UNIVERSIDADE FEEVALE

CRISTIANO HORN

ANÁLISE TÉCNICA E APLICAÇÃO DE BANCO DE DADOS EM MEMÓRIA

Novo Hamburgo

2016

CRISTIANO HORN

ANÁLISE TÉCNICA E APLICAÇÃO DE BANCO DE DADOS EM MEMÓRIA

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel em Sistemas de Informação pela Universidade Feevale

Orientador: Me. Edvar Bergmann Araujo

Novo Hamburgo

2016

AGRADECIMENTOS

Gostaria de agradecer a todos os que, de alguma maneira, contribuíram para a realização desse trabalho de conclusão, em especial:

Aos amigos, namorada e às pessoas que convivem comigo diariamente, colegas de trabalho, ao professor orientador, aos demais professores que acompanharam toda minha vida acadêmica, minha gratidão, pelo apoio emocional nos períodos mais difíceis do trabalho.

RESUMO

Atualmente, várias empresas, dentre as quais Facebook, Google, Amazon, E-Bay, vem tendo um aumento exponencial na quantidade dos dados, provenientes de diversas origens. Isso acarreta em um problema de armazenamento, gerenciamento e análise dessas bases de dados gigantescas, principalmente no que diz respeito a tomadas de decisão e BI *(Business Inteligence)*, que já trabalham com *terabytes* e *petabytes* de dados. Além disso, processos como previsão do tempo, análises financeiras, análises de mercado, inteligência artificial, necessitam do processamento de grandes volumes dados em tempo real. Estes e outros requisitos excedem a capacidade de processamento dos sistemas tradicionais de gerenciamento de banco de dados em disco, ocasionando ociosidade funcional para gerenciar tais dados e dar rápidos resultados em tempo real. Dessa forma, o gerenciamento de dados precisa de novas soluções para lidar com os desafios de volumes de dados e processamento de dados em tempo real, e que busquem recursos que possam melhorar o seu desempenho, escalabilidade e disponibilidade. Um sistema de banco de dados em memória ou *In-memory Database System* (IMDS) é a mais recente geração de sistema de gerenciamento de banco de dados e está se tornando a resposta aos desafios que vem surgindo neste meio. Esse tipo de banco de dados é capaz de processar dados massivos de forma mais rápida, abolindo a necessidade de esperar horas ou até dias para que se tenham condições de tomar uma decisão baseada na análise de dados. Este trabalho visa analisar tecnicamente o funcionamento e desempenho de um IMDS, realizando uma comparação entre o IMDS da Oracle em relação a um banco de dados tradicional.

Palavras-chave: Banco de dados em memória. *In-memory databases*. Sistemas Gerenciadores de Banco de Dados. Computação em memória.

ABSTRACT

Currently, several companies, like Facebook, Google, Amazon, E-Bay, has had an exponential increase in the amount of data from several sources. This leads to a problem of storage, management, and analysis of massive databases, especially about decision-making and Business Intelligence, who already work with terabytes and petabytes of data. In addition, processes such as weather forecasting, financial analysis, market analysis, artificial intelligence, require the processing of large data volumes in real time. These and other requirements exceed the processing capacity of traditional disk database management systems, resulting in functional idleness for managing such data to provide fast results in real time. In this way, the data management needs new solutions to address the challenges of data volumes and data processing in real time, and to seek resources that can improve their performance, scalability, and availability. An In-Memory Database System is the latest generation of database management system and is becoming the answer to the challenges that are emerging in this environment. This type of database can process massive data faster, abolishing the need to wait hours or even days to have been able to decide based on data analysis. This study aims to technically analyze the operation and performance of a IMDS performing a comparison between the IMDS Oracle with a traditional database.

Keywords: In-memory databases. Database Management Systems. In-memory computing.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicidade, Consistência, Isolamento, Durabilidade
API	Application Programming Interface
BI	Business Intelligence
BA	Business Analytics
CPU	Central Processing Unit
CRUD	Create, Read, Update e Delete
DBA	DataBase Adminitrator
DIMM	Dual Inline Memory Module
DML	Data Manipulation Language
DRAM	Dynamic Random-Access Memory
EEPROM	Electrically-Erasable Programmable Read-Only Memory
ETL	Extract, Transform and Load
HD	Hard Disk
IMDB	In-Memory Database Base
IMDS	In-Memory Database System
IP	Internet Protocol
LRU	Least Recently Used
MMDB	Main Memory DataBase
NVDIMM	DIMM Non-Volatile
NVRAM	Non-volatile Random Access Memory
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
RAM	Random-Access Memory
RDBMS	Relational Database Managment System
RLE	Run Length Encoding
ROM	Read-Only Memory
SGDB	Sistema de Gerenciamento de Banco de Dados
SIMD	Single Instruction, Multiple Data
SQL	Structured Query Language
SSD	Solid-State Drive
TB	Terabyte

TCP	Transmission Control Protocol
XML	eXtensible Markup Language
WAN	Wide Area Network
ZB	Zettabyte

SUMÁRIO

INTRODUÇÃO

Com a crescente demanda de dados e a necessidade de processamento de dados em tempo real, sistemas tradicionais de gerenciamento de dados, ou seja, aqueles que armazenam seus dados em disco, vêm sofrendo pressão para melhorar seu desempenho, a fim de atender essa demanda. É esperado que se atinja 40ZB (1ZB = 1 bilhão de *terabytes*) até 2020, isso significa 5247 GB de dados por pessoa (IDC DIGITAL UNIVERSE STUDY, 2012).

Para bancos de dados em disco, operações de I/O (entrada e saída de dados) são o principal gargalo de desempenho, pois são muito lentas e não podem ser otimizadas além daquilo que provém da sua natureza mecânica (GUPTA; VERMA; VERMA, 2013). Ainda é muito comum a utilização de armazenamento de dados em disco nas empresas atuais. Existe um grande esforço na tentativa de melhorar o desempenho dessas operações utilizando técnicas de cache para armazenar dados acessados com mais frequência, trazendo, com essa ação, um benefício performático considerável. Em contrapartida a essa melhoria, existe um custo de sincronização de cache com o disco e vice-versa, além da implementação da lógica complexa para gerenciar transações e recursos. Percebe-se assim que tais artefatos colocam uma barreira nesse processo, limitando novamente o desempenho dos SGDBs convencionais.

Para dar suporte a esse massivo crescimento dos dados, emerge o conceito dos bancos de dados em memória, algo que realmente modifica todo o paradigma dos sistemas de gerenciamento de banco de dados. Conforme Garcia-Molina e Salem (1992), um sistema de banco de dados *in-memory* (IMDB), ou sistema de banco de dados em memória principal, é um tipo de banco de dados que armazena dados totalmente na memória principal em vez de manter no disco. Com a diminuição dos custos da memória principal e do avanço das inovações tecnológicas, torna-se viável o armazenamento de uma grande quantidade de dados na memória principal.

Nesse tipo de arquitetura, a velocidade de leitura e escrita dos dados é extremamente melhoradas, pois eliminam-se as operações de I/O. Essa grande diferença traz como benefício o processamento de dados em tempo real e, portanto, o gerenciamento e processamento de grande quantidade de dados. Além disso, uma vez que todos os dados estão na memória principal, não há necessidade de implementar lógicas de cache complexas e, dessa forma, a sobrecarga que havia em um sistema de banco de dados convencional também é eliminada.

Em uma abordagem mais técnica, o projeto de um IMDS é todo voltado para usar a memória principal para armazenamento de dados, em vez de usar discos. Todo o sistema é

projetado para ter alto desempenho no acesso, manipulação e análise dos dados. Em um IMDS, o foco em otimizar algoritmos para acesso e manipulação de dados em disco é alterado para otimização de algoritmos e estrutura de dados para obter um melhor acesso e utilização da memória principal. Além disso, a otimização de consultas agora está focada em melhorar a estrutura e algoritmos de dados na memória no momento de executar a consulta, em vez de melhorar o tipo de operação I/O, que antes tinha limitações mecânicas.

A sobrecarga para gerenciar as transações simultâneas é menor, isso porque o acesso aos dados é muito mais rápido e, portanto, os bloqueios são liberados mais rapidamente. Uma vez que todos os dados são carregados na memória principal, dados distribuídos em nós de gerenciamento também podem usar a memória principal compartilhada localmente ou uma rede WAN de alta velocidade para que se permita virtualmente um local de dados e acesso a esses dados, de forma mais rápida (GUPTA; VERMA; VERMA, 2013).

Quando da utilização de um sistema de banco dados tradicional e da necessidade de análises em um grande volume de dados, é necessário o uso de um *Data Warehouse*, havendo assim a necessidade de aplicar um processo de ETL *(Extract, Transform and Load)* sobre esses dados. Essa ação normalmente acontece durante a noite, utilizando a informação do dia anterior, podendo não reproduzir com fidelidade o resultado que o negócio ou organização necessita.

Conforme documentação da ORACLE (2015a), os dados que são otimizados para consulta são comprimidos e organizados em um formato de colunas (quando definido dessa forma), o que acelera tanto a recuperação sequencial, quanto aleatória de dados. O formato de compressão é projetado para permitir um processamento em "vetor", usando uma única instrução e dados múltiplos de operações (SIMD – *Single Instruction, Multiple Data*) que são usados para reduzir ainda mais o número de instruções de máquina necessárias para encontrar os dados desejados.

> Os bancos de dados de memória organizam seus dados em formato orientado a linha (para processamento de transações) ou em formato orientado a coluna (para o processamento de consultas). Muito poucos são organizados de tal forma a apoiar tanto transações rápidas quanto processamento de consultas com os mesmos dados. (Oracle, 2015, p. 3, tradução nossa).

