetivo de realizar a população de dados em um DW seguindo a modelagem original *snowflake*. São gerados oito arquivos separados no formato `<nome da tabela>.tbl` – exemplificando, a tabela de dados *part* será gerada como `part.tbl`; onde as linhas de cada arquivo representam as tuplas dentro da respectiva tabela, tendo seus atributos separados por um delimitador *pipe* ("|"). Por padrão, os arquivos são gerados para uma base de dados da classe de 1GB, quando nenhum SF é especificado.

Assim como é necessário conhecer os dados de modo a seguir a modelagem proposta pelo *benchmark*, é preciso formular consultas que visam responder às questões de negócio definidas pelo TPC-H sobre o modelo de dados. O próprio TPC define para seu modelo 22 consultas, cada qual definida por (i) uma questão de negócio, que ilustra o contexto no qual a consulta pode ser aplicada; (ii) uma definição funcional, que corresponde à implementação da consulta utilizando

a Linguagem SQL-92; (iii) parâmetros de substituição, que geram os valores necessários para completar os parâmetros da consulta; e (iv) validação, que descreve como validar a consulta no BD. Igualmente à geração de dados, a geração de consultas também é feita com o uso de um programa fornecido pelo TPC, o QGen.

Com o QGen é possível gerar as consultas de acordo com a especificação do TPC-H inserindo os parâmetros de substituição adequados em cada uma. A Consulta 1 presente no Apêndice A, Consulta de Relatório de Resumo de Preços, possui o parâmetro de substituição [DELTA], correspondente a um número inteiro entre 60 e 120, inserido ao executar o programa QGen para a respectiva consulta. Esses valores podem ser inseridos de maneira aleatória, entretanto o TPC-H define valores padrão para os parâmetros de substituição para fins de validação; portanto as consultas no QGen são geradas utilizando os valores padrão. As 22 consultas com suas respectivas questões de negócio e parâmetros de substituição são descritas no Apêndice A.

5.1 Metodologia

Após a geração dos arquivos é criado em cada SGBD um esquema correspondente às classes utilizadas, tanto para o ambiente normalizado quanto para o denormalizado. Cada SGBD é então executado sobre os dois ambientes propostos. Em cada ambiente será executado o teste de desempenho, que consiste de duas execuções: o teste de força (do inglês *Power Test*) e o teste de vazão (do inglês *Throughput Test*), descritos nas Subseções 5.1.1 e 5.1.2, respectivamente. O tempo resultante de cada etapa é utilizado para calcular o desempenho final do SGBD, medido em *QphH@Size* – quantidade de consultas executadas por hora, dada uma classe de tamanho de banco de dados. Uma execução é composta por:

- *SF*, que representa a classe do banco de dados.
- Q_i, que representa uma consulta, onde $1 \leqslant i \leqslant 22$.
- *S*, que define o número de sessões de consulta do teste de vazão.
- *s*, que representa uma dada sessão, onde $1 \leqslant s \leqslant S$.
- T_s, que representa o tempo, em segundos, resultante do processo inteiro.

- RF_j, que representa a Função de Atualização (do inglês *Refresh Function*, ou RF), onde:
  - RF-1: inserção de novos registros. O número de registros inseridos deve ser igual para o número de registros removidos pela RF-2. O pseudocódigo para a RF-1 é como:

```
loop (SF * 1500) times
    insert <new row into> ORDERS table
    loop random(1, 7) times
        insert <new row into> LINEITEM table
    end loop
end loop
```

  - RF-2: remoção de registros antigos. O pseudocódigo para a RF-1 é como:

```
loop (SF * 1500) times
    delete from ORDERS where O_ORDERKEY = [valor]
    delete from LINEITEM where I_ORDERKEY = [valor]
end loop
```

O conjunto de dados para executar com sucesso as RFs também são gerados pelo DBGen utilizando a *flag* -U.

5.1.1 Teste de Força

O teste de força objetiva medir a execução de uma dada consulta do sistema com um único usuário ativo. Neste teste é criada uma única sessão com o respectivo SGBD e as seguintes instruções são executadas:

- Execução da RF-1.
- Execução de cada consulta proposta pelo TPC-H, ou adaptada, de forma sequencial uma única vez até a última consulta.
- Execução da RF-2.

Será armazenado ao fim do teste o tempo em segundos resultante de cada consulta, bem como o tempo de execução de cada função. Este resultado será utilizado na Equação 5.1, onde *Size* representa o SF.

$$Power@Size = \frac{3600 * SF}{\sqrt[24]{\prod_{i=2}^{i=22} Q(i,0) * \prod_{j=1}^{j=2} RF(j,0)}} \tag{5.1}$$

5.1.2 Teste de Vazão

Este teste mede a capacidade do sistema de processar a maior quantidade possível de consultas no menor intervalo de tempo considerando vários usuários ativos no SGBD. Aqui o TPC-H exige um número mínimo de sessões de consulta de acordo com a classe de tamanho do banco de dados, como mostra a Tabela 5.1.

Tabela 5.1: Número mínimo de sessões para uma classe de banco de dados

Classe do Banco de Dados	**Número de Sessões**
1 GB	2
10 GB	3
30 GB	4
100 GB	5
300 GB	6
1 TB	7
3 TB	8
10 TB	9
30 TB	10
100 TB	11

São geradas *N* duplas de RFs (um arquivo com os registros para inserção e um com os índices a serem removidos), e é criada uma sessão para executar essas duplas de RFs e *N* sessões para executar as consultas, de acordo com o número mínimo que cada classe requer. As instruções são executadas concorrentemente da seguinte forma:

- As RFs ser executadas sequencialmente *N* vezes, onde *N* é o número de sessões para a execução de consultas.
- Para cada sessão de consultas, cada consulta será executada sequencialmente até a última.

Ao fim do teste, é armazenado o tempo em segundos da execução do processo inteiro. O processo inicia quando a primeira sessão, seja de consulta ou de RF, executa sua instrução e finaliza quando a última sessão recebe uma resposta. O resultado é utilizado para calcular o desempenho do SGBD para o teste de vazão conforme a Equação 5.2.

$$Throughput@Size = \frac{S*22*3600}{T_s*SF} \tag{5.2}$$

O resultado dos cálculos de força e vazão serão ponderados para calcular o desempenho final do SGBD da seguinte forma:

$$QphH@Size = \sqrt{Power@Size * Throughput@Size} \tag{5.3}$$

5.2 Ambiente Original Normalizado

O modelo de ambiente proposto pelo TPC-H é um esquema normalizado *snowflake*. Ele é composto por oito tabelas, sendo que destas, seis têm o tamanho multiplicado pelo SF, enquanto que as demais, *nation* e *region*, têm tamanho fixo. A cardinalidade do relacionamento entre as tabelas é *one-to-many* e é apresentado na Figura 5.1, adaptada do manual de especificação do TPC-H [61].

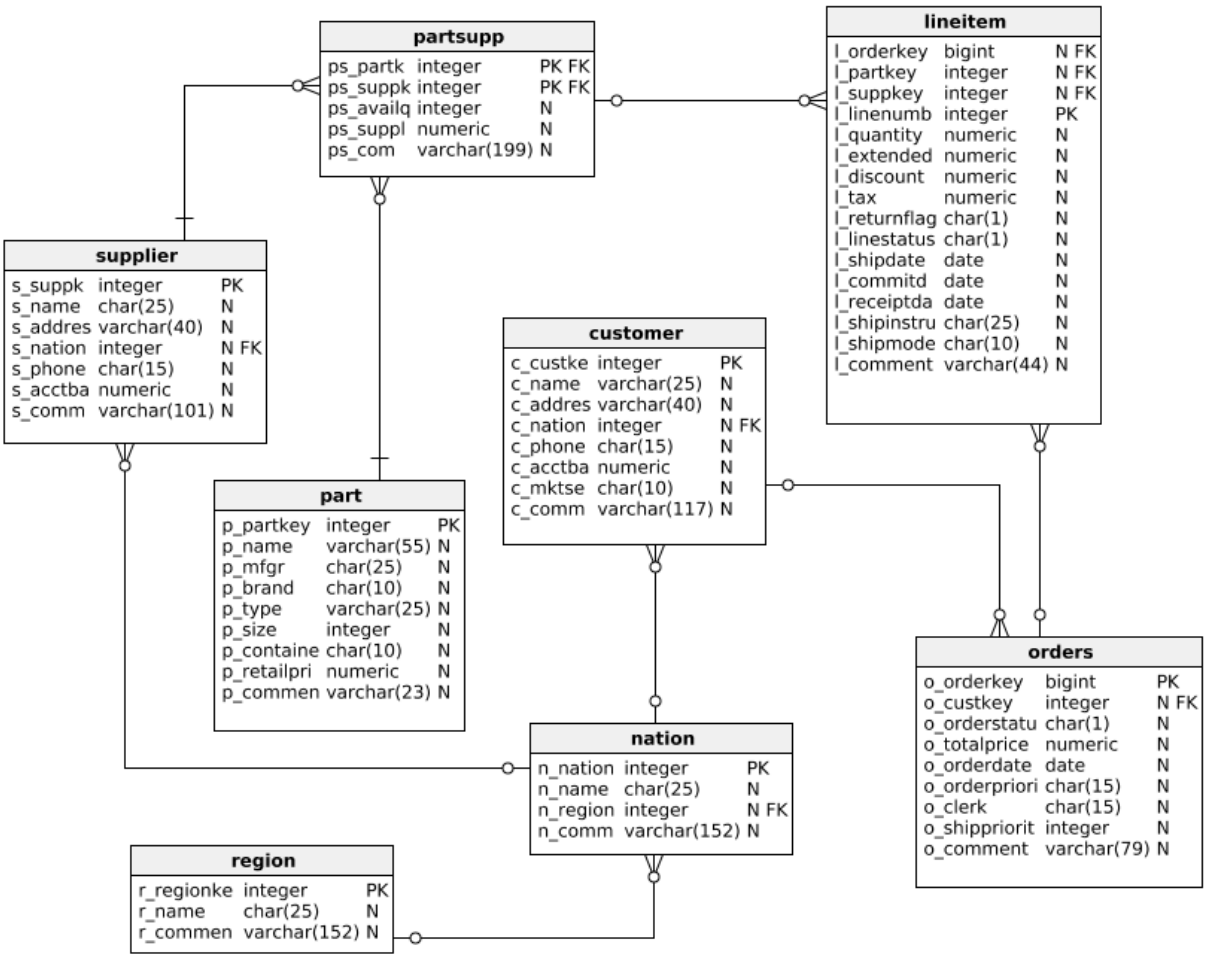


Figura 5.1: Esquema do ambiente normalizado

Os tipos de dados atribuídos aos atributos de uma entidade podem ser:

- Identificador, ou *identifier*: corresponde a um inteiro que define a PK de uma entidade
- Inteiro: int
- Decimal: decimal
- Texto fixo de tamanho N: text(N)
- Texto variável de tamanho N: varchar(N)
- Data: date

Detalhes sobre as consultas implementadas na Linguagem SQL relacionadas a esse ambiente são encontradas no Apêndice A.

5.3 Ambiente Denormalizado

As modificações no modelo do ambiente *snowflake* se fundamentaram na criação de um modelo parecido com o apresentado pela modelagem *star*, com uma Tabela Fato central descrita por Tabelas Dimensão. Também procurou-se eliminar quaisquer tabelas cujos atributos sejam frequentemente utilizados em operações de junção, alguns característicos de um modelo com Tabelas Subdimensão, e que possam ser incluídos nas tabelas que possuam relacionamento com eles. O esquema final do modelo *star* é mostrado na Figura 5.2.

A primeira modificação no modelo *snowflake* foi a remoção das entidades Dimensão *nation* e *region*. Essas entidades são Tabelas Subdimensão das entidades *customer* e *supplier*, sendo frequentemente requeridas quando há a necessidade de consultar a nação e/ou a região de um cliente e/ou de um fornecedor, assim optou-se por incluir os atributos `n_name, n_comment, r_name` e `r_comment` nas entidades *supplier* e *customer*.

Com a intenção de criar uma única Tabela Fato no esquema denormalizado, observou-se (i) a possibilidade de unir as entidades *lineitem* e *orders* em uma única entidade nomeada como *item*, visto que as duas possuem um relacionamento. Isto também eliminaria algumas junções realizadas entre as duas entidades nas consultas listadas no Apêndice A. A PK `c_custkey` de *customer* que antes estava em *orders* como FK é transferida agora para *item*. Também nota-se

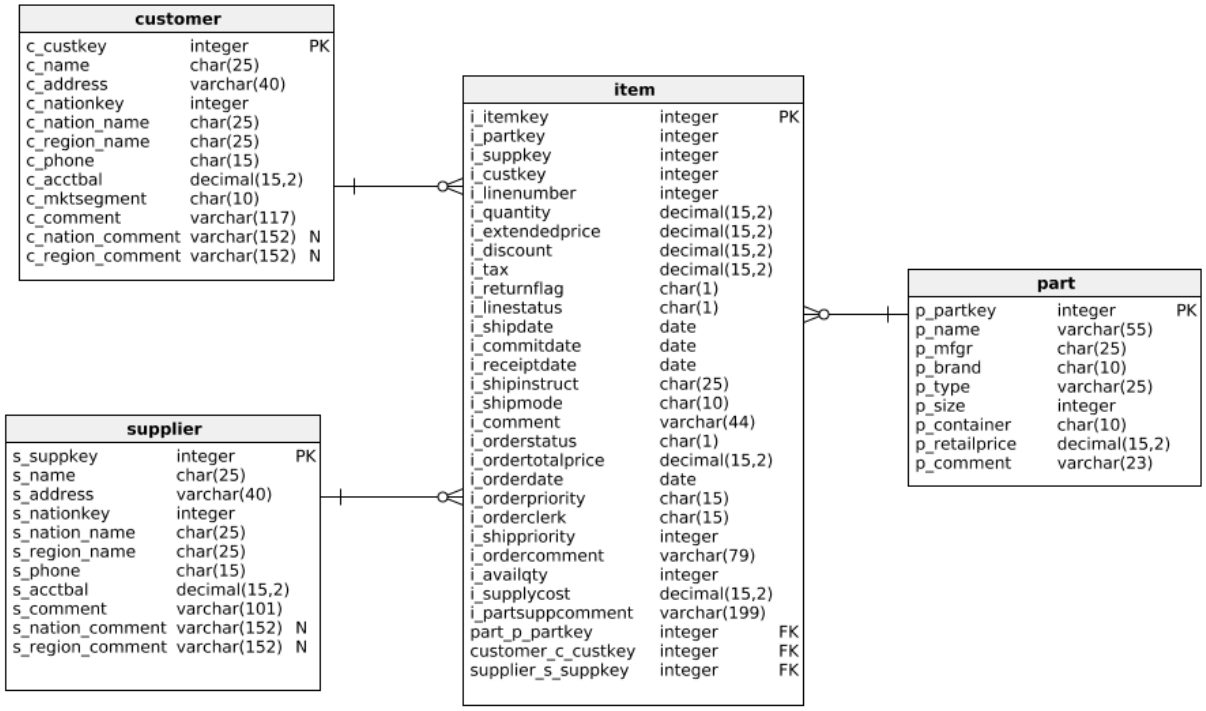


Figura 5.2: Esquema do ambiente denormalizado

que (ii) as entidades *part* e *supplier* têm suas PKs como FKs na entidade antes denominada *lineitem*, provenientes da entidade intermediária *partsupp*. Optou-se assim por incluir os atributos de *partsupp* na nova entidade *item*, excluindo a primeira do esquema e mantendo o relacionamento de *supplier* e *part* diretamente com *item*, através das chaves `p_partkey` e `s_suppkey`.