Analisando um dos bancos de dados em memória disponível no mercado, observou-se que o Oracle TimesTen é projetado para trabalhar tanto com o formato em linha (*row-store*) quanto com o formato em coluna (*column-store*). Nesse IMDB, é suportada a compressão colunar quando o armazenamento é feito dessa forma, o que oferece como benefício, uma ajuda em acomodar grandes conjuntos de dados, permitindo assim cargas de trabalhos OLAP

(processamento analítico online) e BI (*Business Inteligence*). Além disso, estando todos os dados na memória, índices não precisam mais armazenar valores-chave, resultando assim em uma significativa economia de espaço (LAHIRI; NEIMAT; FOLKMAN, 2013). Quando um SGDB tem seus dados armazenados em linha, existirão em uma mesma linha diferentes tipos (domínios), tornando o processo mais complicado. Já no banco orientado a colunas, cada coluna irá conter o mesmo tipo (domínio) de dado, facilitando a busca e otimizando o resultado (ABADI; MADDEN; HACHEM, 2008).

O TimesTen foi projetado para suportar completamente as propriedades atomicidade, consistência, isolamento e durabilidade (ACID). Mesmo que o TimesTen opere em dados que estão armazenados na memória principal, esses dados são persistentes e recuperáveis em caso de falha de software, hardware ou falta de energia (LAHIRI; NEIMAT; FOLKMAN, 2013).

A durabilidade é garantida através de pontos de verificação de registros no disco, algo que normalmente é um gargalo em arquiteturas *multi-core*. Para esse propósito, a ferramenta trabalha com a gravação de dados utilizando um mecanismo *multi-threads*, gerando logs não serializados. Nesse mecanismo, o *buffer log* é dividido em múltiplas partições que podem ser populadas em paralelo por processos concorrentes. Dessa forma, se uma transação é concluída com sucesso, seus efeitos serão persistentes, mesmo que o sistema sofra uma queda antes que esses dados sejam persistidos no disco.

A alta disponibilidade é proporcionada através da replicação, onde se utiliza um banco de dados ativo, em que os dados são alterados e replicados para um banco de dados que é somente para leitura. Se ocorrer uma falha no banco de dados que é dedicado à leitura, ele passa a utilizar o ativo, já se ocorrer uma falha no banco de dados ativo, um banco de dados de leitura é indicado para se tornar o ativo. O isolamento é garantido por padrão no TimesTen, onde o bloqueio é feito a nível de linha e não de tabela inteira, favorecendo assim a performance do banco de dados para operações de escrita e consulta simultânea (LAHIRI; NEIMAT; FOLKMAN, 2013).

O banco de dados Oracle TimesTen fornece suporte à linguagem SQL e PL/SQL, acesso via API’s de banco de dados padrão e contempla extensões para aplicativos de análise de dados massivo. O TimesTen pode ser usado como um único banco de dados de registro para aplicativo ou ser usado com uma configuração de cache IMDB que permite que ele seja implantado como um cache transacional persistente para dados de um banco de dados Oracle.

Este trabalho de conclusão de curso visa apresentar um aprofundamento sobre bancos de dados em memória com base nos conceitos e tecnologias citados, apresentando as principais técnicas e otimizações para armazenamento e processamento de grandes volumes de dados em memória e fazendo comparações de desempenho entre SGDB's tradicionais e em memória.

No primeiro capítulo deste trabalho, é feito um estudo dos conceitos que envolvem *In-Memory Databases* e um comparativo entre algumas características e funcionalidades relativas ao Oracle TimesTen e SAP HANA. O segundo capítulo apresenta um aprofundamento técnico da ferramenta Oracle TimesTen, descrevendo suas principais características e aplicabilidades. O terceiro capítulo introduz a parte prática do trabalho, indicando o propósito dos experimentos, o ambiente de testes e a base de dados a ser estudada, bem como a realização da carga de dados no ambiente de testes. O capítulo quatro apresenta os experimentos realizados utilizando um SGBD Oracle tradicional, com funcionamento baseado em disco, e com o banco de dados em memória da Oracle, o Oracle TimesTen. Ao final, são indicadas as conclusões obtidas neste trabalho.

1. BANCO DE DADOS EM MEMÓRIA

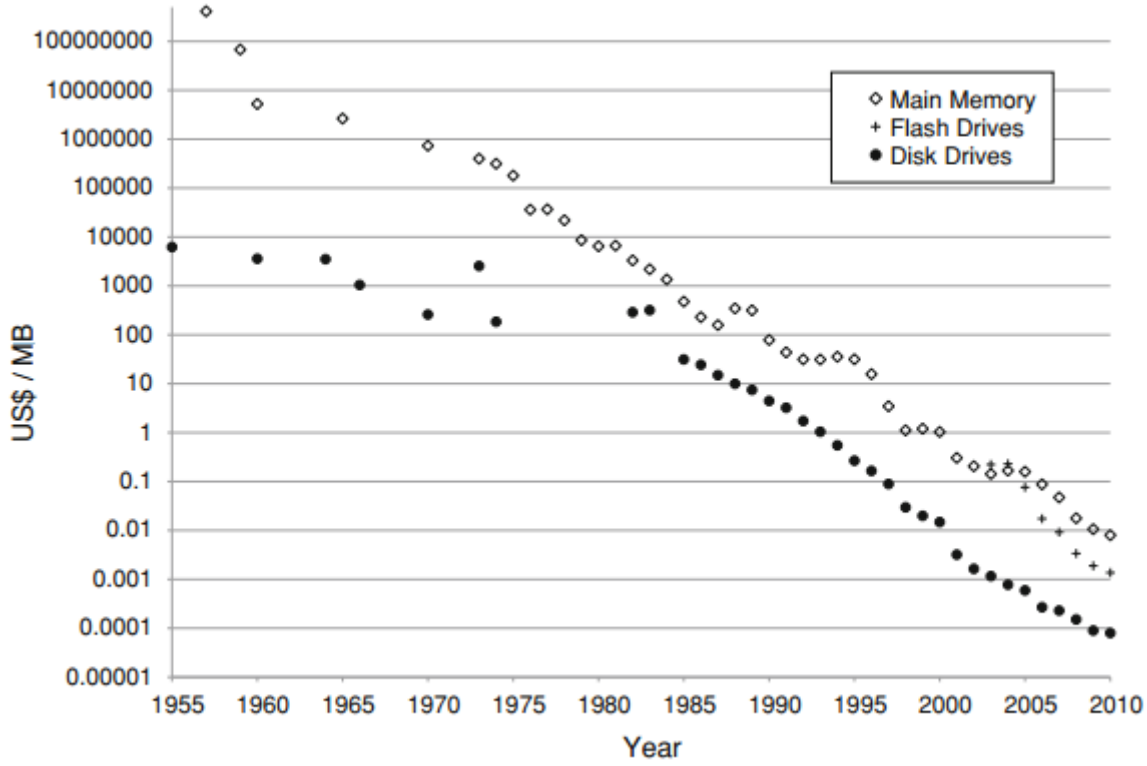
Há anos que a relação entre banco de dados tradicionais e a gravação de dados em disco tem tido uma dependência mútua. Com a era do *Big Data*, surgiram oportunidades para as companhias utilizarem informações captadas e armazenadas para auxiliar na tomada de decisões, promovendo uma vantagem competitiva para a organização.

A informação vem ganhando importância e se tornando um dos principais pilares para o sucesso e crescimento de uma organização. Porém, com o crescimento exponencial das informações geradas por pessoas, softwares e outros meios, os sistemas estão ficando sobrecarregados. Isto gera atrasos ou longo tempo de espera para que esses dados se transformem em informação e possam ser consultados, podendo ocasionar a análise imprecisa ou incorreta de um conjunto de informações. Esse gargalo de tempo é um problema que pode ser sanado através da tecnologia *in-memory*.

A tecnologia *in-memory* tem sido apontada como a solução para problemas de desempenho de banco de dados. O fator principal é a capacidade para carregar e executar todos os dados na memória. Isso remove uma quantidade considerável de entrada/saída (E/S), que causa problemas de desempenho em sistemas de banco de dados. No entanto, as tecnologias IMDB apresentam um risco fundamental que deve ser considerado na implementação: durabilidade dos dados.

A utilização da memória como principal periférico de armazenamento nos bancos de dados vem se tornando possível principalmente devido a maior capacidade de armazenamento e à diminuição dos custos da memória RAM. A Figura 1 demonstra a evolução dos valores relacionados a custos em US$ de armazenamento de 1955 até 2010. Isso demonstra que sistemas de gerenciamento de banco de dados podem ser implantados a um custo acessível.

Figura 1 – Evolução dos preços de armazenamento



Fonte: HPI, 2015

1.1 DEFINIÇÃO

Um IMDB, também conhecido como um banco de dados de memória principal ou *Main Memory DataBase* (MMDB), é um banco de dados cujos dados são armazenados principalmente na memória principal do servidor ao invés de serem gravados no disco como acontece em SGDBs tradicionais. Essa nova forma de armazenamento permite a obtenção de respostas mais rápidas em consultas e análises de informações.

Segundo a Oracle (2014a), comparado a um RDBMS totalmente em cache, a tecnologia IMDB requer muito menos poder de processamento, pois a sobrecarga de gerenciar *buffers* de memória e gerenciar vários locais de dados (disco e memória) é eliminada. Nesse tipo de solução, discos são usados para a persistência e recuperação de dados, e não como o local de armazenamento do banco de dados primário.

Nesse tipo de SGDB, a base de dados é carregada na memória principal do sistema em um formato compactado. Além de fornecer tempos de reposta de consulta extremamente rápidos, análises de dados podem reduzir ou até eliminar a necessidade de indexação de dados e armazenamento de dados pré-carregados em cubos OLAP ou tabelas agregadas, reduzindo

assim os custos de aplicação e provendo uma implementação mais rápida de *Business Inteligence* (BI) e *Business Analytics* (BA) (PC MAGAZINE, 2013).

Uma vez que todos os dados são carregados na memória principal e todas as transações são executadas na memória, se tem o benefício da capacidade de executar operações de E/S inteiramente na memória. Para fins de comparação, pode-se tomar como exemplo um caso em que uma pessoa que memoriza o dicionário inteiro pode responder mais rapidamente a uma consulta da definição de uma palavra do que uma pessoa que não memorizou o dicionário inteiro, e tem que procurar a palavra em um livro impresso.

Quando aplicações são baseadas em dados, existem gargalos de processamento que podem prejudicar o sistema ou negócio, pois a alta interação dos dados e a leitura no disco causa um tempo de espera considerável para o cliente, algo que pode ser indesejável para o negócio (ISACA JOURNAL, 2013).

1.2 DURABILIDADE DAS TRANSAÇÕES

Segundo Prichett (2008), atomicidade, consistência, isolamento e durabilidade (ACID) são propriedades de conformidade que pressupõem que as transações de banco de dados são executadas de forma confiável. Para uma base de dados processar transações de forma íntegra, ela necessita possuir esse conjunto de propriedades.

Conforme o Isaca Journal (2013), a durabilidade costuma variar em implementações de IMDB. A maioria das soluções deste tipo (ex.: SAP HANA, Oracle TimesTen, VMware Gemfire, MySQL Cluster, VoltDB, SQLite) está em conformidade com a ACID. No entanto, elas geralmente variam quando se trata de durabilidade no disco. A “preguiça” da gravação no disco determinará isto. Conforme é abordado no decorrer deste trabalho, algumas soluções (ex.: Oracle TimesTen) permitem que os desenvolvedores ajustem a “preguiça” da gravação no disco.

Em um IMDB, os dados são armazenados em memória volátil, ou seja, uma memória temporária que precisa de energia para se manter ativa e preservar os dados armazenados. Segundo Narayanan e Hodson (2012), a durabilidade, principalmente no caso de um sistema em memória, pode não ser garantida em sua totalidade. Para resolver este problema, os bancos de dados em memória têm adicionado durabilidade através de outro mecanismo.

Snapshot Files, ou imagens de *checkpoint*, registram o estado do banco de dados em um determinado momento no tempo. O sistema normalmente gera estes arquivos periodicamente, ou pelo menos quando o IMDB faz um desligamento controlado. No entanto, eles não garantem a persistência total de dados, uma vez que mudanças mais recentes serão perdidas, provendo assim uma durabilidade parcial. Conforme é citado por Narayanan e Hodson (2012), a fim de garantir a durabilidade completa, os arquivos *snapshot* precisam ser completados com um dos seguintes itens:

- ***Transaction logging***: este mecanismo registra as alterações do banco de dados em um arquivo físico, facilitando a recuperação automática de um banco de dados *in-memory*;
- ***Non-Volatile* DIMM (NVDIMM)**: um módulo de memória que tem uma interface DRAM, muitas vezes combinados com uma memória *flash*, que é um dispositivo semelhante a memória RAM, porém que não precisa de energia para manter os dados armazenados, para assegurar que os dados não sejam perdidos. Com este armazenamento, IMDBs podem retornar de forma segura a partir ao seu estado original após a reinicialização;
- ***Non-volatile random access memory* (NVRAM)**: geralmente na forma de RAM estática apoiada com a energia da bateria RAM, ou um ROM programável passível de limpeza de dados de forma eletrônica (EEPROM), que é um tipo de memória não volátil que não precisa de energia para armazenar dados. Com este armazenamento, o sistema IMDB pode recuperar o armazenamento de dados a partir de seu último estado consistente após ter sido reiniciado.

Sendo a memória principal volátil, a perda de dados é especialmente prejudicial para aplicações orientadas a dados. No entanto, a maioria das soluções *In-Memory* tem um mecanismo para assegurar que os dados sejam preservados, e o mecanismo mais comum é gravar novamente os dados no armazenamento persistente.

No entanto, isso exige a dependência de discos causando lentidão. A fim de contornar esse problema, a maioria das soluções do mercado usa uma solução baseada na gravação em cache e na memória principal “preguiçosa” ou “imprecisa”. Isso significa que a execução da transação é feita inteiramente nos dados armazenados na memória. Essas transações são armazenadas na forma de um *buffer* de log, que também está na memória e o sistema irá gravar os dados em disco para persistência. No caso de falta de energia, há uma chance de perda de

dados se o *buffer* de log não conseguiu completar a gravação em disco. No entanto, a maior parte do banco de dados estará intacta. (NARAYANAN; HODSON, 2012)

Segundo o Isaca Journal (2013), essa limitação é a razão pela qual as implementações de IMDB de alta disponibilidade geralmente pedem o uso de replicação, pois a taxa de transferência da rede ainda é geralmente mais rápida do que a do disco. Essa replicação permite que várias instâncias de IMDB sincronizem os dados contidos no sistema.

1.3 LAYOUT DE ARMAZENAMENTO

Segundo a Oracle (2015b), o formato de armazenamento orientado a linha garante que cada novo registro seja armazenado na base de dados como uma nova linha representado em uma tabela. Essa linha é composta de várias colunas e cada uma representa um atributo diferente deste registro, sendo esse o formato ideal para sistemas OLTP.

A Oracle (2015b) também define que um banco de dados com armazenamento colunar, armazena cada um dos atributos de um registro em uma estrutura de coluna separada. Esse formato é o ideal para análise de dados, pois permite a recuperação de dados de forma mais rápida, principalmente quando somente algumas colunas são selecionadas em um grande conjunto de dados.

Mas o que acontece quando uma operação DML (*insert, update ou delete*) ocorre em cada formato? Um formato de linha é eficiente para o processamento DML pois manipula um registro inteiro em uma única operação, ou seja, para inserir, atualizar ou excluir um registro, é necessário apenas uma instrução. Um formato de coluna não é tão eficiente no processamento destas instruções, pois ao inserir ou excluir um único registro, todas as estruturas na tabela devem ser alteradas (ORACLE, 2015b).

Por exemplo, na comparação entre dados de dois clientes, todos os atributos desses dois clientes necessitam ser carregados para que seja executada a operação de comparação. Em contrapartida, bases de dados orientadas a coluna se beneficiam do seu formato de armazenamento de dados quando um subconjunto dos atributos necessita ser processado para todas, ou para um numeroso conjunto de entradas da tabela da base de dados. Neste caso, a soma do número total de todos os produtos vendidos em determinada época do ano, envolve somente os atributos de data e valor, ignorando todos os outros como Id do produto, descrição, estoque, etc. Usar o armazenamento em linha para esse propósito resultaria no processamento

de todos os atributos da tabela, mesmo que somente dois atributos fossem requisitados. Sendo assim, incorporar um armazenamento em coluna traz o benefício de acessar somente os dados relevantes, além de usar menos operações de procura por identificadores de tupla (*search skipping operations*) (HPI, 2015).

A Figura 2 representa o layout de armazenamento mais comum em bancos de dados tradicionais, onde os dados são armazenados de forma linear e as colunas com as informações são posicionadas uma ao lado da outra. A Figura 3 apresenta armazenamento colunar, onde os dados são armazenados em forma de coluna e identificados por índices que facilitam as queries na sumarização das informações.

Figura 2 – Armazenamento orientado a linha

ORIENTAÇÃO EM LINHA

PESSOA_ID	PESSOA_NOME	PESSOA_IDADE
1	JOÃO DA SILVA	19
2	PEDRO ALCANTARA	26
3	FRANCISCO DE SOUZA	24
4	MARIA AUGUSTINA	36
5	JOANA FRANCISCANA	34

Fonte: do autor, 2016

Figura 3 – Armazenamento orientado a coluna

ORIENTAÇÃO EM COLUNA

PESSOA_NOME	
ID	VALOR
1	JOÃO DA SILVA
2	PEDRO ALCANTARA
3	FRANCISCO DE SOUZA
4	MARIA AUGUSTINA
5	JOANA FRANCISCANA

PESSOA_ID	
ID	VALOR
1	1
2	2
3	3
4	4
5	5

PESSOA_IDADE	
ID	VALOR
1	19
2	26
3	24
4	36
5	34

Fonte: do autor, 2016

1.4 COMPRESSÃO DE DADOS

A compressão de dados tem como objetivo a redução da quantidade de espaço necessária para o armazenamento de um conjunto de informações. Tipicamente, um algoritmo

de compressão procura explorar redundâncias na informação para aumentar a eficiência no consumo do armazenamento. As formas de compressão são diferenciadas em tempo necessário para comprimir e descomprimir os dados e no grau de compressão atingido. Quanto mais complexo o algoritmo de compressão, mais ciclos de CPU são necessários para descomprimir os dados, então deve-se levar em consideração o poder de processamento do servidor que está realizando a operação. Além disso, a utilização da memória principal para processar transações coloca uma restrição no tamanho absoluto dos dados que podem ser processados em um determinado período ou nó de processamento.

Conforme Krueger et al. (2012), a latência para a memória principal é um gargalo para o tempo de execução de cálculos, pois processadores estão desperdiçando ciclos enquanto aguardam os dados chegarem para serem processados. Enquanto os algoritmos da tecnologia de cache são uma maneira de melhorar o desempenho de forma significativa, a compressão de dados faz com que seja reduzida a quantidade de dados transferidos entre as aplicações e a memória principal, e ainda aproveita a hierarquia de cache mais eficientemente, pois os dados são melhores encaixados em cada linha de cache.

Portanto, os dados da maioria das organizações estão bem qualificados para que sejam beneficiados com a compressão, uma vez que estas técnicas exploram a redundância de dados e o conhecimento sobre o domínio de dados para melhores resultados.

Segundo Krueger et al. (2012), os principais motivos para aplicar a compressão em um IMDB são:

- A redução do tamanho total da base de dados para se que se ajuste dentro da memória principal;
- Aumento do desempenho do banco de dados, reduzindo a quantidade de dados transferidos de e para a memória principal.