O esquema final agora possui uma Tabela Fato, *item*; e três tabelas Dimensão, *part*, *customer* e *supplier*, cada qual com sua chave mantendo o relacionamento com a tabela *item*. Os dados presentes nas entidades são do mesmo tipo descritos na Seção 5.2. As consultas em Linguagem SQL relacionadas a esse ambiente são encontradas no Apêndice B.

A inserção dos dados no ambiente *star* será realizada após todos os dados gerados pelo DBGen do ambiente normalizado terem sido populados nos SGBD. Com isso, é possível realizar junções de acordo com as mudanças e o resultado destes será armazenado no esquema correspondente. Por exemplo, ao realizar uma junção entre as entidades *nation* e *region* selecionando todos os seus atributos, estes são enviados como entrada para a inserção em uma nova tabela do esquema Denormalizado correspondente, digamos, *nation_region*. Após, é realizado uma nova junção com as entidades *supplier* e *part*, separadamente, para incluir os atributos de *nation_region*: `n_name, n_comment, r_name` e `r_comment`; bem como os atributos

originais das duas entidades anteriores, nas novas entidades *supplier* e *part*.

Capítulo 6

O Experimento

Como mencionado no Capítulo 3, muitas empresas ainda utilizam bancos relacionais como forma de gerenciar seus dados. Existem vários SGBDR atuantes no mercado, como MySQL, PostgreSQL e algumas variantes da Oracle. Por se tratar de um SGBD *open source* e ser amplamente conhecido, foi escolhido para o estudo o PostgreSQL. Ele possui vasta documentação, tutoriais e suporte em fóruns sobre bancos de dados e programação de forma geral, além de receber recorrentes atualizações.

A categoria de NoSQL escolhida foi a colunar, por ser a mais recomendada para gerenciamento do grande volume de dados de um DW. O fato do PostgreSQL utilizar a Linguagem SQL como forma de manipulação de dados faz com que seja necessário também escolher o banco colunar para comparação utilizando esse filtro. A escolha também se deu com base na simplicidade do uso do SGBD, e na praticidade de migração dos dados armazenados entre um banco e outro. Para tanto, o MonetDB foi definido para o estudo.

O gerenciamento dos SGBD foi realizado através do *driver* JDBC disponibilizados nas páginas oficiais do PostgreSQL e do MonetDB. Seu uso se deu devido à simplificação no desenvolvimento de aplicações, por dar suporte a diferentes SGBD e possuir boa documentação e tutoriais. Na análise de tempo foi considerada a latência da Máquina Virtual Java (JVM), assim incluindo essa latência no tempo total. As versões dos SGBD utilizadas foram o PostgreSQL 9.6 e o MonetDB 11.29.3 e foram instalados no servidor do laboratório GIA, operando sob o sistema operacional Ubuntu Linux 16.04 LTS, com 16Gb de RAM e dois processadores Intel Xeon CPU E5620 2.40GHz.

Os cenários de teste consistiram de três bases de dados de tamanhos diferentes, de 1Gb, 10Gb e 30Gb para simular desde uma base pequena até bases maiores. Cada cenário foi execu-

tado nos ambientes *snowflake* e *star*.

Alguns fatores devem ser considerados ao executar testes de *benchmark* no escopo de banco de dados [62]:

1. O ambiente de execução deve ser o mesmo para todos os SGBD, bem como a lógica e sequência de execução.

2. A modelagem do banco deve ser idêntica para todos os modelos, e isso inclui os tipos de dados e restrições de integridade. Ao analisar os resultados do *benchmark*, a menos que seja especificado no estudo, não há como saber qual é o tipo de dado de um atributo quando existem diferentes tipos que podem ser usados para a mesma função, e essa diferença acarreta em resultados diferentes de desempenho. Por exemplo, em Raasveldt et al. [62] o SGBD MariaDB apresentou diferenças quando implementado utilizando DECIMAL e DOUBLE.

3. *Cold run* vs. *Hot run*: em alguns sistemas a primeira execução de testes (*cold*) costuma levar mais tempo para executar que as execuções subsequentes (*hot*). Isso se dá pelo buffer do banco de dados ou o próprio sistema operacional ter armazenado em cache os dados requeridos.

Os itens 1 e 2 foram tratados antes da execução do TPC-H. Nos Apêndices C e D se encontram as SQL utilizadas para criar os esquemas *snowflake* e *star* nos SGBD. O fluxograma da Figura 6.1 ilustra ainda a sequência de passos desde a criação dos bancos até a execução do *benchmark*.

O item 3 foi tratado durante a execução das consultas do teste de força: para cada cenário é realizada três análises diferentes, a primeira considerando a primeira execução de consultas; a segunda considerando o melhor resultado entre três execuções; e a terceira limpando parte da cache do sistema operacional antes de rodar a terceira execução. Essa análise é feita somente sob as consultas do teste de força por ser o primeiro teste a ser feito, como pode ser observado na Figura 6.2, que ilustra o fluxograma do teste de força. No teste de vazão todas as consultas já foram executadas uma vez e as funções de atualização trabalham sobre dados diferentes a cada vez que são executadas, impossibilitando o armazenamento em cache.

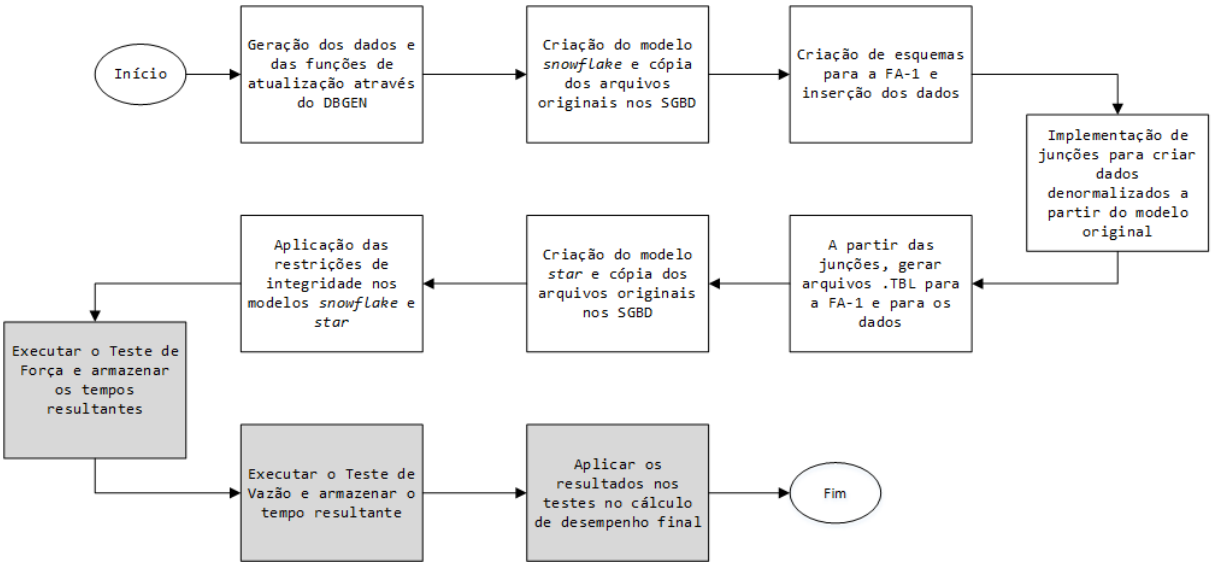


Figura 6.1: Fluxograma do desenvolvimento até o cálculo final de desempenho do TPC-H

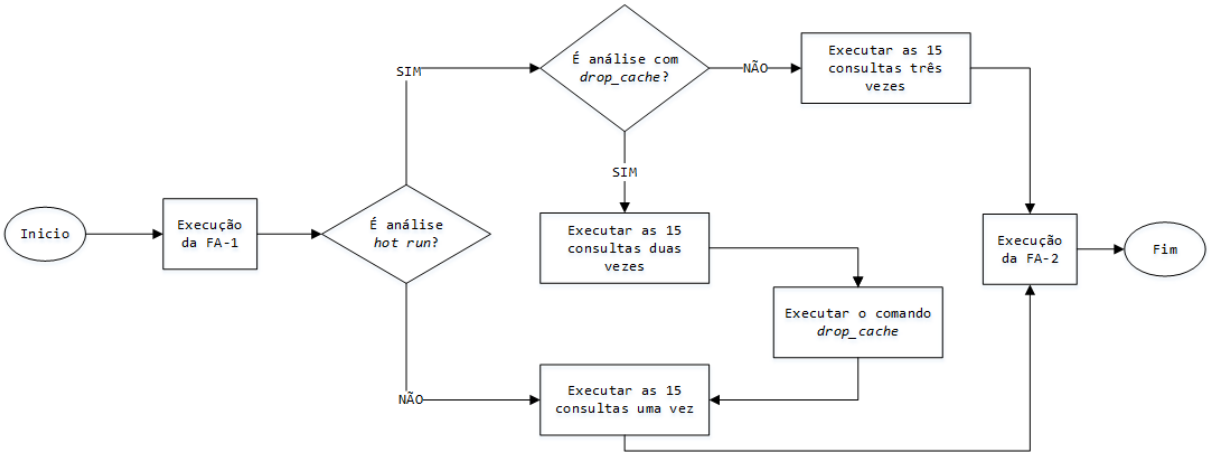


Figura 6.2: Fluxograma do desenvolvimento do teste de força considerando todas as variações de cenário

Para averiguar se todas as 22 consultas do TPC-H executam de forma correta e em tempo hábil, principalmente atentando-se ao fato de que no ambiente denormalizado elas foram modificadas, foi realizada uma execução prévia de cada uma, sob a menor base de dados, utilizando o JDBC. Dessa execução, foram eliminadas as consultas Q1, Q2, Q4, Q15, Q17, Q20 e Q21 por terem demorado horas no ambiente denormalizado – e algumas tendo a execução cancelada antes – enquanto que as demais levaram apenas alguns segundos para executar. Essa mudança no número de consultas fez com que as equações originais fossem adaptadas para a nova quantidade, 15.

$$Power@Size = \frac{3600*SF}{\sqrt[17]{\prod_{i=1}^{i=15} Q(i,0)*\prod_{j=1}^{j=2} RF(j,0)}} \tag{6.1}$$

$$Throughput@Size = \frac{S*15*3600}{T_s*SF} \tag{6.2}$$

6.1 Carregamento de Dados

Após gerar todos os arquivos contendo os registros para inserção nos SGBD, foi utilizando o comando *COPY*. Para se obter um melhor desempenho na inserção de dados as restrições de chaves foram aplicadas após a inserção de todos os dados. Caso a restrição fosse feita na criação do banco a cada novo registro inserido a chave primária seria verificada. Esta carga de dados também foi realizada através dos SGBD de forma direta, sem o uso de *drivers* e outras ferramentas que possam atuar como cliente para conectar-se ao banco, assim evitando incluir o tempo de latência destas no resultado. Os comandos no MonetDB e no PostgreSQL são, respectivamente:

```
COPY INTO tabela FROM arquivo USING DELIMITERS delimitadores;
COPY tabela FROM arquivo DELIMITER delimitadores;
```

A Tabela 6.1 mostra o tempo em segundos para a inserção de dados nos SGBD. Logo ao analisar os resultados de carregamento de dados nota-se que um banco denormalizado apresenta certa demora na inserção de dados em relação a um normalizado. Isso se dá por causa da redundância de dados existente neste tipo de modelo, fazendo com que existam mais atributos duplicados em uma mesma entidade; aumentando o tamanho das entidades a serem inseridas.

Tabela 6.1: Tempo de carregamento em segundos para os cenários de *benchmark*

SGBD	Base de Dados (Gb)			Ambiente
	1	10	30	
MonetDB	44	409	1351	*Snowflake*
PostgreSQL	104	816	4979	
MonetDB	88	689	2216	*Star*
PostgreSQL	166	2365	7216	

Toma-se como exemplo a entidade *nation*: no modelo *snowflake* ela é uma tabela subdimensão e possui 25 registros apenas, enquanto que no modelo denormalizado ela faz parte das tabelas dimensão *supplier* e *customer*, fazendo com que estas duas entidades tenham mais bytes armazenados. A Tabela 6.2 mostra o tamanho dos arquivos gerados para cada modelo para a base de dados de 1Gb. Note que *part* não tem seu valor alterado por não sofrer alteração entre as modelagens.

Tabela 6.2: Tamanho em bytes das entidades

	Snowflake	Star
Supplier	1.409.184	2.990.430
Customer	24.346.144	48.055.517
Part	24.135.125	24.135.125
Lineitem/Item	759.863.287	2.226.620.122

Um ponto importante a ser analisado foi o tamanho final de cada SGBD. A Tabela 6.3 mostra que mesmo para a mesma quantidade de dados o MonetDB tem o tamanho em bytes reduzido, mostrando a capacidade que um SGBD colunar tem de realizar compressão de dados de forma mais eficiente.