A compressão de dados em memória será novamente abordada no capítulo 4 deste trabalho, através de experimentos e análises realizadas em uma estrutura de dados com uma quantidade de registros significativa para o uso de compressão.

1.5 OUTROS TÓPICOS PERTINENTES À IMDB

Além dos itens já citados, existem outros mecanismos que aumentam significativamente o desempenho no processamento de informações em bancos de dados. Esses mecanismos

podem ser observados tanto em bancos de dados em memória como em bancos de dados tradicionais com armazenamento em disco, podendo resultar um ganho de performance ainda maior quando associados a IMDBs.

1.5.1 Processamento Paralelo

O processamento paralelo pode ser alcançado de diversas maneiras na camada de aplicações de sistemas: desde a execução de processos nos servidores de aplicação até a execução de queries no sistema de bases de dados.

Segundo Plattner e Zeier (2011), o processamento de múltiplas queries pode ser realizado por aplicações multitarefa, que não irão parar quando houver a necessidade de lidar com a execução de mais de uma *query*. Para a utilização dos recursos disponíveis de hardware, as *Threads* precisam ser mapeadas fisicamente, possibilitando que diferentes tarefas possam ser executadas concorrentemente. O tempo de resposta terá o modelo ideal se for possível mapear cada *query* para um único núcleo de processamento.

O processamento de queries também envolve processamento de dados, o que significa que a base de dados também precisa ser processada em paralelo. O modelo de base de dados ideal será aquele capaz de distribuir a carga de trabalho entre os múltiplos núcleos de um sistema. Se a carga de trabalho exceder a capacidade física de um único sistema, múltiplos servidores necessitam ser adicionados para distribuição do trabalho para atingir um comportamento de processamento ideal. Na perspectiva de base de dados, o particionamento do conjunto de dados permite o processamento paralelo, visto que múltiplos núcleos situados em diversos servidores interconectados podem ser usados para o processamento dos dados.

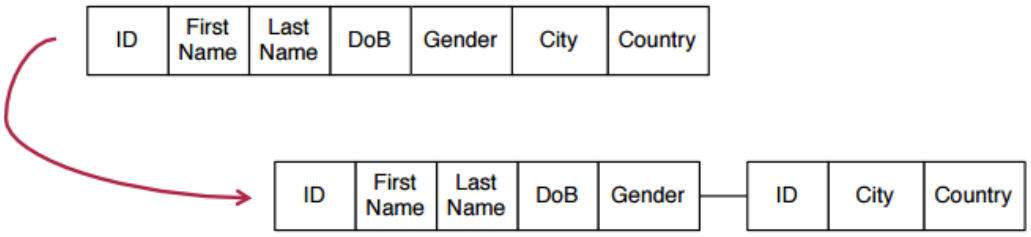
1.5.2 Particionamento

Tradicionalmente, existem dois tipos de particionamento: vertical e horizontal, sendo possível também a combinação de ambos. O particionamento vertical se refere à reorganização de colunas individuais de bases de dados. Isto se torna possível através da separação das colunas da tabela em dois ou mais conjuntos de colunas. Cada conjunto de colunas pode ser distribuído em servidores distintos.

O particionamento vertical pode ser usado para construir colunas de bases de dados com diferentes ordenações a fim de alcançar um desempenho de busca ainda superior e garantir uma

alta disponibilidade dos dados. A chave para o sucesso de um particionamento vertical é o entendimento do padrão de acesso dos dados da aplicação. Atributos que são acessados pela mesma *query* devem residir na mesma partição, visto que localizar a unir partições adicionais podem degradar o desempenho final da *query*. Na Figura 4 é possível observar o particionamento vertical, onde uma tabela é dividida em grupos de atributos, e tipicamente a chave primária da tabela é replicada.

Figura 4 – Particionamento Vertical



Fonte: HPI, 2015

Em contraste, o particionamento horizontal endereça grandes tabelas e divide elas em porções menores de dados. Como resultado, cada porção da tabela contém um subconjunto dos dados completos. A separação dos dados em partições horizontais equivalentes é usada para suportar operações de busca e para melhorar a escalabilidade. Por exemplo, uma requisição qualquer que resulte na leitura completa de uma tabela: sem nenhuma partição, um único *thread* precisa acessar todas as entradas da tabela e checar pelo predicado de seleção. Quando a mesma operação é executada um uma tabela com dez partições, a leitura completa da tabela pode ser executada simultaneamente por dez *threads*, o que reduz o tempo de resposta para aproximadamente um nono se comparado ao tempo total do mesmo procedimento em uma tabela não particionada (HPI, 2015). A Figura 5 representa um exemplo de particionamento horizontal, onde uma tabela é dividida em grupos de tuplas.

Figura 5 – Particionamento Horizontal

ID	First Name	Last Name	DoB	Gender	City	Country
1	John	Dillan	1943/05/12	m	Berlin	Germany
2	Peter	Black	1982/06/02	m	Austin	USA
3	Nina	Burg	1952/12/12	w	London	UK

ID	First Name	Last Name	DoB	Gender	City	Country
4	Lucy	Sehan	1990/01/20	w	Jerusalem	Israel
5	Ariel	Shiva	1984/07/18	w	Tokio	Japan
6	Sharon	Lokida	1982/02/24	m	Madrid	Spain

Fonte: HPI, 2015

1.5.3 Ativo e Passivo

Nas bases de dados em memória, geralmente todos os dados residem na memória principal a fim de obter alta performance no acesso as informações. Os dados podem ser classificados entre os acessados com maior frequência (*hot data*) e dados acessados de forma histórica (*cold data).* Com essa classificação, não é determinante que todos os dados residam na memória principal, uma vez que isso exigiria o aumento da quantidade de memória disponível requerida sem necessidade. Sendo assim, o *hot data* pode ser armazenado em um banco de dados ativo, ficando disponível para ser acessado na memória principal. Assim, dados históricos podem ser armazenados de forma passiva, em mídias SSDs ou discos rígidos, fornecendo desempenho suficiente para possíveis acessos a um baixo custo de implementação e arquitetura. Indiferentemente da integração entre IMDB e SGDB baseado em disco, uma transição dinâmica entre dados ativos e passivos também pode ser suportada pela base de dados (HPI, 2015).

1.5.4 Redundância de Dados

Conforme MCOBJECT LLC (2003), habitualmente, bancos de dados em disco armazenam uma grande quantidade de dados redundantes, por exemplo, dados duplicados são mantidos em estruturas de índice para permitir que o SGDB possa buscar os registros do índice para aplicá-lo sobre a consulta final. Os projetistas não se preocuparam nesse ponto com a performance do sistema, uma vez que o espaço em disco é barato e praticamente ilimitado.

Já em sistemas com armazenamento em memória, todos os dados podem ser armazenados no mais alto nível de granularidade possível, uma vez que, dado a velocidade de agregação dos dados em bases de dados em memória, todos os níveis de agregação necessários para os mais diversos tipos de aplicações e necessidades podem ser obtidos por uma única tabela detalhada em tempo real. Mesmo com a granularidade apurada das informações, queries podem ser estruturadas de forma que atendam em tempo real aos tomadores de decisão.

Ao armazenar dados dessa forma, não será necessário o armazenamento de dados redundantes para atender a necessidade de cada aplicação, proporcionando também um melhor desempenho e diminuindo drasticamente a complexidade de código da aplicação, o que por sua vez, torna a manutenção do sistema mais fácil e economiza espaço em memória que seriam disponibilizados para armazenar dados agregados e redundantes (HPI, 2015).

1.5.5 Projeção Mínima

Normalmente, os aplicativos corporativos transacionais seguem um padrão de acesso muito simples, buscando sempre os dados específicos do predicado utilizado. Uma pesquisa por certo predicado é seguida pela leitura de todas as tuplas que satisfazem a condição. É muito fácil e rápido para bancos de dados tradicionais baseados em disco ler todos os atributos da tabela, pois eles são alocados lado-a-lado fisicamente. Não importará quantos atributos serão projetados na consulta, o tempo de processamento total é muito alto devido ao custo de I/O.

Entretanto, para bases de dados colunares em memória, a situação é diferente uma vez que para cada tupla selecionada, o acesso a cada atributo projetado irá tocar uma locação diferente da memória, incluindo um pequeno atraso. Assim, para aumentar o desempenho total, apenas o conjunto mínimo de atributos que devem ser projetados para cada *query* são selecionados. Isso proporciona duas vantagens importantes: em primeiro lugar, diminui drasticamente a quantidade de dados que são transferidos entre o cliente e o servidor e em

segundo lugar, reduz o número de acessos para locações de memória randômicas e aumenta assim o desempenho global da aplicação (PLATTNER; ZEIER, 2012).

1.5.6 Escalabilidade

Conforme Isaca Journal (2013), existem duas maneiras de escalar aplicações IMDB: horizontal e verticalmente. Escalar horizontalmente permite que se criem aplicações que podem ser utilizadas simplesmente adicionando nós de computação quando precisam de maior capacidade. Em geral, as aplicações que exigem uma grande quantidade de dados de trabalho atômico ou a realização de uma grande quantidade de operações exclusivas são adequadas para escalar horizontalmente. Geralmente, aplicações em que cada transação é independente de outras transações concorrentes, pode ter escalabilidade horizontal, ou seja, cada transação pode ser encaminhada para os nós de computação separados para processamento. Escalar de forma vertical envolve o aumento da capacidade interna ou de arquitetura de um sistema, para que ele possa lidar com mais transações. Esse normalmente é o modo mais rápido para aumentar a capacidade de um sistema, sem alterar de forma considerável o ambiente de operação.

Soluções de armazenamento em memória, como o SAP HANA, suportam arquiteturas não compartilhadas, ou seja, a computação em cada nó de processamento é independente e autossuficiente, não havendo um único ponto de contenção em todo o sistema, sendo que nenhum dos nós compartilha memória ou armazenamento em disco, permitem o escalamento horizontal simplesmente com a adição de nós (ISACA JOURNAL, 2013). Segundo a Oracle (2014a), o Oracle TimesTen pode ser escalado de forma horizontal, permitindo a adição de nós de processamento, dividindo a carga de trabalhando e otimizando o processamento de dados.

1.6 IMDS X RDBMS – VANTAGENS

A principal razão que RDBMSs atuais não podem realizar as consultas necessárias com a rapidez suficiente é que os dados da consulta devem ser recuperados do disco. Os sistemas modernos fazem uso extensivo de captura dos dados mais acessados para armazenamento na memória principal, mas para consultas que processam grandes quantidades de dados, a leitura do disco ainda é necessária. Acessar e ler os dados a partir do disco pode levar uma quantidade significativa de tempo. A Tabela 1 mostra o acesso e tempos de leitura de disco e da memória principal (PLATTNER; ZEIER, 2011), onde pode ser visto que o tempo de para acesso à

memória RAM é 50 mil vezes mais rápida do que no disco, e o tempo para a leitura de 1 *MegaByte* de dados da memória, é 120X mais rápido em comparação ao disco.

Tabela 1 – Tempos de acesso e leitura de disco e memória principal

Action	Time
Main memory access	100 ns
Read 1 MB sequentially from memory	250,000 ns
Disk seek	5,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns

Fonte: HPI, 2015

De acordo com Garcia-Molina e Salem (1992), nos últimos anos a *Dynamic Random Access Memory* (DRAM) se tornou acessível o suficiente para transformar IMDBs em uma opção viável para grandes empresas, considerando o fato de que a memória principal já existe desde a década de 1980. Um IMDB com grande capacidade de armazenamento pode oferecer um tempo de resposta suficientemente bom para qualquer consulta de negócios diretamente nos dados transacionais. A capacidade de processar grandes volumes de dados na camada de banco de dados de forma rápida é fundamental para alcançar o objetivo de remover a necessidade de um sistema analítico separado do transacional.

Nos dias atuais, as empresas já podem contar com discos SSD (*solid-state drive*), que são considerados a evolução do disco rígido (HD). Um SSD pode substituir os tradicionais HD's nos servidores de armazenamento de dados, resultando em uma melhoria significativa no acesso aos dados bem como no gargalo de I/O. Mesmo com o uso desse novo tipo de armazenamento, não serão eliminados os complexos algoritmos responsáveis pelo acesso aos dados e nem a necessidade de desenvolver e melhorar queries para garantir uma melhora no desempenho, não atingindo a performance alcançada em IMDBs.

1.7 MIGRAÇÃO

Em geral, a maioria das aplicações de banco de dados podem se beneficiar da tecnologia IMDB, conforme afirma ISACA JOURNAL (2013), em grande parte porque muitas aplicações usam somente um subconjunto simples da Linguagem de Consulta Estruturada (SQL). No entanto, as soluções IMDB geralmente não têm o conjunto completo de funcionalidades disponíveis para sistemas de gerenciamento de banco de dados relacionais com base em disco

(RDBMS). Por exemplo, alguns IMDBs não suportam acionadores de banco de dados e não teriam o mesmo nível de granularidade para restrições de campo. As limitações a restrições de campo (ou seja, os caracteres unicode, formatos numéricos) são muito importantes, pois as aplicações podem ser gravadas para depender da aplicação de restrições de campo para banco de dados. Se a migração para o IMDB suaviza as restrições esperadas anteriormente, isso levanta uma série de questões relacionadas à validação do campo, como ataques do tipo *injection*.

Algumas plataformas IMDB não oferecem o mesmo nível de gerenciamento de usuário e privilégios, que é comum em bancos de dados relacionais baseados em disco. Em alguns casos, o acesso a uma instância de banco de dados permite o acesso a todos os dados contidos nessa instância e dessa forma, os administradores são obrigados a criar instâncias separadas do banco de dados para aplicações distintas, gerando assim um paradigma de gerenciamento de usuário diferente do comumente adotado (ISACA JOURNAL, 2013).

Os usuários também devem considerar os recursos exigidos para suportar os IMDBs. O recurso principal exigido é a memória. Em especial, bancos de dados muito grandes podem não se encaixar em quantidades comercialmente disponíveis de RAM. Atualmente o espaço em disco é geralmente medido em *Terabytes*, a memória, por outro lado, é medida em dezenas de *Gigabytes*.

Quando abordada a questão de mercado, deverão ser considerados também os aspectos relacionados a custos, pois empresas que oferecem recursos de armazenamento em memória podem exigir a utilização de um hardware próprio, a fim de garantir o aproveitamento total dos benefícios *in-memory*¸

1.8 SAP HANA X ORACLE TIMESTEN

Existem diversos bancos de dados que trabalham parcial ou totalmente com suas bases de dados armazenadas em memória. Alguns desses bancos de dados podem ser observados no Quadro 1. Para fins de comparação e estudo, esta seção aborda comparações de algumas características e funcionalidades oferecidas por dois bancos de dados em memória. O primeiro, objeto de estudo deste trabalho, Oracle TimesTen, e em contraste, o SAP HANA, da empresa alemã SAP. O SAP HANA é um IMDB está em ascendência no mercado atual, e por este motivo foi escolhido como objeto de comparação.

Quadro 1 – Principais IMDBs existentes no mercado

NOME	DESENVOLVEDOR
TimesTen (Oracle, 2016)	Oracle
Sap Hana (SAP, 2016)	SAP Coporation
SolidDB (Unicon Systems, 2016)	Unicon Systems
ExtremeDB	McObject

Fonte: do autor, 2016

As características do Quadro 2 foram levantadas com base em documentos e estudos feitos, principalmente, pelas próprias empresas que são concorrentes em seus serviços.

Quadro 2 – Quadro próprio de comparação TimesTen x SAP HANA

CARACTERÍSTICA	ORACLE TIMESTEN	SAP HANA
Facilidade de implantação	Simples e inclusa na última versão da aplicação (ORACLE, 2015a)	Necessita migração de tecnologia (SAP, 2016d)
Gerenciamento e acesso aos dados na memória RAM	Sim	Sim
Layout de armazenamento linear e colunar	Sim (ORACLE, 2015b)	Sim (SAP, 2016a)
Compressão de dados	*Dictionary Enconding* *Run Length Encoding* *Bit-Packing* (ORACLE, 2015b)	*Dictionary Enconding* *Run Lenght Encoding* (RLE) *Cluster Encoding* *Sparse Encoding* *Indirect Encoding* (SAP, 2016b)
Persistência dos dados em disco	Sim	Sim
Hardware proprietário	Pode ser executado em qualquer plataforma	Até 2014, somente era executada em uma

		plataforma certificada SAP x86.
Contratação de pessoal especializado	Não (ORACLE, 2014c)	Sim (ORACLE, 2014c)
Escalabilidade	Sim	Sim
Linguagem	PL/SQL e R	SQLScript e R
Atendimento das propriedades ACID	Sim	Sim
Compartilhamento de memória	Sim (ORACLE, 2016b)	Não (ISACA JOURNAL, 2013)
APIs e conectividade	ODBC JDBC PRO*C/C++ Precompiler ODP.NET (Oracle, 2016)	ODBC JDBC Python DB API (SAP, 2016e)

Fonte: do autor, 2016

Além dos recursos e características apresentados no Quadro 2, o Quadro 3, criado pela Oracle (2016a), compara o Oracle TimesTen e SAP HANA apresentando de forma mais resumida e enfática algumas questões que já foram abordadas neste capítulo.

Quadro 3 – Quadro de comparação TimesTen x SAP HANA

APLICAÇÃO IN-MEMORY	ORACLE TIMESTEN	SAP HANA
Formato Colunar	X	X
Compressão	X	X
Processamento de vetor SIMD	X	X
Agregação em memória	X	
Tamanho não limitado pela RAM	X	X
Suporte à armazenamento em linha e em coluna	X	X
Transparência de 100% na aplicação	X	

Persistência no armazenamento colunar	X	X
Visualizações materializadas com armazenamento colunar	X	
Integração com a linguagem R	X	X
Compartilhamento de Memória		X
Cumprimento das propriedades ACID	X	X

Fonte: Oracle, 2016a

Em termos gerais, os dois sistemas de gerenciamento de bancos de dados citados nesta seção, trabalham para integrar da melhor forma possível os conceitos de OLAP e OLTP, visando um impacto mínimo no que diz relação à migração, segurança e forma do armazenamento de seus dados. Algumas questões referentes a compressão de dados e análise de desempenho na consulta das informações, serão abordadas nos próximos capítulos deste trabalho.

2. ESTUDO DA FERRAMENTA – ORACLE TIMESTEN

Neste capítulo será realizado um estudo sobre o *Oracle TimesTen In-Memory Database*, um banco de dados em memória com recursos otimizados para memória e para banco de dados relacional. Com o cumprimento das propriedades ACID, o TimesTen é utilizado tanto para aplicações OLTP de alto desempenho como para aplicações OLAP e de *Business Analytics* (BA). O TimesTen pode ser usado como uma base de dados independente ou pode ser utilizado como um cache transacional de alto desempenho que perfeitamente armazena dados de um RDBMS Oracle subjacente. Esta configuração é adequada para aplicações de RDBMS da Oracle que requerem gerenciamento em tempo real de alguns de seus dados e *scale-out* em nuvens privadas ou públicas (LAHIRI; NEIMAT; FOLKMAN, 2013).

O Oracle TimesTen, é um recurso do Oracle Database 12c que promete maior agilidade e organização na gestão de bancos de dados e de informações estratégicas. É tido como uma forma de auxiliar na aceleração dos processos de desenvolvimento e integração de sistemas, de forma a revolucionar muitos conceitos envolvidos em seu gerenciamento (LAHIRI; NEIMAT; FOLKMAN, 2013).