Tabela 6.3: Tamanho do banco de dados em Mb para os cenários de *benchmark*

SGBD	Base de Dados (Gb)			Ambiente
	1	10	30	
MonetDB	771	8087	21352	*Snowflake*
PostgreSQL	1567	15510	44371	
MonetDB	952	9352	26271	*Star*
PostgreSQL	2565	25487	71586	

6.2 Execução *Cold Run*

O primeiro cenário de execução consiste em analisar os resultados de desempenho considerando a primeira execução sequencial de todas as consultas do teste de força.

6.2.1 Base de Dados de 1Gb

Considerando fator de escala de 1Gb, o número de registros das tabelas *supplier, customer, part, partsupp, orders* e *lineitem* são respectivamente 10.000, 150.000, 200.000, 800.000, 1.500.000 e 6.001.215. O número de registros das tabelas *region* e *nation* são fixos em 5 e 25, respectivamente. Também, em *lineitem* o número de registros depende de um pedido em *orders*, sendo definido sob um número aleatório entre um e quatro.

Tabela 6.4: Tempo em segundos de todas as consultas do teste de força e funções de atualização sob *cold run* para 1Gb

	Snowflake		***Star***	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
RF-1	2.742	49.125	2.975	45.889
Q3	9.094	13.615	18.056	65.649
Q5	0.985	2.673	1.877	3.851
Q6	1.846	3.365	2.053	2.779
Q7	0.892	1.648	1.324	2.913
Q8	1.256	2.685	2.341	5.132
Q9	3.485	5.870	3.867	5.375
Q10	0.983	2.975	1.468	3.427
Q11	0.356	0.503	2.224	7.608
Q12	2.199	2.958	2.853	3.097
Q13	2.502	2.639	11.019	12.157
Q14	0.129	2.012	0.266	2.691
Q16	0.617	1.626	1.766	9.761
Q18	1.613	6.968	1.280	15.757
Q19	0.646	2.578	0.922	3.508
Q22	0.323	0.979	0.967	4.738
Total	26.925	53.092	52.285	148.443
RF-2	40.812	38.091	20.260	20.575

O primeiro conjunto de resultados para a execução de 1Gb pode ser visto nas Tabelas 6.4 e 6.5. A primeira mostra os tempos em segundos que os dois SGBD levaram para executar as RF e as 15 consultas. Nela é possível verificar que a primeira consulta (Q3) é a que mais

consome tempo de execução, e que o MonetDB não tem desempenho melhor que o PostgreSQL na remoção de registros no ambiente *snowflake*, e ao contrário do que se esperava, este SGBD em ambiente denormalizado levou mais tempo para processar as consultas ao normalizado.

Na Tabela 6.5 é possível perceber que o SGBD que obteve resultados dentro do esperado foi o PostgreSQL. O desempenho considerando um único usuário ativo foi melhor no ambiente normalizado, assim como no teste simulando vários usuários ativos. O MonetDB só alcançou resultados melhores no ambiente denormalizado para o último teste.

Tabela 6.5: Valores do teste de força e vazão sob *cold run* para 1Gb

	Teste de Força		**Teste de Vazão**	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
Snowflake	2541.636	980.630	1622.929	958.028
Star	1504.254	501.254	2468.020	637.310

6.2.2 Base de Dados de 10Gb

Tabela 6.6: Tempo em segundos de todas as consultas do teste de força e funções de atualização sob *cold run* para 10Gb

	Snowflake		***Star***	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
RF-1	14.351	1304.641	12.596	1042.060
Q3	125.044	108.010	164.697	194.322
Q5	8.762	37.100	10.069	192.718
Q6	18.130	19.793	13.621	155.163
Q7	2.913	28.333	0.327	151.504
Q8	9.072	25.755	9.396	151.010
Q9	18.126	93.686	14.613	209.017
Q10	5.551	46.611	5.347	137.693
Q11	3.416	10.951	15.150	261.915
Q12	15.206	29.331	20.039	129.920
Q13	19.837	35.913	43.488	256.449
Q14	0.443	19.775	2.696	132.184
Q16	4.251	14.154	14.163	225.790
Q18	8.330	137.693	8.901	349.738
Q19	4.347	25.855	2.962	134.205
Q22	2.367	14.979	8.144	234.272
Total	245.795	647.938	333.613	2915.899
RF-2	526.547	379.049	157.183	196.822

O número de registros das tabelas *supplier, customer, part, partsupp* e *orders* correspondem aos valores da base de 1Gb, porém 10 vezes maior – exceto *lineitem*, cujo valor é 59.986.052. São respectivamente 100.000, 1.500.000, 2.000.000, 8.000.000, 15.000.000.

Nesta execução fica evidente a diferença de tempo da primeira consulta para as demais no banco colunar. Tomando a segunda mais lenta no MonetDB normalizado e denormalizado (Q13), a Q3 é 6.3 vezes no primeiro ambiente, e quase quatro vezes no segundo, mais demorada. Ainda, o ambiente denormalizado continua tendo desempenho pior que o normalizado no MonetDB, como é possível verificar na Tabela 6.6. A Tabela 6.7 continua apresentando a mesma lógica que a Tabela 6.5 nos resultados.

Tabela 6.7: Valores do teste de força e vazão sob *cold run* para 10Gb

	Teste de Força		**Teste de Vazão**	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
Snowflake	3687.621	775.652	330.479	364.541
Star	3116.068	174.381	583.560	149.455

6.2.3 Base de Dados de 30Gb

O número de registros das tabelas *supplier, customer, part, partsupp* e *orders* correspondem aos valores da base de 1Gb, porém 30 vezes maior. São respectivamente 300.000, 4.500.000, 6.000.000, 24.000.000, 45.000.000, e *lineitem* tem agora 179.998.372 registros.

Pela Tabela 6.9 pode-se verificar que no banco colunar o comportamento permanece o mesmo às bases anteriores. Desta vez ainda a escalabilidade do tempo da consulta Q14 aumentou em 208 vezes em comparação à base de 1Gb no ambiente normalizado e 334 vezes no denormalizado, enquanto que as demais consultas escalaram em um intervalo de valores com menos discrepâncias, ficando entre 13 e 75, e 12 e 69. Os resultados do teste de força e vazão são apresentados na Tabela 6.8.

Tabela 6.8: Valores do teste de força e vazão sob *cold run* para 30Gb

	Teste de Força		**Teste de Vazão**	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
Snowflake	2329.918	369.019	112.572	220.005
Star	1474.766	145.064	165.419	120.578

Tabela 6.9: Tempo em segundos de todas as consultas do teste de força e funções de atualização sob *cold run* para 30Gb

	Snowflake		***Star***	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
RF-1	45.464	4346.899	29.397	4011.410
Q3	250.579	369.306	507.637	566.055
Q5	43.135	415.032	79.365	581.650
Q6	87.109	203.049	70.875	525.149
Q7	12.287	302.879	24.758	542.459
Q8	24.375	286.555	45.394	545.627
Q9	104.949	597.415	54.817	717.576
Q10	14.495	296.734	19.087	545.421
Q11	9.308	50.995	68.687	1073.476
Q12	96.438	260.277	79.686	531.999
Q13	62.886	124.179	194.227	937.462
Q14	26.823	219.780	89.018	549.628
Q16	11.826	74.054	68.930	834.195
Q18	109.423	796.233	88.697	1225.532
Q19	48.729	191.799	46.524	482.566
Q22	8.121	88.714	25.656	969.976
Total	910.485	4277.000	1463.358	10628.772
RF-2	1392.401	1120.928	735.083	586.293

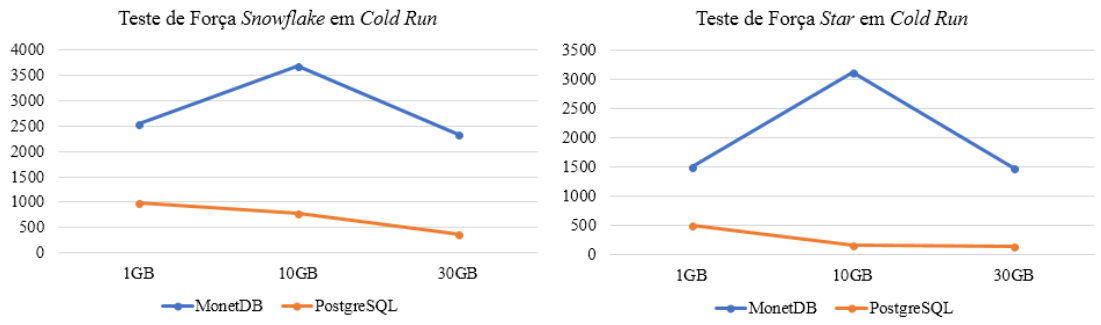
6.2.4 Desempenho Geral

No cenário rodando sob *cold run* o MonetDB não obteve resultados considerados estáveis como o PostgreSQL. Embora haja a conclusão que o SGBD colunar é melhor que o relacional, avaliando os dados dentro do MonetDB não se sabe ao certo qual modelagem é melhor, trazendo incerteza na sua escolha.

Considerando vários usuários ativos o resultado foi como esperado. Se considerar também somente as RF, o MonetDB teve valores muito superiores para inserção de dados, e na remoção somente para uma base maior no ambiente denormalizado o PostgreSQL foi melhor. O que impactou nos resultados do banco colunar foi o comportamento anormal das consultas.

O comportamento do teste de força no banco MonetDB não segue uma queda conforme a base de dados aumenta. Existe uma inflexão na base de dados de 10Gb, pois não há um comportamento similar entre as consultas como acontece no PostgreSQL. Os gráficos das Figuras 6.3(a) e 6.3(b) evidenciam isso.

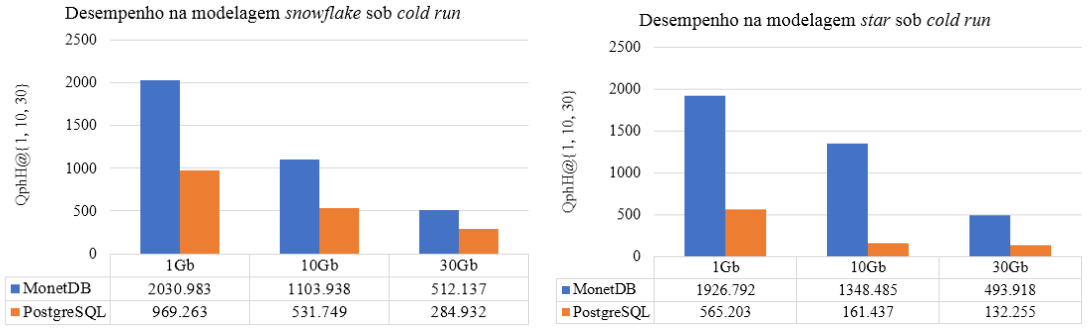
O cálculo de desempenho final foi afetado por essa inflexão. Se analisar os resultados con-



(a) (b)

Figura 6.3: Gráficos de força entre ambientes sob *cold run*

siderando os SGBD sob as modelagens *snowflake* e *star*, o MonetDB se destaca no resultado final, fazendo dele a melhor escolha entre os bancos, como mostram os gráficos da Figuras 6.4(a) e 6.4(b) e os ganhos obtidos por ele em relação ao PostgreSQL conforme a Tabela 6.10. Porém ao considerar os ambientes sob cada SGBD, a única conclusão coerente que se tem é de que o PostgreSQL é melhor em ambientes normalizados, como ilustra o gráfico da Figura 6.5(b), no qual conforme a base aumenta o cálculo de desempenho é menor e sempre há ganhos do normalizado em relação ao denormalizado, sendo estes **42%, 70%** e **54%** para 1Gb, 10Gb e 30Gb respectivamente.

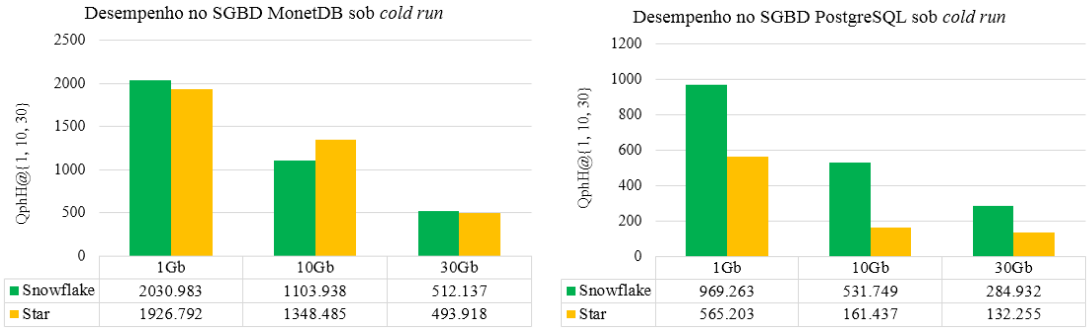


(a) (b)

Figura 6.4: Gráficos de desempenho entre ambientes sob *cold run*

Tabela 6.10: Porcentagem de ganho de desempenho do MonetDB em relação ao PostgreSQL sob *cold run*

Ambiente	**Base de Dados (Gb)**		
	1	**10**	**30**
Snowflake	110%	108%	80%
Star	241%	735%	273%



(a) (b)

Figura 6.5: Gráficos de desempenho entre SGBD sob *cold run*

O MonetDB não segue uma lógica da qual se pode tirar alguma conclusão de acordo com a modelagem. Conforme o tamanho da base de dados aumenta o valor diminui assim como no PostgreSQL, porém entre as modelagens há uma inconsistência na comparação entre seus valores, visto que para 1Gb o ambiente normalizado foi melhor com ganho de **5%**, já para 10Gb foi o denormalizado com **22%** de ganho, e para 30Gb o comportamento é como em 1Gb com **4%** de ganho do normalizado, como ilustra o gráfico da Figura 6.5(a).

6.3 Execução *Hot Run*

O segundo cenário de execução analisa os resultados de desempenho considerando a terceira execução sequencial de todas as consultas do teste de força, sendo esta executada imediatamente após as duas primeiras. O objetivo é verificar se há diferença nos resultados encontrados no cenário anterior. Os registros e a quantidade são os mesmos dos cenários anteriores.

6.3.1 Base de Dados de 1Gb

Na terceira execução de todas as consultas já nota-se uma grande diferença nos valores do MonetDB, tanto no ambiente normalizado quanto no denormalizado. A primeira consulta já não é a mais lenta, e todas as demais têm valores similares considerando que a base de dados é pequena, como mostra a Tabela 6.12. Percebe-se também que agora o ambiente denormalizado não é mais lento que o normalizado. Este resultado impacta nos valores de força, visto que pode-se analisar a partir da Tabela 6.11 que agora o ambiente *star* tem valor superior ao *snowflake* no MonetDB. Não apenas superior, os valores resultantes do teste de força possuem magnitude

maior que os do cenário anterior devido a eficiência do processamento das consultas.