Segundo a Oracle (2015a), o TimesTen se mostra como um verdadeiro acelerador de aplicativos, representando uma grande inovação para companhias que fazem uso de grandes bancos de dados, onde predomina a necessidade de gerenciar tudo em tempo real e da diminuição do período de tempo levado para dar retorno em queries de consulta, garantindo também a otimização da taxa de transferência para possíveis migrações e/ou backups.

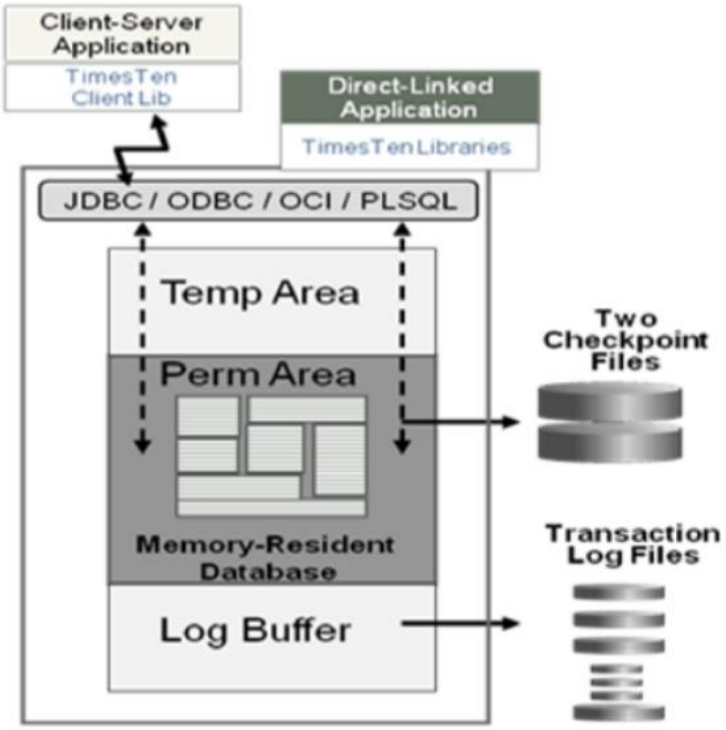
Segundo Lahiri, Neimat e Folkman (2013), com o aumento das densidades de memória RAM, sistemas com centenas de *Gigabytes*, ou até mesmo *Terabytes* de memória estão se tornando comuns. Uma grande fração de dados da empresa, ou em muitos casos, todos os dados da empresa, podem ser armazenados inteiramente na memória RAM, evitando a necessidade de operações de I/O para processamento de consultas.

O conjunto de recursos do TimesTen tem a intenção de ser aplicável a uma grande variedade de cargas de trabalho e conjuntos de dados. Conta também com suporte SQL *full-featured* e suporta APIs padrões para desenvolvimento, bem como completos mecanismos que garantem alta disponibilidade. O TimesTen é utilizado em diferentes áreas de aplicação, tais como telecomunicações, serviços financeiros, comércio eletrônico, detecção de fraude, etc., todos com rigorosas necessidades de tempo de resposta imediato (LAHIRI; NEIMAT; FOLKMAN, 2013).

2.1 ESTRUTURA BÁSICA

Apesar se ter seus dados completamente resididos na memória principal e volátil, o banco de dados TimesTen é completamente persistente. Como mostra a Figura 6, ele conta com um segmento de memória compartilhada para armazenar a base de dados. O segmento de memória é dividido em três áreas de tamanhos configuráveis: uma parte permanente, para armazenamento de objetos de banco de dados, uma parte temporária, para alocações de tempo de execução, e um buffer para o log de transações. As estruturas de disco compreendem dois arquivos de ponto de verificação (checkpoint) para capturar o estado da partição permanente na memória, e arquivos de log de transações para alterações dos registros da partição permanente do banco de dados (LAHIRI; NEIMAT; FOLKMAN, 2013).

Figura 6 – Arquitetura Oracle TimesTen



Fonte: Oracle, 2014b

2.2 PROJETO BÁSICO DO GERENCIAMENTO DO BANCO DE DADOS

Alguns princípios básicos de projeto orientaram a criação do gerenciador de armazenamento de dados de forma que mecanismos de controle de concorrência fossem feitos diretamente sobre a linha do registro acessado, sendo ainda refinados para permitir o dimensionamento máximo em uma arquitetura com múltiplos núcleos (*multi-core*).

Outro princípio mais abrangente do projeto é o uso de endereçamento baseado em memória ao invés de endereçamento lógico. Por exemplo, os índices no TimesTen contêm ponteiros para as tuplas da tabela base. Os meta-dados que descrevem o layout de uma tabela, contêm ponteiros para as páginas que constituem esta tabela. Portanto, ambas as varreduras de índice e buscas em tabelas podem operar via este ponteiro. Esta abordagem de projeto é repetida várias vezes no gerenciador de armazenamento. Como resultado, o TimesTen é significativamente mais rápido do que um banco de dados com armazenamento em disco, mesmo que este esteja completamente em cache, uma vez que não há sobrecarga ao traduzir *row-ids* lógicos para endereços de memória física dos *buffers* em um cache de *buffer* (LAHIRI; KISSLING, 2015).

Segundo Lahiri, Neimat e Folkman (2013), vários tipos de índices são suportados pelo TimesTen. **Índices *Hash*** são usados para acelerar as queries de consultas, **Índices de Bitmap** para acelerar *Joins* com predicados complexos e **Índices de Intervalo** para acelerar as verificações de intervalo. Como os dados estão sempre na memória, os índices não precisam armazenar valores-chave, economizando espaço de forma significativa. Os índices são utilizados muito mais agressivamente pelo otimizador TimesTen do que por um otimizador baseado em disco: eles são usados para varreduras, *joins*, cálculos, etc. Por exemplo, frequentemente o otimizador opta por fazer varreduras de índice ao invés de varreduras da tabela completa, pois o custo de CPU para um escaneamento por índice é geralmente menor, a menos que a coluna que está sendo escaneada esteja comprimida. O uso extensivo de índices é uma vantagem que IMDBs têm sobre bancos de dados baseados em disco, pois o uso de índices em um RDBMS resultará em operações de I/O, se o índice correspondente não estiver armazenado no *buffer cache* do RDBMS.

2.3 DURABILIDADE DA TRANSAÇÃO

Como citado anteriormente, o TimesTen foi projetado para atender completamente as propriedades ACID. Dessa forma, ele trabalha com pontos de salvamento (*checkpoint)* em log

para garantir durabilidade. A utilização de log é frequentemente um gargalo de performance em arquiteturas com multi-processadores. A fim de resolver este problema, o TimesTen tem um mecanismo de gerenciamento de log *multi-threaded*, onde a geração do log não é serializada, ou seja, pode trabalhar de forma assíncrona. Nesse mecanismo, o *buffer* de log é dividido em várias partições que podem ser populadas em paralelo por processos simultâneos. A ordenação sequencial do log é restaurada quando o log é lido a partir do disco (LAHIRI; NEIMAT; FOLKMAN, 2013).

Segundo Lahiri e Kissling (2015), para garantir mais desempenho, o TimesTen oferece opcionalmente um retardo de durabilidade, ou seja, uma aplicação se compromete a escrever um registro no *buffer* de log, sem esperar que este registro seja forçado a ser escrito no disco e assim, o log é periodicamente liberado para ser escrito no disco a cada 100 milissegundos, como uma atividade em segundo plano. Este modo de operação permite um rendimento mais elevado, com a possibilidade de uma pequena quantidade de perda de dados, a menos que o sistema esteja configurado com a replicação de *two-safe*, que será vista mais adiante neste capítulo.

Lahiri, Neimat e Folkman (2013) afirmam que o banco de dados TimesTen contempla dois arquivos de *checkpoint* para garantir a persistência dos dados e um número variável de arquivos de log. Uma vez que um ponto de verificação do banco de dados for concluído, a operação de *checkpoint* muda para o outro arquivo. Com essa abordagem, um dos dois arquivos de ponto de verificação é sempre um ponto de verificação concluído do conteúdo na memória e pode ser usado para backup e restauração dos dados. O TimesTen suporta pontos de checkpoint distintos e a frequência com que as gravações ocorrem no disco físico podem ser configuradas pelo administrador.

O TimesTen proporciona uma completa leitura de dados com isolamento transacional por padrão. Esse recurso garante a minimização dos gargalos das aplicações com um mecanismo de multi-versionamento para I/O simultâneo, utilizando bloqueio a nível de linha. Devido ao multi-versionamento, permite se que as leituras não tenham qualquer bloqueio por completo, enquanto o bloqueio em nível de linha fornece alta escalabilidade para cargas de trabalho de atualização intensiva (LAHIRI; KISSLING, 2015).

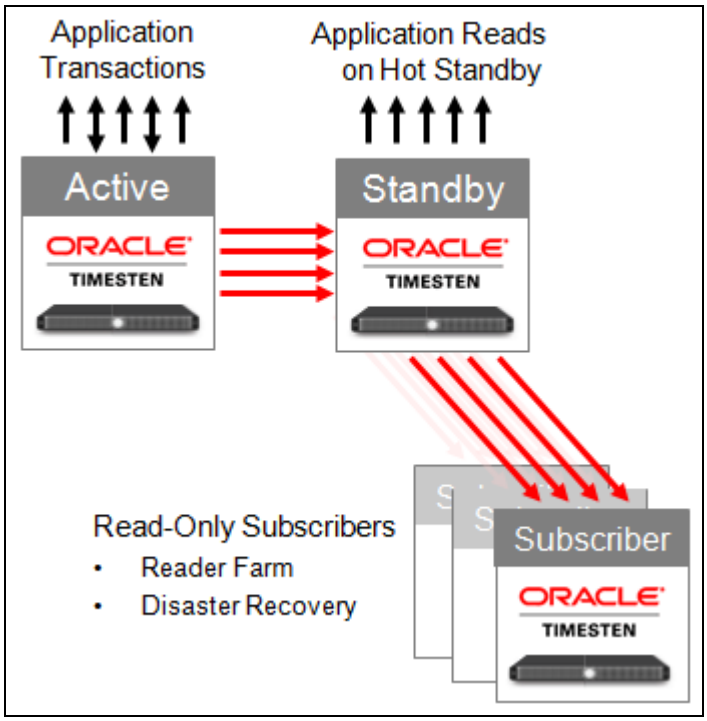
2.4 ALTA DISPONIBILIDADE

A maioria das implantações do TimesTen utiliza a replicação para garantir disponibilidade. A replicação é baseada em arquivos de log, contando com registros de log de

transporte para transações completas de um banco de dados transmissor para um banco de dados receptor, onde são aplicadas as alterações no registro. A replicação pode ser configurada em vários níveis de resiliência. Para garantir uma maior resiliência, o TimesTen fornece a replicação denominada *two-safe*, onde uma transação é confirmada localmente apenas após a confirmação ter sido reconhecida com sucesso pelo receptor. Esta combinação permite que a aplicação armazene o arquivo principal em duas memórias, garantindo a sua durabilidade sem a necessidade de qualquer escrita em disco (LAHIRI; NEIMAT; FOLKMAN, 2013).

Conforme Lahiri, Neimat e Folkman (2013), a replicação pode ser definida a nível de tabelas individuais ou ao nível de toda a base de dados, podendo ser configurada para ser *multimaster* e *bi-direcional*. A configuração preferencial de replicação é a configuração que realiza a replicação do banco de dados inteiro, tendo a origem em um banco de dados ativo e o destino, um banco de dados em espera (*standby*). A replicação do banco de dados pode ser propagada para várias bases de dados de leitura, conforme mostra a Figura 7.

Figura 7 – Arquitetura de replicação do Oracle TimesTen



Fonte: ORACLE, 2014b

Neste tipo de replicação, o banco de dados **ativo** pode hospedar arquivos que poderão ter operações tanto de escrita quanto de leitura. Já o banco de dados que está em *standby*, comporta dados onde as aplicações podem realizar somente leitura, sendo o essencial para

consultas em grandes camadas de dados. Se ocorrer uma falha do banco de dados ativo, o banco de dados que está em espera pode ser promovido para que se torne o banco de dados ativo, permitindo então operações de escrita e leitura. Se ocorrer uma falha do banco de dados que está em espera, o banco de dados ativo (também chamado de mestre) replicará os dados para as bases de dados assinantes (também chamadas de escravos) e quando o banco de dados principal voltar, um agente de replicação atualizará os dados alterados a fim de garantir a igualdade de todos os dispositivos envolvidos.

Segundo a Oracle (2015b), para a replicação bidirecional, um servidor é configurado para ser o emissor de atualizações e um servidor assinante é configurado para ser o receptor. Um servidor pode ser tanto mestre quanto assinante, garantindo assim a replicação. Ao realizar a configuração da replicação, um agente de replicação é iniciado para cada banco de dados. Se vários bancos de dados no mesmo servidor estão configurados para a replicação, então haverá um agente de replicação separado para cada banco de dados. Cada agente de replicação pode enviar atualizações para um ou mais assinantes, e receber atualizações de um ou mais servidores mestres. Cada uma destas ligações é implementada como um segmento separado de execução dentro do processo do agente de replicação, que se comunicam com outros servidores através de *stream sockets* no protocolo TCP/IP.

Para garantir mais desempenho, o agente de replicação detecta alterações em um banco de dados monitorando o log de transações e envia atualizações para os assinantes em lotes, sempre que possível. Apenas as operações confirmadas são replicadas. Na base de dados do assinante, o agente de replicação atualiza a base de dados através de uma eficiente interface de baixo nível, evitando-se a sobrecarga da camada SQL.

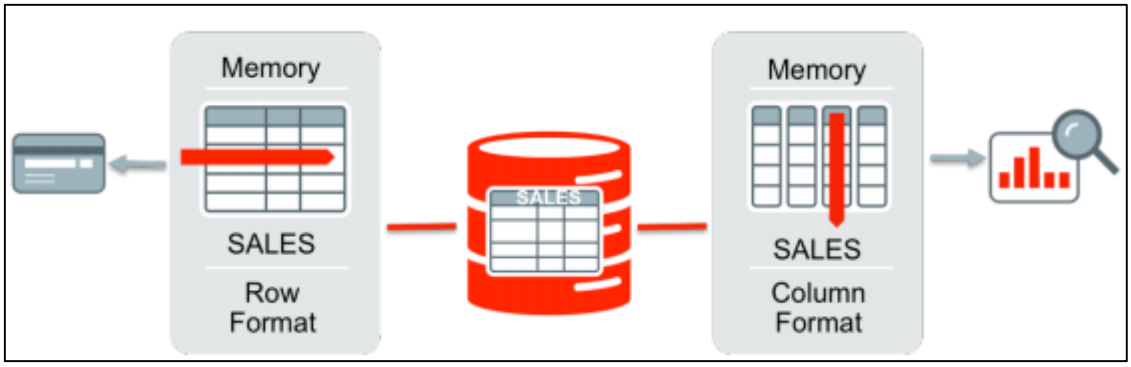
2.5 LAYOUT DE ARMAZENAMENTO

A arquitetura do Oracle TimesTen permite aproveitar ambos layouts de armazenamento, garantindo que os dados possam ser preenchidos simultaneamente tanto em formato de linha, quanto em formato de coluna, conforme demonstra a Figura 8.

A arquitetura de ambos os formatos não precisa duplicar o armazenamento necessário na memória, pois o formato de coluna em memória é dimensionado para acomodar os objetos que devem ser armazenados, uma vez que o *buffer* cache vem sendo otimizado para funcionar de forma eficaz com um tamanho muito menor do que o tamanho do banco de dados. Na prática, se espera que a arquitetura de duplo formato ocasione uma sobrecarga de 20% em termos de

requisito de memória total necessária. Segundo a ORACLE (2015b), este é um pequeno preço a se pagar para um desempenho ideal abrangendo todas as cargas de trabalho crítico.

Figura 8 – Arquitetura de armazenamento em linha e armazenamento em coluna - Oracle TimesTen



Fonte: Oracle, 2015b

Com essa abordagem aplicada pela Oracle, continuará havendo apenas uma única cópia da tabela armazenada, não havendo custos de armazenamento adicionais ou problemas de sincronização de dados. O banco de dados mantém a consistência transacional completa entre o formato linha e coluna, assim como mantém a consistência entre as tabelas e índices. O otimizador da Oracle, denominado Oracle *Optimizer*, é responsável pela otimização de consultas à base de dados. Este recurso é capaz de trabalhar de forma consciente em relação ao armazenamento em formato de coluna. Isso quer dizer que ele direciona automaticamente as rotas analíticas para o formato de coluna e operações OLTP para o formato de linha, garantindo um excelente desempenho e consistência de dados para todas as cargas de trabalho, sem quaisquer alterações nos aplicativos (ORACLE, 2015b).

2.6 COMPRESSÃO DE DADOS

Segundo a ORACLE (2015b), normalmente a compressão é considerada apenas como um mecanismo de economia de espaço. No entanto, os dados preenchidos e armazenados em uma coluna *in-memory* são comprimidos usando um conjunto de algoritmos de compressão, que não só ajuda a economizar espaço, mas também a melhorar o desempenho da consulta.

Esse método de compressão permite que consultas sejam executadas diretamente sobre colunas comprimidas, isso significa que todas as operações de verificação e filtragem serão executadas em uma quantidade muito menor de dados e dessa forma, somente serão descomprimidos quando houver a necessidade de apresentar um conjunto de resultados. A

compressão de dados pode ser feita em seis diferentes níveis. Cada um garante um nível distinto de compressão e performance, conforme demonstrado no Quadro 4.

Quadro 4 – Níveis de compressão de dados - Oracle TimesTen

NÍVEL DE COMPRESSÃO	DESCRIÇÃO
NO MEMCOMPRESS	Os dados são armazenados sem nenhuma compressão
MEMCOMPRESS FOR DML	Nível de compressão mínima é utilizada a fim de garantir performance para operações DML
MEMCOMPRESS FOR QUERY LOW	Nível de compressão otimizado para desempenho em operações de leitura
MEMCOMPRESS FOR QUERY HIGH	Nível de compressão otimizado para desempenho em operações de leitura, assim como com economia de espaço
MEMCOMPRESS FOR CAPACITY LOW	Nível de compressão balanceado com uma tendência maior para economia de espaço
MEMCOMPRESS FOR CAPACITY HIGH	Nível de compressão para economia de espaço

Fonte: Oracle, 2015b

Por padrão, os dados são comprimidos usando a opção *FOR QUERY LOW*, que oferece o melhor desempenho para consultas. Esta opção utiliza técnicas de compressão comuns, tais como:

- *Dictionary Encoding*: com esse recurso, os valores da coluna são substituídos com identificadores de uma tabela dicionário. Cada coluna tem um dicionário separado dos valores originais, e os dados comprimidos abrangem toda a coluna (ou um grupo de colunas). O método de compressão colunar tem oferecido um benefício de compressão de 5 a 10x e ajuda a acomodar grandes conjuntos de dados, como é típico de BI e cargas de trabalho de OLAP (LAHIRI; NEIMAT; FOLKMAN, 2013);
- *Run Lenght Encoding*: Com este recurso, valores repetidos são substituídos por expressões que identifiquem essa repetição, diminuindo o número de caracteres a serem armazenados. Por exemplo, uma expressão “1000000001ab” poderá ser representada por “1~80ab”.