Tabela 6.11: Valores do teste de força e vazão sob *hot run* para 1Gb

	Teste de Força		**Teste de Vazão**	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
Snowflake	14043.395	1222.570	1622.929	958.028
Star	21811.705	659.758	2468.020	637.310

Tabela 6.12: Tempo em segundos de todas as consultas do teste de força e funções de atualização sob *hot run* para 1Gb

	Snowflake		***Star***	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
RF-1	2.742	49.125	2.975	45.889
Q3	0.184	1.874	0.079	3.231
Q5	0.116	1.165	0.063	3.303
Q6	0.047	2.126	0.051	2.762
Q7	0.149	1.611	0.067	2.796
Q8	0.519	2.178	0.078	3.344
Q9	0.301	5.004	0.089	5.187
Q10	0.090	2.703	0.049	3.381
Q11	0.099	0.499	0.102	7.510
Q12	0.161	2.955	0.070	3.085
Q13	0.155	2.609	0.092	12.106
Q14	0.050	2.008	0.060	2.593
Q16	0.383	1.708	0.667	9.359
Q18	0.768	6.949	0.508	8.970
Q19	0.140	2.553	0.121	3.204
Q22	0.073	0.973	0.090	3.777
Total	3.236	36.918	2.184	74.606
RF-2	40.812	38.091	20.260	20.575

6.3.2 Base de Dados de 10Gb

Assim como a base de dados de 1Gb, a de 10Gb possui um comportamento que trazem resultados melhores que no cenário considerando *cold run*, e novamente o MonetDB apresentou valores mais consistentes, e o ambiente denormalizado apresentou resultados melhores que o normalizado.

Tabela 6.13: Valores do teste de força e vazão sob *hot run* para 10Gb

	Teste de Força		**Teste de Vazão**	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
Snowflake	11861.997	870.994	330.479	364.541
Star	17902.155	190.307	583.560	149.455

Tabela 6.14: Tempo em segundos de todas as consultas do teste de força e funções de atualização sob *hot run* para 10Gb

	Snowflake		***Star***	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
RF-1	14.351	1304.641	12.596	1042.060
Q3	2.118	36.095	1.506	125.089
Q5	1.171	34.593	1.273	159.875
Q6	0.438	19.789	0.602	132.427
Q7	2.679	28.960	1.277	146.097
Q8	6.416	24.142	1.428	126.529
Q9	3.004	107.818	0.597	170.805
Q10	1.324	35.949	0.223	128.092
Q11	2.515	5.494	0.686	263.927
Q12	3.649	29.372	1.505	122.578
Q13	6.173	31.265	8.303	261.873
Q14	0.390	18.737	0.161	132.393
Q16	1.161	14.239	6.757	225.479
Q18	8.136	125.887	5.643	339.079
Q19	2.356	26.911	1.801	125.114
Q22	0.570	19.855	2.736	215.812
Total	42.100	559.107	34.498	2675.167
RF-2	526.547	379.049	157.183	196.822

6.3.3 Base de Dados de 30Gb

Para a base de 30Gb além de manter-se o comportamento das anteriores, percebe-se um distanciamento maior dos valores do MonetDB para o PostgreSQL em *hot run*. Os valores da Tabela 6.16 mostram isso e a Tabela 6.15 apresenta os valores de força e vazão.

6.3.4 Desempenho Geral

Ao contrário do cenário sob *cold run*, utilizar o terceiro conjunto de consultas trouxe estabilidade nos valores das consultas do teste de força no MonetDB. Isso fez com que a inflexão que antes acontecia não acontecesse mais, podendo agora encontrar uma lógica na análise dos

Tabela 6.15: Valores do teste de força e vazão sob *hot run* para 30Gb

	Teste de Força		**Teste de Vazão**	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
Snowflake	5457.325	363.441	112.572	220.005
Star	6873.122	149.514	165.419	120.578

Tabela 6.16: Tempo em segundos de todas as consultas do teste de força e funções de atualização sob *hot run* para 30Gb

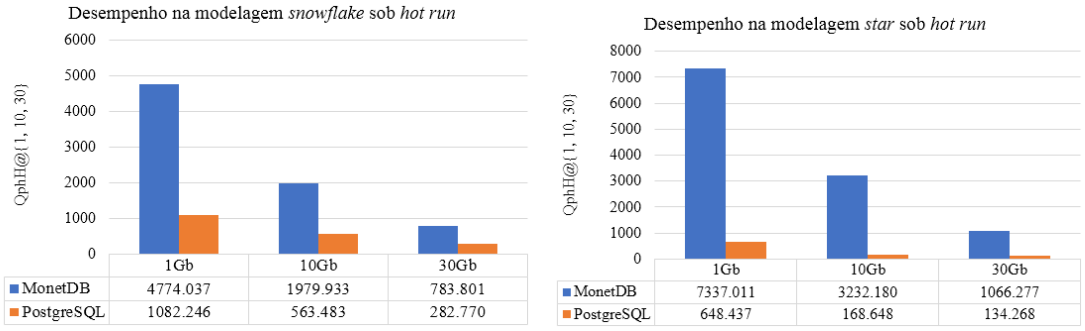
	Snowflake		***Star***	
SGBD	**MonetDB**	**PostgreSQL**	**MonetDB**	**PostgreSQL**
RF-1	45.464	4346.899	29.397	4011.410
Q3	68.066	329.075	57.567	553.795
Q5	12.812	373.031	12.553	537.089
Q6	2.579	186.510	1.608	473.006
Q7	3.086	279.495	1.487	501.833
Q8	6.984	281.131	6.220	507.657
Q9	60.413	607.088	30.875	671.788
Q10	11.154	310.681	9.409	518.880
Q11	9.597	68.390	9.784	980.716
Q12	42.212	261.946	40.280	476.657
Q13	62.234	144.715	56.124	916.111
Q14	1.576	224.288	1.297	551.986
Q16	11.634	75.528	13.937	835.383
Q18	56.660	771.133	49.197	1224.231
Q19	28.730	186.968	27.263	495.281
Q22	7.037	104.941	6.001	1127.978
Total	384.775	4204.921	323.601	10372.392
RF-2	1392.401	1120.928	735.083	586.293

ambientes considerando o SGBD. As Figuras 6.6(a) e 6.6(b) apresentam os gráficos do teste de força, agora apresentando queda nos valores conforme a base de dados aumenta.

Pelo gráfico das Figuras 6.7(a) e 6.7(b) percebe-se que o MonetDB continua melhor que o PostgreSQL, conforme os ganhos apresentados na Tabela 6.17, e agora seus valores também são superiores aos calculados no desempenho final considerando a *cold run*. Ainda, analisando os ambientes sob cada SGBD pelos gráficos das Figuras 6.8(a) e 6.8(b) nota-se pela segunda que o PostgreSQL continua tendo desempenho melhor no ambiente normalizado ao denormalizado, com ganhos de **40%, 70%** e **53%** enquanto que a primeira mostra que agora o MonetDB possui valores superiores em todas as bases para o ambiente denormalizado, com ganhos de **54%, 63%** e **63%**.

(a) (b)

Figura 6.6: Gráficos de força entre ambientes sob *hot run*

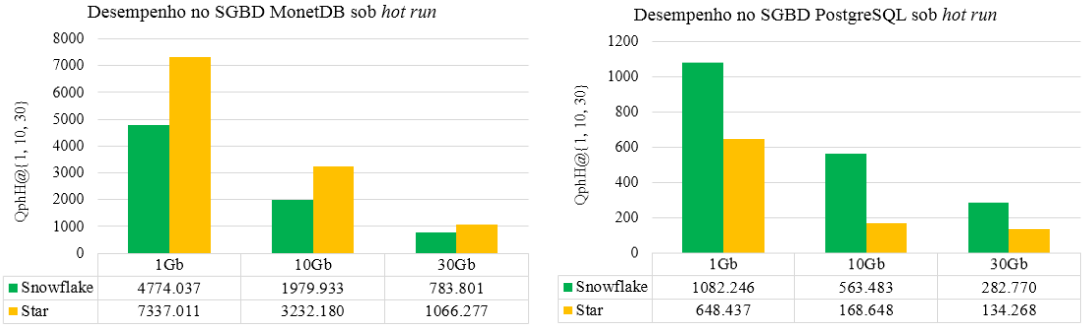


(a) (b)

Figura 6.7: Gráficos de desempenho entre ambientes sob *hot run*

Tabela 6.17: Porcentagem de ganho de desempenho do MonetDB em relação ao PostgreSQL sob *hot run*

Ambiente	**Base de Dados (Gb)**		
	1	**10**	**30**
Snowflake	341%	251%	177%
Star	1031%	1817%	694%



(a) (b)

Figura 6.8: Gráficos de desempenho entre SGBD sob *hot run*

Os valores do SGBD PostgreSQL não tiveram mudanças drásticas se comparar as execuções *cold* e *hot*, inclusive na base de dados 30Gb sob a modelagem normalizada a primeira foi melhor que a segunda. A Tabela 6.18 mostra a porcentagem de ganho e perdas entre os valores obtidos na terceira execução em relação à primeira execução para o PostgreSQL. Entretanto, os valores do MonetDB se alteraram de forma que houve impacto no resultado final tanto do teste de força como no cálculo final de *benchmark*, como mostram os ganhos da *hot run* em relação à *cold run* na Tabela 6.19.

Tabela 6.18: Porcentagem de ganho e perda de desempenho do PostgreSQL da *hot run* em relação à *cold run*

Ambiente	**Base de Dados (Gb)**		
	1	**10**	**30**
Snowflake	10%	6%	-1%
Star	13%	4%	1%

Tabela 6.19: Porcentagem de ganho de desempenho do MonetDB da *hot run* em relação à *cold run*

Ambiente	**Base de Dados (Gb)**		
	1	**10**	**30**
Snowflake	57%	44%	35%
Star	74%	58%	54%

De acordo com a página oficial do MonetDB [36], este SGBD se destaca quando dados acessados pelo usuário podem ser mantidos na memória principal do servidor ou quando algumas colunas são solicitadas e são suficientes para manipular uma solicitação, como é o caso do ambiente analítico. Isso explica o porquê das consultas da primeira execução apresentarem desempenho inferior às da terceira execução. Na segunda e na terceira execução alguns dados das consultas já estão em *cache*, assim o SGBD consegue recuperá-los mais rapidamente.

Dois exemplos que mostram a estabilidade nos valores alcançados pela *cache* estão na primeira consulta (Q3) e na 11ª (Q14). No cenário de *cold run* a primeira consulta sempre levava mais tempo que as demais e a Q14 demorava excessivamente conforme a base crescia, em detrimento das demais. Considerando este cenário uma justificativa a se utilizar seria a de que as consultas eram demoradas, porém sob o cenário *hot run* se vê que elas não apresentam o mesmo comportamento, tendo influência da memória *cache*.

6.4 Execução com *Drop Cache*

O último cenário consiste em, levando em consideração que o MonetDB sofre influência da memória *cache*, executar as consultas de força três vezes como no cenário anterior, porém antes da terceira execução efetuar a limpeza na memória *cache* do sistema. A intenção é averiguar se os resultados serão alterados ao mudar o estado da *cache*.

6.4.1 Base de Dados de 1Gb

O primeiro detalhe que se nota de acordo com os valores de tempo totais, em segundos, de cada execução de consultas é que a segunda execução é sempre a mais rápida devido a alguns valores já estarem armazenados em *cache*. Após ter limpado parte da *cache*, a terceira execução já torna-se mais lenta que a segunda, porém não mais que a primeira, pois alguns valores ainda podem ter ficado armazenados na memória.

Tabela 6.20: Tempo total de execução das consultas da 1ª, 2ª e 3ª execução para 1Gb

SGBD	1ª Execução	2ª Execução	3ª Execução	Ambiente
MonetDB	20.842	2.811	9.973	*Snowflake*
PostgreSQL	77.689	37.906	54.328	
MonetDB	54.197	3.082	26.147	*Star*
PostgreSQL	94.184	75.135	91.649	

Dentre todas as consultas, a que mais sofreu impacto da oscilação da *cache* foi a primeira. A Tabela 6.21 mostra os valores para a Q3 de todas as execuções.

Tabela 6.21: Tempo total de execução da primeira consulta para 1Gb

SGBD	1ª Execução	2ª Execução	3ª Execução	Ambiente
MonetDB	7.758	0.228	1.034	*Snowflake*
PostgreSQL	36.574	1.855	13.698	
MonetDB	18.345	0.071	9.656	*Star*
PostgreSQL	20.054	3.038	19.919	

6.4.2 Base de Dados de 10Gb

A mesma lógica da base de dados de 1Gb segue para a de 10Gb, segundo a Tabela 6.22. Na execução do PostgreSQL em ambiente *snowflake*, entretanto, houve uma queda no tempo

de execução da segunda execução para a terceira, indicando possivelmente menor influência da *cache* neste SGBD.

Tabela 6.22: Tempo total de execução das consultas da 1ª, 2ª e 3ª execução para 10Gb

SGBD	1ª Execução	2ª Execução	3ª Execução	Ambiente
MonetDB	412.301	31.825	90.525	*Snowflake*
PostgreSQL	950.210	750.324	744.324	
MonetDB	398.951	50.781	294.667	*Star*
PostgreSQL	3042.546	2772.970	2884.537	

Nesta base já nota-se de forma acentual a influência da memória na primeira consulta como mostra a Tabela 6.23, e consequentemente no resultado final de tempo de execução, podendo ocasionar a mesma inflexão apresentada pela primeira execução dos testes.

Tabela 6.23: Tempo total de execução da primeira consulta para 10Gb

SGBD	1ª Execução	2ª Execução	3ª Execução	Ambiente
MonetDB	273.352	2.460	11.734	*Snowflake*
PostgreSQL	112.322	43.252	113.553	
MonetDB	216.923	22.201	155.161	*Star*
PostgreSQL	200.664	149.393	218.902	

6.4.3 Base de Dados de 30Gb

Assim como as anteriores, a Tabela 6.24 mostra que a terceira execução é afetada pela *cache*, bem como a primeira consulta, cujos valores são mostrados na Tabela 6.25.

Tabela 6.24: Tempo total de execução das consultas da 1ª, 2ª e 3ª execução para 30Gb

SGBD	1ª Execução	2ª Execução	3ª Execução	Ambiente
MonetDB	1062.709	495.334	688.593	*Snowflake*
PostgreSQL	4574.974	4433.061	4300.868	
MonetDB	1338.430	674.249	1243.989	*Star*
PostgreSQL	11128.495	10788.900	10884.908	

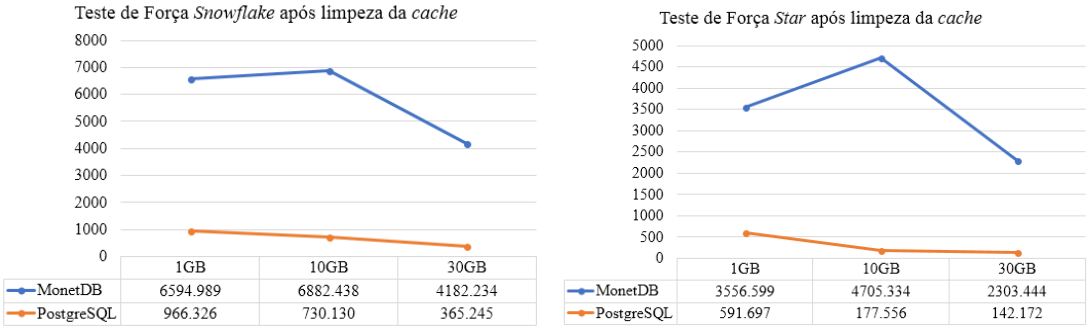
6.4.4 Desempenho Geral

As três subseções anteriores mostraram como a *cache* influencia nos resultados do processamento das consultas, principalmente no SGBD MonetDB. Aplicando o cálculo do teste de força

Tabela 6.25: Tempo total de execução da primeira consulta para 30Gb

SGBD	1ª Execução	2ª Execução	3ª Execução	Ambiente
MonetDB	487.614	122.496	322.512	*Snowflake*
PostgreSQL	381.837	353.038	361.421	
MonetDB	517.566	146.583	465.564	*Star*
PostgreSQL	586.428	570.704	571.626	

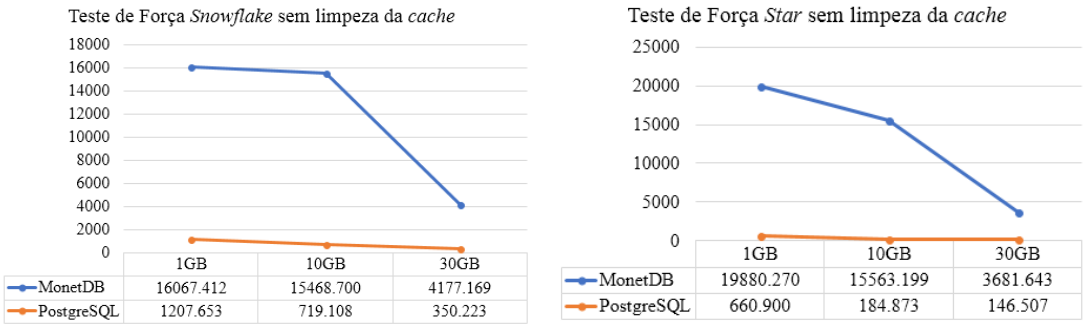
nos valores oriundos da terceira execução é observado a volta da inflexão na base de 10Gb, como mostram os gráficos das Figuras 6.9(a) e 6.9(b).



(a) (b)

Figura 6.9: Gráfico de força entre ambientes sob influência da limpeza de *cache*

Para averiguar de fato essa influência da limpeza de *cache* na inflexão, o mesmo teste de força foi aplicado nos valores de tempo da segunda execução, que já tinha alguns dados armazenados na memória. Os gráficos das Figuras 6.10(a) e 6.10(b) ilustram que não ocorreu inflexão na segunda base de dados ao não limpar a *cache*.

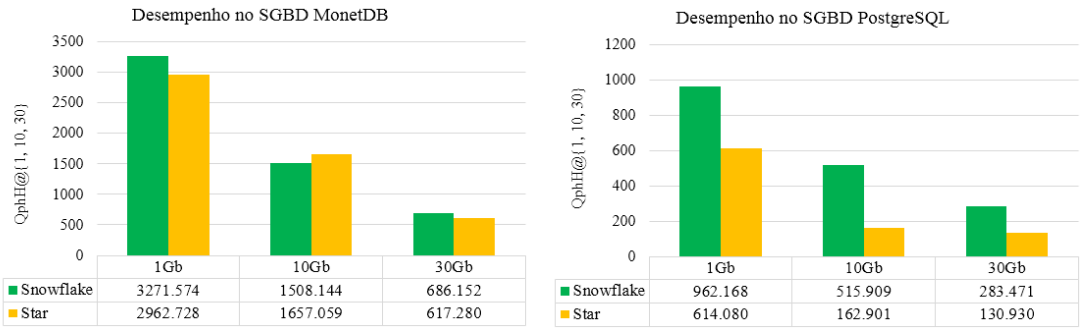


(a) (b)

Figura 6.10: Gráficos de força sem limpeza de *cache*

A influência da inflexão pode ser vista na Figura 6.11(a), que assim como a Figura 6.5(a) não apresenta conclusão acerca da escolha da modelagem para o banco MonetDB. A Figura

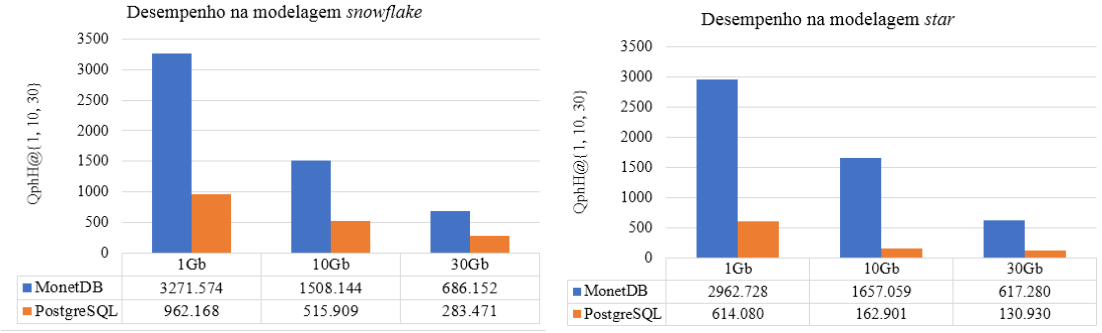
6.11(b) mostra os valores finais entre os ambientes no PostgreSQL, e a Figura 6.12 os gráficos comparando os SGBD dentro de cada ambiente.



(a) (b)

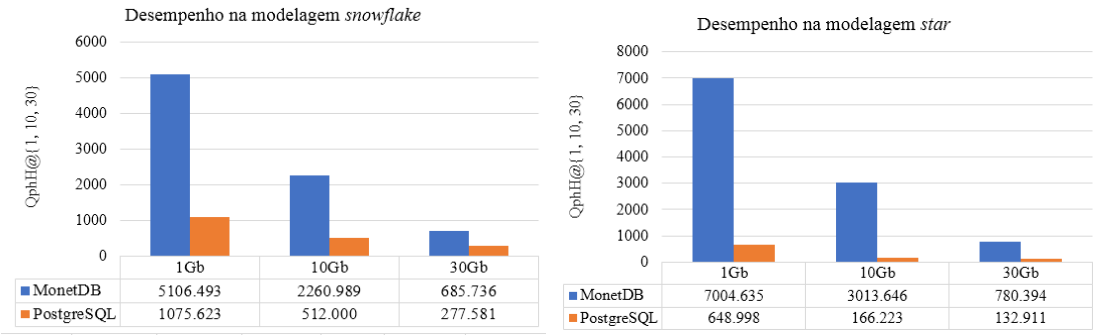
Figura 6.11: Gráficos de desempenho entre SGBD sob influência da limpeza de *cache*

Considerando a segunda execução, o resultado final de desempenho pode ser visto nos gráficos das Figuras 6.13 e 6.14. Nelas percebe-se a mesma lógica que as Figuras 6.7 e 6.8 apresentam.



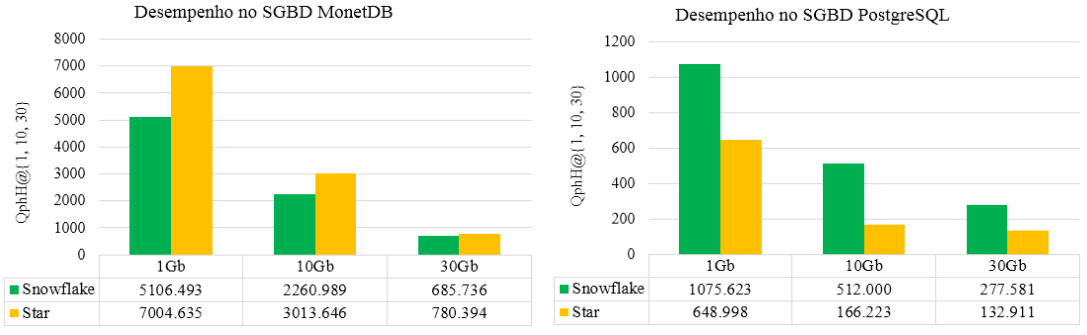
(a) (b)

Figura 6.12: Gráficos de desempenho entre ambientes sob influência da limpeza de *cache*



(a) (b)

Figura 6.13: Gráficos de desempenho entre ambientes sem limpeza de *cache*



(a) (b)

Figura 6.14: Gráficos de desempenho entre SGBD sem limpeza de *cache*

Capítulo 7

Conclusões

A principal discussão em um ambiente analítico é a necessidade por desempenho na recuperação de informações para tomada de decisão. Para conseguir alcançar de forma eficiente esse desempenho bancos relacionais são muito utilizados, porém podem trazer alguns problemas de acordo com a modelagem dos dados dos DWs. Para tanto, bancos NoSQL, mais especificamente da classe orientada a colunas, ganham destaque por satisfazerem questões mais complexas em SGBD relacionais, tornando possível a recuperação rápida de dados.

Entre os destaques encontrados em um modelo de SGBD colunar estão (i) a capacidade de escalabilidade horizontal devido ao melhor desempenho sob modelagens denormalizadas no formato *star*, o que faz com que não seja preciso normalizar um modelo a um nível que gere alto grau de acoplamento entre as tabelas do banco, (ii) busca dos atributos requeridos de forma direta, sem precisar processar tuplas inteiras, satisfazendo as requisições de consultas analíticas por poucos atributos, gerando rapidez na recuperação de dados, e (iii) alta capacidade de compressão de dados. Ao realizar a avaliação experimental os dois últimos pontos ficaram bem evidentes.

Desde o carregamento de dados nota-se uma grande diferença entre os SGBD MonetDB e PostgreSQL. O comando *COPY INTO* utilizado pelo MonetDB consegue inserir os dados no SGBD concorrentemente – um fator que o torna preferível ao invés de vários *INSERT*, por tratar o comando como uma única *query*[1] independentemente de ter o modo de *auto-commit* das transações ativo ou não. Mesmo que o comando *COPY FROM* do PostgreSQL assim como do MonetDB trate o comando como uma única *query*, o mesmo desempenho não foi atingido.

[1] Aqui *query* não significa exatamente uma *consulta*, portanto optou-se por deixar a palavra no inglês por não encontrar uma tradução adequada

A Tabela 7.1 mostra os ganhos do MonetDB em relação ao PostgreSQL no tempo de carregamento. Logo após a inserção de dados também percebe-se um grande diferencial dos bancos colunares: a compressão de dados, uma vez que o MonetDB realiza compressão de *strings* utilizando codificação por dicionário.

Tabela 7.1: Porcentagem de ganho de tempo de carregamento do MonetDB em relação ao PostgreSQL

Ambiente	**Base de Dados (Gb)**		
	1	**10**	**30**
Snowflake	58%	50%	73%
Star	47%	71%	69%

Como um todo a teoria sobre SGBDR vs. SGBDC se aplicou bem à prática. O PostgreSQL teve um ótimo desempenho se comparar o ambiente mais normalizado ao denormalizado. Por ser desenvolvido visando modelos relacionais, o processamento de dados nas consultas tanto com uma quanto com várias *streams* foi mais rápido no ambiente normalizado. Considerando vários usuários ativos o PostgreSQL conseguiu ser mais rápido que o MonetDB no ambiente normalizado para as bases de dados maiores (10Gb e 30Gb). Porém se analisar o desempenho final isso não fez com que o banco relacional pudesse ter desempenho melhor que o colunar no ambiente analítico.

Um fator que trouxe resultados melhores no MonetDB, além dos já citados de bancos colunares, foi o fato de que MonetDB faz uso intensivo de memória para processamento – vale frisar que ele não requer que todos os dados caibam em toda memória física disponível, pois utiliza o espaço disponível em *swap* e arquivos mapeados em memória. Devido a esse uso intensivo de memória, medir o tempo de uma consulta é algo delicado no MonetDB. Os testes comparando *cold run*, *hot run* e limpeza de *cache* mostraram como a mesma consulta executada várias vezes e sob um cenário alterando a memória trazem resultados diferentes nesse SGBD, fazendo com que sob a primeira execução os resultados não acordem com a teoria descrita sobre o desempenho superior em ambientes *star*.

Retomando a discussão entre OLAP e OLTP, armazenar os dados de um ambiente transacional em memória pode trazer problemas, tornando mais interessante de fato para um ambiente transacional que seus dados sejam armazenados em um dispositivo não-volátil. Um ambiente altamente normalizado perde por não escalar horizontalmente de forma tão simples, o que já

acarreta demora na recuperação do servidor em caso de queda, portanto não é interessante que ocorra perda de dados em memória. Em um ambiente analítico, a inserção de dados já é mais rápida considerando o uso do MonetDB e ele pode ser escalável horizontalmente, fazendo com que uma queda e perda de dados em memória não seja tão crítica.

Para ambientes transacionais cuja modelagem ainda segue os padrões de normalização o uso de um SGBD como o PostgreSQL continua sendo vantajoso, pois além dos resultados obtidos deve se considerar que um SGBDR preza pela consistência nos dados, e para um sistema transacional que envolva transações bancárias, por exemplo, a consistência é fundamental. Em ambiente analítico é confirmado que um banco colunar traz melhor desempenho para a recuperação de informações e o MonetDB ainda possui a vantagem de utilizar dados em memória. Para que se faça um bom uso da *cache* nesse SGBD é possível aproveitar justamente o fator de consultas analíticas, embora não idênticas, serem similares por simular cenários com agrupamentos e atributos Fato, para realizar um *warm-up* de dados para que estes já estejam em *cache*.

7.1 Trabalhos Futuros

Como continuidade para a pesquisa sugere-se:

- Realizar um estudo mais aprofundado sobre escalabilidade horizontal em bancos NoSQL, e se possível aplicar o *benchmark* considerando um cenário utilizando um único servidor para processamento de consultas e outro com vários servidores, a fim de simular problemas que teoricamente são solucionados pelo particionamento horizontal, como queda, ou falha, e distribuição de dados.

- Utilizar bases de dados maiores, a partir de 100Gb, para que se possa também comparar os resultados com os contidos na página oficial do TPC-H.

- Aplicar os resultados encontrados nesta pesquisa em empresas que ainda utilizem bancos relacionais ou, mesmo que usem outras alternativas, que ainda tenham uma modelagem normalizada, realizando a migração tanto do SGBD quanto da modelagem para um modelo colunar denormalizado.

- Comparar diferentes modelos colunares que, de preferência, trabalhem sob diferentes algoritmos de compressão, como o Cassandra [39] e o C-Store [37]. Pode-se também incluir diferentes classes de NoSQL a fim de verificar se alguma consegue desempenho melhor que a classe colunar em ambiente analítico, como o MongoDB [34].

Apêndice A

Consultas do Ambiente Original Normalizado

São apresentadas aqui as 15 consultas realizadas no Ambiente *snowflake*, cada qual com sua questão de negócio brevemente explicada.

1. Consulta de Prioridade de Envio

   Retorna a prioridade de envio dos pedidos com a maior receita entre aqueles que ainda não foram enviados em uma determinada data.

Parâmetro de Substituição	Valor
SEGMENT	BUILDING
DATE	1995-03-15

```
select
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
from
    customer,
    orders,
    lineitem
where
    c_mktsegment = '[SEGMENT]'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate < date '[DATE]'
    and l_shipdate > date '[DATE]'
group by
    l_orderkey,
    o_orderdate,
    o_shippriority
```

```
order by
   revenue desc,
   o_orderdate;
set rowcount 10
go
```

2. Consulta de Volume do Fornecedor Local

   Lista para cada nação em uma dada região o volume de receita originado de uma transação na qual cliente e fornecedor eram daquela nação.

Parâmetro de Substituição	Valor
REGION	ASIA
DATE	1994-01-01

```
select
   n_name,
   sum(l_extendedprice * (1 - l_discount)) as revenue
from
   customer,
   orders,
   lineitem,
   supplier,
   nation,
   region
where
   c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and l_suppkey = s_suppkey
   and c_nationkey = s_nationkey
   and s_nationkey = n_nationkey
   and n_regionkey = r_regionkey
   and r_name = '[REGION]'
   and o_orderdate >= date '[DATE]'
   and o_orderdate < date '[DATE]' + interval '1' year
group by
   n_name
order by
   revenue desc;
```

3. Previsão de Mudança de Receita

   Informa o quanto a receita pode aumentar eliminando alguns descontos em um dado ano.

Parâmetro de Substituição	Valor
DATE	1994-01-01
DISCOUNT	.06
QUANTITY	24

```
select
   sum(l_extendedprice * l_discount) as revenue
from
   lineitem
where
   l_shipdate >= date '[DATE]'
   and l_shipdate < date '[DATE]' + interval '1' year
   and l_discount between [DISCOUNT] - 0.01 and [DISCOUNT] + ↩
      0.01
   and l_quantity < [QUANTITY];
```

4. Consulta de Quantidade de Envio

   Determina o valor de bens enviados entre nações para auxiliar na renegociação de contratos de envio.

Parâmetro de Substituição	Valor
NATION1	FRANCE
NATION2	GERMANY

```
select
   supp_nation,
   cust_nation,
   l_year,
   sum(volume) as revenue
from
   (
      select
         n1.n_name as supp_nation,
         n2.n_name as cust_nation,
         extract(year from l_shipdate) as l_year,
         l_extendedprice * (1 - l_discount) as volume
      from
         supplier,
         lineitem,
         orders,
         customer,
         nation n1,
         nation n2
      where
         s_suppkey = l_suppkey
```

```
            and o_orderkey = l_orderkey
            and c_custkey = o_custkey
            and s_nationkey = n1.n_nationkey
            and c_nationkey = n2.n_nationkey
            and (
               (n1.n_name = '[NATION1]' and n2.n_name = '[NATION2]↩
                  ')
               or (n1.n_name = '[NATION2]' and n2.n_name = '[↩
                  NATION1]')
            )
            and l_shipdate between date '1995-01-01' and date '↩
               1996-12-31'
   ) as shipping
group by
   supp_nation,
   cust_nation,
   l_year
order by
   supp_nation,
   cust_nation,
   l_year;
```

5. Consulta de Quota de Mercado Nacional

Determina quanto a quota de mercado de uma dada nação em uma dada região mudou em dois anos para um determinado tipo de peça.

Parâmetro de Substituição	Valor
NATION	BRAZIL
REGION	AMERICA
TYPE	ECONOMY ANODIZED STEEL

```
select
   o_year,
   sum(case
      when nation = '[NATION]'
      then volume
      else 0
   end) / sum(volume) as mkt_share
from
   (
      select
         extract(year from o_orderdate) as o_year,
         l_extendedprice * (1 - l_discount) as volume,
         n2.n_name as nation
      from
         part,
```

```
            supplier,
            lineitem,
            orders,
            customer,
            nation n1,
            nation n2,
            region
        where
            p_partkey = l_partkey
            and s_suppkey = l_suppkey
            and l_orderkey = o_orderkey
            and o_custkey = c_custkey
            and c_nationkey = n1.n_nationkey
            and n1.n_regionkey = r_regionkey
            and r_name = '[REGION]'
            and s_nationkey = n2.n_nationkey
            and o_orderdate between date '1995-01-01' and date '↩
                1996-12-31'
                    and p_type = '[TYPE]'
    ) as all_nations
group by
    o_year
order by
    o_year;
```

6. Consulta ao Lucro de um Tipo de Produto

Encontra para cada nação em cada ano o lucro de todas as peças pedidas naquele ano contendo uma substring específica em seus nomes que foram preenchidos por um fornecedor naquela nação.

Parâmetro de Substituição	Valor
COLOR	green

```
select
    nation,
    o_year,
    sum(amount) as sum_profit
from
    (
        select
            n_name as nation,
            extract(year from o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) - ps_supplycost * ↩
                l_quantity as a
mount
```

```
        from
            part,
            supplier,
            lineitem,
            partsupp,
            orders,
            nation
        where
            s_suppkey = l_suppkey
            and ps_suppkey = l_suppkey
            and ps_partkey = l_partkey
            and p_partkey = l_partkey
            and o_orderkey = l_orderkey
            and s_nationkey = n_nationkey
            and p_name like '%[COLOR]%'
    ) as profit
group by
    nation,
    o_year
order by
    nation,
    o_year desc;
```

7. Consulta de Relatório de Itens Retornados

   Retorna os 20 principais clientes que podem estar tendo problemas com peças que foram enviadas a eles em um dado trimestre.

Parâmetro de Substituição	Valor
DATE	1993-10-01

```
select
    c_custkey,
    c_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
from
    customer,
    orders,
    lineitem,
    nation
where
    c_custkey = o_custkey
```

```
    and l_orderkey = o_orderkey
    and o_orderdate >= date '[DATE]'
    and o_orderdate < date '[DATE]' + interval '3' month
    and l_returnflag = 'R'
    and c_nationkey = n_nationkey
group by
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
order by
    revenue desc;
set rowcount 20
go
```

8. Consulta a Identificação de Estoques Importantes

Avalia de todos os estoques de fornecedores disponíveis em uma dada nação as peças com uma percentagem significativa do total de peças disponíveis.

Parâmetro de Substituição	Valor
NATION	GERMANY
FRACTION	.0001

```
select
    ps_partkey,
    sum(ps_supplycost * ps_availqty) as value
from
    partsupp,
    supplier,
    nation
where
    ps_suppkey = s_suppkey
    and s_nationkey = n_nationkey
    and n_name = '[NATION]'
group by
    ps_partkey having
        sum(ps_supplycost * ps_availqty) > (
            select
                sum(ps_supplycost * ps_availqty) * [FRACTION]
            from
                partsupp,
                supplier,
                nation
```

```
            where
                ps_suppkey = s_suppkey
                and s_nationkey = n_nationkey
                and n_name = '[NATION]'
        )
order by
    value desc;
```

9. Consulta a Modos de Envio e Prioridade de Pedidos

   Determina quando selecionar modos de envio mais baratos afeta negativamente pedidos com alta prioridade.

Parâmetro de Substituição	Valor
SHIPMODE1	MAIL
SHIPMODE2	SHIP
DATE	1994-01-01

```
select
    l_shipmode,
    sum(case
        when o_orderpriority = '1-URGENT'
            or o_orderpriority = '2-HIGH'
            then 1
        else 0
    end) as high_line_count,
    sum(case
        when o_orderpriority <> '1-URGENT'
            and o_orderpriority <> '2-HIGH'
            then 1
        else 0
    end) as low_line_count
from
    orders,
    lineitem
where
    o_orderkey = l_orderkey
    and l_shipmode in ('[SHIPMODE1]', '[SHIPMODE2]')
    and l_commitdate < l_receiptdate
    and l_shipdate < l_commitdate
    and l_receiptdate >= date '[DATE]'
    and l_receiptdate < date '[DATE]' + interval '1' year
group by
    l_shipmode
order by
    l_shipmode;
```

10. Consulta à Distribuição de Clientes

Determina a distribuição de clientes pelo número de pedidos que eles fizeram.

Parâmetro de Substituição	Valor
WORD1	special
WORD2	requests

```
select
   c_count,
   count(*) as custdist
from
   (
      select
         c_custkey,
         count(o_orderkey)
      from
         customer left outer join orders on
            c_custkey = o_custkey
            and o_comment not like '%[WORD1]%[WORD2]%'
      group by
         c_custkey
   ) as c_orders (c_custkey, c_count)
group by
   c_count
order by
   custdist desc,
   c_count desc;
```

11. Consulta aos Efeitos de Promoção

Informa o retorno de mercado a uma propaganda, como um comercial de televisão ou uma campanha especial.

Parâmetro de Substituição	Valor
DATE	1995-09-01

```
select
   100.00 * sum(case
      when p_type like 'PROMO%'
         then l_extendedprice * (1 - l_discount)
      else 0
   end) / sum(l_extendedprice * (1 - l_discount)) as ↩
      promo_revenue
from
   lineitem,
   part
```

```
where
   l_partkey = p_partkey
   and l_shipdate >= date '[DATE]'
   and l_shipdate < date '[DATE]' + interval '1' month;
```

12. Consulta à Relação *Part/Supplier*

Descobre quantos fornecedores podem fornecer peças com determinados atributos requeridos por um cliente.

Parâmetro de Substituição	Valor
BRAND	Brand#45
TYPE	MEDIUM POLISHED
SIZE1	49
SIZE2	14
SIZE3	23
SIZE4	45
SIZE5	19
SIZE6	3
SIZE7	36
SIZE8	9

```
select
   p_brand,
   p_type,
   p_size,
   count(distinct ps_suppkey) as supplier_cnt
from
   partsupp,
   part
where
   p_partkey = ps_partkey
   and p_brand <> '[BRAND]'
   and p_type not like '[TYPE]%'
   and p_size in ([SIZE1], [SIZE2], [SIZE3], [SIZE4], [SIZE5], ↩
      [SIZE6], [SIZE7], [SIZE8])
   and ps_suppkey not in (
      select
         s_suppkey
      from
         supplier
      where
         s_comment like '%Customer%Complaints%'
   )
group by
   p_brand,
```

```
    p_type,
    p_size
order by
    supplier_cnt desc,
    p_brand,
    p_type,
    p_size;
```

13. Consulta a Grandes Volumes de Clientes

Classifica os 100 principais clientes que já realizaram grandes quantidades de pedidos.

Parâmetro de Substituição	Valor
QUANTITY	300

```
select
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice,
    sum(l_quantity)
from
    customer,
    orders,
    lineitem
where
    o_orderkey in (
        select
            l_orderkey
        from
            lineitem
        group by
            l_orderkey having
                sum(l_quantity) > [QUANTITY]
    )
    and c_custkey = o_custkey
    and o_orderkey = l_orderkey
group by
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice
order by
    o_totalprice desc,
    o_orderdate;
```

14. Consulta a Desconto de Receita

Encontra o desconto bruto de todos os pedidos para três diferentes tipos de peças que foram enviadas por via aérea e entregues pessoalmente.

Parâmetro de Substituição	Valor
QUANTITY1	300
BRAND1	Brand#12
QUANTITY2	10
BRAND2	Brand#23
QUANTITY3	20
BRAND3	Brand#34

```
select
   sum(l_extendedprice* (1 - l_discount)) as revenue
from
   lineitem,
   part
where
   (
      p_partkey = l_partkey
      and p_brand = '[BRAND1]'
      and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM ↩
         PKG')
      and l_quantity >= [QUANTITY1] and l_quantity <= [↩
         QUANTITY1] + 10
      and p_size between 1 and 5
      and l_shipmode in ('AIR', 'AIR REG')
      and l_shipinstruct = 'DELIVER IN PERSON'
   )
   or
   (
      p_partkey = l_partkey
      and p_brand = '[BRAND2]'
      and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED↩
          PACK')
      and l_quantity >= [QUANTITY2] and l_quantity <= [↩
         QUANTITY2] + 10
      and p_size between 1 and 10
      and l_shipmode in ('AIR', 'AIR REG')
      and l_shipinstruct = 'DELIVER IN PERSON'
   )
   or
   (
      p_partkey = l_partkey
      and p_brand = '[BRAND3]'
      and p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG ↩
```

```
        PKG')
    and l_quantity >= [QUANTITY3] and l_quantity <= [↩
        QUANTITY3] + 10
    and p_size between 1 and 15
    and l_shipmode in ('AIR', 'AIR REG')
    and l_shipinstruct = 'DELIVER IN PERSON'
);
```

15. Consulta a Oportunidades de Vendas Globais

Conta quantos clientes de determinados países não realizaram pedidos durante sete anos, porém têm chances de realizar um pedido.

Parâmetro de Substituição	Valor
I1	13
I2	31
I3	23
I4	29
I5	30
I6	18
I7	17

```
select
    cntrycode,
    count(*) as numcust,
    sum(c_acctbal) as totacctbal
from
    (
        select
            substring(c_phone from 1 for 2) as cntrycode,
            c_acctbal
        from
            customer
        where
            substring(c_phone from 1 for 2) in
                ('[I1]', '[I2]', '[I3]', '[I4]', '[I5]', '[I6]', '[↩
                    I7]')
            and c_acctbal > (
                select
                    avg(c_acctbal)
                from
                    customer
                where
                    c_acctbal > 0.00
                    and substring(c_phone from 1 for 2) in
                        ('[I1]', '[I2]', '[I3]', '[I4]', '[I5]', '[I6↩
                            ]', '[I7]')
```

```
        )
        and not exists (
            select
                *
            from
                orders
            where
                o_custkey = c_custkey
        )
    ) as custsale
group by
    cntrycode
order by
    cntrycode;
```

Apêndice B

Consultas do Ambiente Denormalizado

São apresentadas aqui as 15 consultas realizadas no Ambiente Denormalizado adaptado da proposta original do TPC-H. As questões de negócio são as mesmas que as apresentadas no Apêndice A.

1. Consulta de Prioridade de Envio

Parâmetro de Substituição	Valor
SEGMENT	BUILDING
DATE	1995-03-15

```
select
   i_itemkey,
   sum(i_extendedprice * (1 - i_discount)) as revenue,
   i_order_orderdate,
   i_order_shippriority
from
   customer,
   item
where
   c_mktsegment = '[SEGMENT]'
   and c_custkey = i_custkey
   and i_order_orderdate < date '[DATE]'
   and i_shipdate > date '[DATE]'
group by
   i_itemkey,
   i_order_orderdate,
   i_order_shippriority
order by
   revenue desc,
   o_orderdate
set rowcount 10
go
```

2. Consulta de Volume do Fornecedor Local

Parâmetro de Substituição	Valor
REGION	ASIA
DATE	1994-01-01

```
select
   s_nation_name,
   sum(i_extendedprice * (1 - i_discount)) as revenue
from
   customer,
   item
   supplier,
where
   c_custkey = i_custkey
   and i_suppkey = s_suppkey
   and c_nation_name = s_nation_name
   and s_region_name = '[REGION]'
   and i_order_orderdate >= date '[DATE]'
   and i_order_orderdate < date '[DATE]' + interval '1' year
group by
   s_nation_name
order by
   revenue desc;
```

3. Previsão de Mudança de Receita

Parâmetro de Substituição	Valor
DATE	1994-01-01
DISCOUNT	.06
QUANTITY	24

```
select
   sum(i_extendedprice * i_discount) as revenue
from
   item
where
   i_shipdate >= date '[DATE]'
   and i_shipdate < date '[DATE]' + interval '1' year
   and i_discount between [DISCOUNT] - 0.01 and [DISCOUNT] + ↩
      0.01
   and i_quantity < [QUANTITY];
```

4. Consulta de Quantidade de Envio

Parâmetro de Substituição	Valor
NATION1	FRANCE
NATION2	GERMANY

```
select
   supp_nation,
   cust_nation,
   i_year,
   sum(volume) as revenue
from
   (
      select
         s_nation_name as supp_nation,
         c_nation_name as cust_nation,
         extract(year from i_shipdate) as i_year,
         i_extendedprice * (1 - i_discount) as volume
      from
         supplier,
         item,
         customer,
      where
         s_suppkey = i_suppkey
         and c_custkey = i_custkey
         and (
            (s_nation_name = '[NATION1]' and c_nation_name = '[↩
               NATION2]')
            or (s_nation_name = '[NATION2]' and c_nation_name =↩
                '[NATION1]')
         )
         and i_shipdate between date '1995-01-01' and date '↩
            1996-12-31'
   ) as shipping
group by
   supp_nation,
   cust_nation,
   i_year
order by
   supp_nation,
   cust_nation,
   i_year;
```

5. Consulta de Quota de Mercado Nacional

Parâmetro de Substituição	Valor
NATION	BRAZIL
REGION	AMERICA
TYPE	ECONOMY ANODIZED STEEL

```
select
   order_year,
   sum(case
      when nation = '[NATION]' then volume
```

```
        else 0
    end) / sum(volume) as mkt_share
from
    (
        select
            extract(year from i_order_orderdate) as order_year,
            i_extendedprice * (1 - i_discount) as volume,
            s_nation_name as nation
        from
            part,
            supplier,
            item,
            customer
        where
            p_partkey = i_partkey
            and s_suppkey = i_suppkey
            and i_custkey = c_custkey
            and c_region_name = '[REGION]'
            and i_order_orderdate between date '1995-01-01' and ↩
                date '1996-12-31'
            and p_type = '[TYPE]'
    ) as all_nations
group by
    order_year
order by
    order_year;
```

6. Consulta ao Lucro de um Tipo de Produto

Parâmetro de Substituição	Valor
COLOR	green

```
select
    nation,
    order_year,
    sum(amount) as sum_profit
from
    (
        select
            s_nation_name as nation,
            extract(year from i_order_orderdate) as order_year,
            i_extendedprice * (1 - i_discount) - ↩
                i_partsupp_supplycost * i_quantity as amount
        from
            part,
            supplier,
            item
        where
```

```
            s_suppkey = i_suppkey
            and p_partkey = i_partkey
            and p_name like '%[COLOR]%'
    ) as profit
group by
    nation,
    order_year
order by
    nation,
    order_year desc;
```

7. Consulta de Relatório de Itens Retornados

Parâmetro de Substituição	Valor
DATE	1993-10-01

```
select
    c_custkey,
    c_name,
    sum(i_extendedprice * (1 - i_discount)) as revenue,
    c_acctbal,
    c_nation_name,
    c_address,
    c_phone,
    c_comment
from
    customer,
    item
where
    c_custkey = i_custkey
    and i_order_orderdate >= date '[DATE]'
    and i_order_orderdate < date '[DATE]' + interval '3' month
    and i_returnflag = 'R'
group by
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    c_nation_name,
    c_address,
    c_comment
order by
    revenue desc;

set rowcount 20
go
```

8. Consulta a Identificação de Estoques Importantes

Parâmetro de Substituição	Valor
NATION	GERMANY
FRACTION	.0001

```
select
   i_partkey,
   sum(i_partsupp_supplycost * i_partsupp_availqty) as value
from
   item,
   supplier
where
   i_suppkey = s_suppkey
   and s_nation_name = '[NATION]'
group by
   i_partkey having
      sum(i_partsupp_supplycost * i_partsupp_availqty) > (
         select
            sum(i_partsupp_supplycost * i_partsupp_availqty) * ↩
               [FRACTION]
         from
            item,
            supplier
         where
            i_suppkey = s_suppkey
            and s_nation_name = '[NATION]'
      )
order by
   value desc;
```

9. Consulta a Modos de Envio e Prioridade de Pedidos

Parâmetro de Substituição	Valor
SHIPMODE1	MAIL
SHIPMODE2	SHIP
DATE	1994-01-01

```
select
   i_shipmode,
   sum(case
      when i_order_orderpriority = '1-URGENT'
         or i_order_orderpriority = '2-HIGH'
         then 1
      else 0
   end) as high_line_count,
   sum(case
```

```
        when i_order_orderpriority <> '1-URGENT'
           and i_order_orderpriority <> '2-HIGH'
           then 1
        else 0
    end) as low_line_count
from
    item
where
    and i_shipmode in ('[SHIPMODE1]', '[SHIPMODE2]')
    and i_commitdate < i_receiptdate
    and i_shipdate < i_commitdate
    and i_receiptdate >= date '[DATE]'
    and i_receiptdate < date '[DATE]' + interval '1' year
group by
    i_shipmode
order by
    i_shipmode;
```

10. Consulta à Distribuição de Clientes

Parâmetro de Substituição	Valor
WORD1	special
WORD2	requests

```
select
    c_count,
    count(*) as custdist
from
    (
        select
            c_custkey,
            count(i_itemkey)
        from
            customer left outer join item on
                c_custkey = i_custkey
                and i_order_comment not like '%[WORD1]%[WORD2]%↩
                    '
        group by
                c_custkey
    ) as c_orders (c_custkey, c_count)
group by
    c_count
order by
    custdist desc,
    c_count desc;
```

11. Consulta aos Efeitos de Promoção

Parâmetro de Substituição	Valor
DATE	1995-09-01

```
select
   100.00 * sum(case
      when p_type like 'PROMO%'
         then i_extendedprice * (1 - i_discount)
      else 0
   end) / sum(i_extendedprice * (1 - i_discount)) as ↩
      promo_revenue
from
   item,
   part
where
   i_partkey = p_partkey
   and i_shipdate >= date '[DATE]'
   and i_shipdate < date '[DATE]' + interval '1' month;
```

12. Consulta à Relação *Part/Supplier*

Parâmetro de Substituição	Valor
BRAND	Brand#45
TYPE	MEDIUM POLISHED
SIZE1	49
SIZE2	14
SIZE3	23
SIZE4	45
SIZE5	19
SIZE6	3
SIZE7	36
SIZE8	9

```
select
    p_brand,
    p_type,
    p_size,
    count(distinct i_suppkey) as supplier_cnt
from
   item,
    part
where
    p_partkey = i_partkey
    and p_brand <> '[BRAND]'
    and p_type not like '[TYPE]%'
    and p_size in ([SIZE1], [SIZE2], [SIZE3], [SIZE4], [SIZE5],↩
        [SIZE6], [SIZE7], [SIZE8])
```

```
    and i_suppkey not in (
        select
            s_suppkey
        from
            supplier
        where
            s_comment like '%Customer%Complaints%'
    )
group by
    p_brand,
    p_type,
    p_size
order by
    supplier_cnt desc,
    p_brand,
    p_type,
    p_size;
go
```

13. Consulta a Grandes Volumes de Clientes

Parâmetro de Substituição	Valor
QUANTITY	300

```
select
    c_name,
    c_custkey,
    i_itemkey,
    i_order_orderdate,
    i_order_totalprice,
    sum(i_quantity)
from
    customer,
    item
where
    i_itemkey in (
        select
            i_itemkey
        from
            item
        group by
            i_itemkey having
                sum(i_quantity) > [QUANTITY]
    )
    and c_custkey = i_custkey
group by
    c_name,
    c_custkey,
```

```
    i_itemkey,
    i_order_orderdate,
    i_order_totalprice
order by
    i_order_totalprice desc,
    i_order_orderdate;
set rowcount 100
go
```

14. Consulta a Desconto de Receita

Parâmetro de Substituição	Valor
QUANTITY1	300
BRAND1	Brand#12
QUANTITY2	10
BRAND2	Brand#23
QUANTITY3	20
BRAND3	Brand#34

```
select
    sum(i_extendedprice* (1 - i_discount)) as revenue
from
    item,
    part
where
    (
        p_partkey = i_partkey
        and p_brand = '[BRAND1]'
        and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM↩
            PKG')
        and i_quantity >= [QUANTITY1] and i_quantity <= [↩
           QUANTITY1] + 10
        and p_size between 1 and 5
        and i_shipmode in ('AIR', 'AIR REG')
        and i_shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p_partkey = i_partkey
        and p_brand = '[BRAND2]'
        and p_container in ('MED BAG', 'MED BOX', 'MED PKG', '↩
           MED PACK')
        and i_quantity >= [QUANTITY2] and i_quantity <= [↩
           QUANTITY2] + 10
        and p_size between 1 and 10
        and i_shipmode in ('AIR', 'AIR REG')
        and i_shipinstruct = 'DELIVER IN PERSON'
    )
```

```
    or
    (
        p_partkey = i_partkey
        and p_brand = '[BRAND3]'
        and p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG↩
            PKG')
        and i_quantity >= [QUANTITY3] and i_quantity <= [↩
           QUANTITY3] + 10
        and p_size between 1 and 15
        and i_shipmode in ('AIR', 'AIR REG')
        and i_shipinstruct = 'DELIVER IN PERSON'
    );
```

15. Consulta a Oportunidades de Vendas Globais

Parâmetro de Substituição	Valor
I1	13
I2	31
I3	23
I4	29
I5	30
I6	18
I7	17

```
select
    cntrycode,
    count(*) as numcust,
    sum(c_acctbal) as totacctbal
from
    (
        select
            substring(c_phone from 1 for 2) as cntrycode,
            c_acctbal
        from
            customer
        where
            substring(c_phone from 1 for 2) in
                ('[I1]', '[I2]', '[I3]', '[I4]', '[I5]', '[I6]'↩
                    , '[I7]')
            and c_acctbal > (
                select
                    avg(c_acctbal)
                from
                    customer
                where
                    c_acctbal > 0.00
                    and substring(c_phone from 1 for 2) in
```

```
                                ('[I1]', '[I2]', '[I3]', '[I4]', '[↩
                                   I5]', '[I6]', '[I7]')
            )
                and not exists (
                    select
                        *
                    from
                        item
                    where
                        i_custkey = c_custkey
                )
    ) as custsale
group by
    cntrycode
order by
    cntrycode;
```

Apêndice C

DDL para Criação do Ambiente *Snowflake*

É apresentado aqui o código em SQL para criação das entidades para o ambiente *snowflake*.

```
CREATE TABLE  region (
    r_regionkey INTEGER,
    r_name CHAR(25),
    r_comment VARCHAR(152)
);

CREATE TABLE  nation (
    n_nationkey INTEGER,
    n_name CHAR(25),
    n_regionkey INTEGER,
    n_comment VARCHAR(152)
);

CREATE TABLE  supplier (
    s_suppkey  INTEGER,
    s_name CHAR(25),
    s_address VARCHAR(40),
    s_nationkey INTEGER,
    s_phone CHAR(15),
    s_acctbal NUMERIC,
    s_comment VARCHAR(101)

);

CREATE TABLE  part (
    p_partkey INTEGER,
    p_name VARCHAR(55),
    p_mfgr CHAR(25),
    p_brand CHAR(10),
    p_type VARCHAR(25),
    p_size INTEGER,
    p_container CHAR(10),
```

```
    p_retailprice NUMERIC,
    p_comment VARCHAR(23)
);

CREATE TABLE  partsupp (
    ps_partkey INTEGER,
    ps_suppkey INTEGER,
    ps_availqty INTEGER,
    ps_supplycost NUMERIC,
    ps_comment VARCHAR(199)
);

CREATE TABLE  customer (
    c_custkey INTEGER,
    c_name VARCHAR(25),
    c_address VARCHAR(40),
    c_nationkey INTEGER,
    c_phone CHAR(15),
    c_acctbal NUMERIC,
    c_mktsegment CHAR(10),
    c_comment VARCHAR(117)
);

CREATE TABLE orders (
    o_orderkey BIGINT,
    o_custkey INTEGER,
    o_orderstatus CHAR(1),
    o_totalprice NUMERIC,
    o_orderdate DATE,
    o_orderpriority CHAR(15),
    o_clerk CHAR(15),
    o_shippriority INTEGER,
    o_comment VARCHAR(79)
);

CREATE TABLE  lineitem (
    l_orderkey BIGINT,
    l_partkey INTEGER,
    l_suppkey INTEGER,
    l_linenumber INTEGER,
    l_quantity NUMERIC,
    l_extendedprice NUMERIC,
    l_discount NUMERIC,
    l_tax NUMERIC,
    l_returnflag CHAR(1),
    l_linestatus CHAR(1),
    l_shipdate DATE,
    l_commitdate DATE,
```

```
    l_receiptdate DATE,
    l_shipinstruct CHAR(25),
    l_shipmode CHAR(10),
    l_comment VARCHAR(44)
);
```

Apêndice D

DDL para Criação do Ambiente *Star*

É apresentado aqui o código em SQL para criação das entidades para o ambiente *star*, adaptado do *snowflake*.

```
CREATE TABLE supplier (
    s_suppkey INTEGER NOT NULL,
    s_name CHAR(25) NOT NULL,
    s_address VARCHAR(40) NOT NULL,
    s_nationkey INTEGER NOT NULL,
    s_nation_name CHAR(25) NOT NULL,
    s_region_name CHAR(25) NOT NULL,
    s_phone CHAR(15) NOT NULL,
    s_acctbal DECIMAL(15,2) NOT NULL,
    s_comment VARCHAR(101) NOT NULL,
    s_nation_comment VARCHAR(152),
    s_region_comment VARCHAR(152)
);

CREATE TABLE customer (
    c_custkey INTEGER NOT NULL,
    c_name CHAR(25) NOT NULL,
    c_address VARCHAR(40) NOT NULL,
    c_nationkey INTEGER NOT NULL,
    c_nation_name CHAR(25) NOT NULL,
    c_region_name CHAR(25) NOT NULL,
    c_phone CHAR(15) NOT NULL,
    c_acctbal DECIMAL(15,2) NOT NULL,
    c_mktsegment  CHAR(10) NOT NULL,
    c_comment VARCHAR(117) NOT NULL,
    c_nation_comment VARCHAR(152),
    c_region_comment VARCHAR(152)
);

CREATE TABLE part (
```

```
    p_partkey     INTEGER NOT NULL,
    p_name        VARCHAR(55) NOT NULL,
    p_mfgr        CHAR(25) NOT NULL,
    p_brand       CHAR(10) NOT NULL,
    p_type        VARCHAR(25) NOT NULL,
    p_size        INTEGER NOT NULL,
    p_container   CHAR(10) NOT NULL,
    p_retailprice DECIMAL(15,2) NOT NULL,
    p_comment     VARCHAR(23) NOT NULL
);

CREATE TABLE item (
    i_itemkey INTEGER NOT NULL,
    i_partkey INTEGER NOT NULL,
    i_suppkey INTEGER NOT NULL,
    i_custkey INTEGER NOT NULL,
    i_linenumber INTEGER NOT NULL,
    i_quantity DECIMAL (15,2) NOT NULL,
    i_extendedprice DECIMAL (15,2) NOT NULL,
    i_discount DECIMAL (15,2) NOT NULL,
    i_tax DECIMAL (15,2) NOT NULL,
    i_returnflag char(1) not null,
    i_linestatus char(1) not null,
    i_shipdate date not null,
    i_commitdate date not null,
    i_receiptdate date not null,
    i_shipinstruct char(25) not null,
    i_shipmode char(10) not null,
    i_comment varchar(44) not null,
    i_orderstatus char(1) not null,
    i_ordertotalprice decimal(15,2) not null,
    i_orderdate date not null,
    i_orderpriority char(15) not null,
    i_orderclerk char(15) not null,
    i_shippriority integer not null,
    i_ordercomment varchar(79) not null,
    i_availqty integer not null,
    i_supplycost decimal(15,2)  not null,
    i_partsuppcomment varchar(199) not null
);
```

Referências Bibliográficas

[1] WREMBEL, R.; KONCILIA, C. *Data warehouses and OLAP: concepts, architectures, and solutions*. [S.l.]: Igi Global, 2007.

[2] CODD, E.; CODD, S.; SALLEY, C. Providing olap (on-line analytical processing) to user-analysis: An it mandate. *White paper*, 1993.

[3] CHAUDHURI, S.; DAYAL, U. An overview of data warehousing and olap technology. *ACM Sigmod record*, ACM, v. 26, n. 1, p. 65–74, 1997.

[4] KIMBALL, R.; ROSS, M. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. 2. ed. [S.l.]: Wiley Computer Publishing, 2002.

[5] ELMASRI, R.; NAVATHE, S. B. *Sistemas de Banco de Dados*. 6. ed. [S.l.]: Pearson Education do Brasil, 2011.

[6] Good Data Help. *Column Storage and Compression in Data Warehouse*. 2017. Acessado em: 12 de maio de 2017. Disponível em: <https://goo.gl/f1yeh8>.

[7] BOUCKAERT, S. et al. Benchmarking computers and computer networks. *EU FIRE White Paper*, 2010.

[8] TRANSACTION PROCESSING PERFORMANCE COUNCIL. *TPC Homepage*. 2017. Acessado em: 12 de agosto de 2017. Disponível em: <http://www.tpc.org>.

[9] TRANSACTION PROCESSING PERFORMANCE COUNCIL. *TPC-H Homepage*. 2017. Acessado em: 19 de maio de 2017. Disponível em: <http://www.tpc.org/tpch/>.

[10] BAX, M. P.; SOUZA, R. Modelagem estratégica de dados: Normalização versus "dimensionalização". *KMBRASIL, Anais... São Paulo: SBGC*, 2003.

[11] INMON, W. H. *Building the data warehouse*. [S.l.]: John wiley & sons, 2005.

[12] SHIM, J. P. et al. Past, present, and future of decision support technology. *Decision support systems*, Elsevier, v. 33, n. 2, p. 111–126, 2002.

[13] ZELEN, A. *OLTP vs. OLAP — What's the Difference?* 2017. `https://academy.vertabelo.com/blog/oltp-vs-olap-whats-difference/`. Acessado em: 12 de abril de 2018.

[14] SEN, A.; SINHA, A. P. A comparison of data warehousing methodologies. *Communications of the ACM*, ACM, v. 48, n. 3, p. 79–84, 2005.

[15] LEVENE, M.; LOIZOU, G. Why is the snowflake schema a good data warehouse design? *Information Systems*, Elsevier, v. 28, n. 3, p. 225–240, 2003.

[16] CODD, E. F. A relational model of data for large shared data banks. *Communications of the ACM*, ACM, v. 13, n. 6, p. 377–387, 1970.

[17] POSTGRESQL: The World's Most Advanced Open Source Relational Database. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://www.postgresql.org/>.

[18] MYSQL: The world's most popular open source database. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://www.mysql.com/>.

[19] MICROSOFT SQL Server. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://www.microsoft.com/en-us/sql-server/sql-server-2017>.

[20] MICROSOFT SQL Server. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://www.sqlite.org/index.html>.

[21] MARIADB - Supporting continuity and open collaboration. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://mariadb.org/>.

[22] DATABASE and Cloud Database - Oracle. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://www.oracle.com/database/index.html>.

[23] HAN, J. et al. Survey on nosql database. In: *2011 6th International Conference on Pervasive Computing and Applications*. [S.l.: s.n.], 2011. p. 363–366.

[24] PRITCHETT, D. Base: An acid alternative. *Queue*, ACM, v. 6, n. 3, p. 48–55, 2008.

[25] SHARDING or Data Partitioning. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://www.educative.io/collection/page/5668639101419520/5649050225344512/5146118144917504>.

[26] TWITTER growth prompts switch from MySQL to 'NoSQL' database. 2010. Acessado em: 20 de maio de 2018. Disponível em: <https://www.computerworld.com/article/2520084/database-administration/twitter-growth-prompts-switch-from-mysql-to–nosql–database.html>.

[27] CHANG, F. et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 26, n. 2, p. 4, 2008.

[28] DECANDIA, G. et al. Dynamo: amazon's highly available key-value store. In: ACM. *ACM SIGOPS operating systems review*. [S.l.], 2007. v. 41, n. 6, p. 205–220.

[29] THE Neo4j Graph Plataform - The #1 Platform for Connected Data. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://neo4j.com/>.

[30] PROJECT Voldemort: A distributed database. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://www.project-voldemort.com/voldemort/>.

[31] AMAZON DynamoDB - NoSQL Database Service. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://aws.amazon.com/dynamodb/>.

[32] RIAK. 2018. Acessado em: 20 de maio de 2018. Disponível em: <http://basho.com/products/>.

[33] REDIS. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://redis.io/>.

[34] MONGODB for Giant Ideas. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://www.mongodb.com/>.

[35] APACHE CouchDB. 2018. Acessado em: 20 de maio de 2018. Disponível em: <http://couchdb.apache.org/>.

[36] MONETDB – The column-store pioneer. 2017. Acessado em: 20 de agosto de 2017. Disponível em: <https://www.monetdb.org/Home>.

[37] C-STORE: A Column-Oriented DBMS. 2018. Acessado em: 20 de maio de 2018. Disponível em: <http://db.lcs.mit.edu/projects/cstore/>.

[38] BIGTABLE - Scalable NoSQL Database Service. 2018. Acessado em: 20 de maio de 2018. Disponível em: <https://cloud.google.com/bigtable/>.

[39] APACHE Cassandra. 2018. Acessado em: 20 de maio de 2018. Disponível em: <http://cassandra.apache.org/>.

[40] BREWER, E. A. Towards robust distributed systems. In: *PODC*. [S.l.: s.n.], 2000. v. 7.

[41] GILBERT, S.; LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, ACM, v. 33, n. 2, p. 51–59, 2002.

[42] KHOSHAFIAN, S. et al. A query processing strategy for the decomposed storage model. In: IEEE. *Data Engineering, 1987 IEEE Third International Conference on*. [S.l.], 1987. p. 636–643.

[43] MATEI, G.; BANK, R. C. Column-oriented databases, an alternative for analytical environment. *Database Systems Journal*, Academy of Economic Studies-Bucharest, Romania, v. 1, n. 2, p. 3–16, 2010.

[44] ABADI, D. J.; MADDEN, S. R.; HACHEM, N. Column-stores vs. row-stores: how different are they really? In: ACM. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. [S.l.], 2008. p. 967–980.

[45] ABADI, D. et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, Now Publishers, Inc., v. 5, n. 3, p. 197–280, 2013.

[46] ABADI, D. J. et al. Materialization strategies in a column-oriented dbms. In: IEEE. *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. [S.l.], 2007. p. 466–475.

[47] ABADI, D. J. *Query execution in column-oriented database systems*. Tese (Doutorado) — Massachusetts Institute of Technology, 2008.

[48] STONEBRAKER, M. et al. C-store: a column-oriented dbms. In: VLDB ENDOWMENT. *Proceedings of the 31st international conference on Very large data bases*. [S.l.], 2005. p. 553–564.

[49] BONCZ, P. A.; ZUKOWSKI, M.; NES, N. Monetdb/x100: Hyper-pipelining query execution. In: *Cidr*. [S.l.: s.n.], 2005. v. 5, p. 225–237.

[50] WESTMANN, T. et al. The implementation and performance of compressed databases. *ACM Sigmod Record*, ACM, v. 29, n. 3, p. 55–67, 2000.

[51] ROTH, M. A.; HORN, S. J. V. Database compression. *ACM Sigmod Record*, ACM, v. 22, n. 3, p. 31–39, 1993.

[52] ABADI, D.; MADDEN, S.; FERREIRA, M. Integrating compression and execution in column-oriented database systems. In: ACM. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. [S.l.], 2006. p. 671–682.

[53] BHUTTA, K. S.; HUQ, F. Benchmarking–best practices: an integrated approach. *Benchmarking: An International Journal*, MCB UP Ltd, v. 6, n. 3, p. 254–268, 1999.

[54] KYRÖ, P. Revising the concept and forms of benchmarking. *Benchmarking: An International Journal*, MCB UP Ltd, v. 10, n. 3, p. 210–225, 2003.

[55] NGAMSURIYAROJ, S.; PORNPATTANA, R. Performance evaluation of tpc-h queries on mysql cluster. In: IEEE. *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on*. [S.l.], 2010. p. 1035–1040.

[56] NADEE, W.; NGAMSURIYAROJ, S. Performance evaluation of tpc-h queries on java ee cluster. In: IEEE. *Computer Science and Software Engineering (JCSSE), 2012 International Joint Conference on*. [S.l.], 2012. p. 385–390.

[57] THANOPOULOU, A.; CARREIRA, P.; GALHARDAS, H. Benchmarking with tpc-h on off-the-shelf hardware. *ICEIS (1)*, p. 205–208, 2012.

[58] BARATA, M.; BERNARDINO, J.; FURTADO, P. Ycsb and tpc-h: Big data and decision support benchmarks. In: IEEE. *Big Data (BigData Congress), 2014 IEEE International Congress on*. [S.l.], 2014. p. 800–801.

[59] RUTISHAUSER, N.; NOURELDIN, A. Tpc-h applied to mongodb: How a nosql database performs. Feb, 2012.

[60] SOARES, B. E. Uma avaliação experimental de desempenho entre sistemas gerenciadores de bancos de dados colunares e relacionais. 2012.

[61] TRANSACTION PROCESSING PERFORMANCE COUNCIL. *TPC BENCHMARK H Standard Specification. Revision 2.17.2*. San Francisco, 2017. Acessado em: 12 de maio de 2017. Disponível em: <https://goo.gl/uaRRcv>.

[62] RAASVELDT, M. et al. Fair benchmarking considered difficult: Common pitfalls in database performance testing. 2018.