2.7 GRIDS DE CACHE

Conforme Lahiri, Neimat e Folkman (2013), o IMDB da Oracle é capaz de garantir durabilidade, persistência e disponibilidade, podendo ser usado em qualquer aplicação de uma organização. De todo modo, caso não se deseje aplicar uma solução puramente *in-memory*, a configuração de cache do IMDB permite que o TimesTen possa ser implantado como um cache transacional persistente, para dados que estejam armazenados em um banco de dados Oracle.

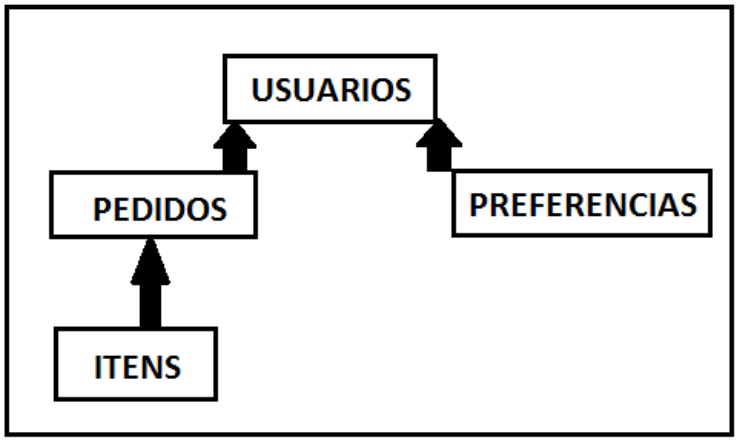
Se o TimesTen for implantado como um cache da camada de aplicação em paralelo a uma base de dados baseada em disco, proverá uma aceleração da aplicação mesmo que o conjunto de trabalho de banco de dados esteja totalmente armazenado em cache na memória dentro do cache de *buffer*. Isto acontece por duas razões:

1. *Application Proximity*: O banco de dados TimesTen pode ser colocado na camada de aplicação, resultando em menores custos de comunicação com o banco de dados para acessar os dados em cache. Para obter o melhor tempo de resposta, o TimesTen pode ser diretamente ligado à aplicação fornecendo a execução em processo de operações sobre dados em cache.
2. *In-memory otimization*: O DBA de armazenamento TimesTen é construído assumindo que todos os dados estarão resididos na memória, requerendo uma diminuição considerável de instruções para operações de banco de dados do que os bancos de dados baseados em disco.

2.7.1 Grupos de Cache

O Oracle TimesTen permite que uma coleção de tabelas seja declarada como um Grupo de Cache (*Cache Group*), trabalhando em comparação com as tabelas correspondentes de um banco de dados Oracle. Um grupo de cache é uma declaração sintática e compreende uma tabela principal (*Root table*), e uma tabela filha opcional (*Child Table*) relacionadas por restrições de chave estrangeira. Por exemplo, para um aplicativo de comércio eletrônico, uma abordagem seria a criação de um grupo de cache, onde é criada uma tabela com todos os seus usuários e perfis, e uma tabela relacionada para pedidos em aberto feitos por cada usuário, bem como a lista de preferências de cada usuário (LAHIRI; KISSLING, 2015). Este grupo de cache está representado na Figura 9.

Figura 9 – Grid de Cache do IMDB - TimesTen



Fonte: do autor, 2016

Cache Groups podem ser carregados a partir de um banco de dados Oracle de duas maneiras:

- **Pré-carregado (*Pre-loaded)***: neste modo, o grupo de cache é explicitamente carregado antes da carga de trabalho ser executada. No exemplo acima, todos os usuários poderiam ser pré-carregado a partir das RDBMS da Oracle para o grupo de cache;
- **Carregado dinamicamente (*Dynamically Loaded*)**: neste modo, os dados são trazidos para o cache quando referenciados. No exemplo acima, quando um usuário acessa o sistema, os dados são carregados de todos os registros de perfil de usuário e todos os pedidos associados a cada usuário, bem como os registros de preferência são trazidos para o grupo de cache, fazendo com que todas as referências subsequentes para este usuário se beneficiem do processamento em memória (com um processamento crítico somente na primeira referência ao objeto). Os dados a serem referenciados devem ser identificados por um predicado de igualdade na chave primária da tabela principal (neste caso, pelo identificados do perfil do usuário). A linha da tabela principal e as linhas da tabela relacionada são referidas como uma instância de cache.

Para grupos de cache carregados dinamicamente, o usuário pode definir uma de duas políticas de envelhecimento diferentes, com o objetivo de remover os dados mais antigos, garantindo que o banco de dados não ficará sem espaço. O TimesTen suporta o envelhecimento baseado em idade (com base em uma coluna do tipo *timestamp*) ou envelhecimento LRU (*Least*

Recently Used), baseado no carácter recente de utilização. Quando o mecanismo de envelhecimento remove linhas de um grupo de cache, ele os remove em unidades de instâncias de cache.

Com o objetivo de lidar com uma variedade maior de cenários de armazenamento de cache, existem dois tipos básicos de grupos de cache:

- ***Read-only cache groups***: para os dados que são atualizados com pouca frequência, mas repetidamente lidos, um grupo de cache pode ser criado no TimesTen para liberar processamento no banco de dados Oracle do cliente. Dados muito acessados, como catálogos online, portões de chegada e partida de companhias aéreas, etc. podem se beneficiar desse tipo de armazenamento em cache. As tabelas do banco de dados Oracle correspondentes as tabelas de somente leitura são atualizadas na base de dados e as atualizações de registros são feitas automaticamente com um mecanismo de atualização do TimesTen;
- ***Updatable Cahe Groups***: para dados frequentemente atualizados, um grupo de cache atualizável com sincronização na escrita é o mais apropriado. A verificação do saldo de uma conta em uma aplicação de comércio eletrônico, a localização dos assinantes de uma rede de celular, etc, são todos candidatos para esse tipo de grupo de cache. O TimesTen oferece uma série de mecanismos alternativos para a propagação de escritas para o Oracle, mas o mais utilizado e de melhor desempenho é conhecido como *Asynchronous Write-through*, onde as alterações são replicadas para o Oracle usando um mecanismo de transporte baseado em log. O mecanismo também é capaz de aplicar as alterações no Oracle paralelamente, de acordo com o projeto paralelo de todo o sistema em si.

É possível implantar vários bancos de dados de cache TimesTen em um único banco de dados Oracle. Esta arquitetura é referenciada como *In-memory Database Cache Grid*. Grupos de cache podem ser classificados em duas categorias de visibilidade dos dados sobre um grid:

- **Grupo de cache local (*Local Cache Groups*)**: o conteúdo do grupo de cache não é compartilhado e é visível apenas aos membros definidos no grid. Esse tipo de grupo de cache é útil quando os dados podem ser particionados estaticamente entre os membros do grid. Por exemplo, diferentes faixas de perfis de usuários podem ser armazenados em membros diferentes de cache no mesmo grid;

- **Grupo de cache global (*Global Cache Groups*)***:* em muitos casos, uma aplicação não pode ser estaticamente particionada e esse tipo de visibilidade permite que as aplicações compartilhem conteúdo em cache de forma transparente entre bases de dados TimesTen independentes. Com esse tipo de grupo de cache, instâncias de cache são migradas através do grid. Somente mudanças confirmadas (com *commit*) são propagadas por todo o grid. No exemplo, um perfil do usuário e os registros relacionados podem ser carregados em um membro inicial do grid no momento do início da seção desse usuário. Quando o usuário desconecta e inicia uma seção novamente, mas em um membro do grid diferente, os registros do perfil do usuário e os relacionados são migrados do primeiro membro do grid para o segundo membro do grid. Assim, o conteúdo do grupo de cache global está acessível em qualquer local, através do transporte de dados (LAHIRI; NEIMAT; FOLKMAN, 2013).

O grid de cache também oferece compartilhamento de dados via função de transporte ou consulta global. Uma consulta global é uma consulta executada em paralelo em vários membros do grid. Por exemplo, se uma consulta global precisa executar um *COUNT*(*) na tabela de perfis de usuário, a operação de contagem seria feita em todos os membros da rede, e em seguida, agruparia os resultados pela soma das contagens recebidas. Este mecanismo pode ser generalizado para consultas muito mais complexas que envolvem associações, agregações e agrupamentos. Consultas globais são úteis para a execução de operações de relações em um grid, por exemplo: Qual é o valor médio dos pedidos pendentes, para todos os usuários no grid? (LAHIRI; NEIMAT; FOLKMAN, 2013).

O próximo capítulo detalha a escolha e realização da carga de um conjunto de dados, em um banco de dados Oracle em disco e no Oracle TimesTen em memória. Essa carga de dados tornará possível um envolvimento maior com um IMDB, no qual posteriormente serão desenvolvidos experimentos relacionados aos objetivos deste trabalho.

3. INTRODUÇÃO AOS EXPERIMENTOS

Este capítulo apresenta uma introdução aos experimentos realizados neste trabalho, iniciando por uma indicação da metodologia utilizada, continuando com a configuração do ambiente para os testes e detalhando a base de dados usada nos experimentos. Desde o início, a intenção foi experimentar na prática a versão tradicional do SGBD Oracle, com funcionamento baseado em disco comparando com a versão *in-memory*, o Oracle TimesTen. Uma premissa importante, era encontrar um conjunto de dados com volume suficiente para tornar os experimentos mais interessantes.

Buscou-se a geração de uma carga de dados que possibilitasse o estudo e análise de resultados, que experimentasse e comprovasse características e recursos disponíveis na opção *in-memory*, vistos no capítulo anterior. Para a realização da carga e execução dos experimentos, foi utilizada uma base de dados aberta do fórum *StackOverflow*, que se remete a um fórum de perguntas e respostas para programadores profissionais e entusiastas da programação, onde são encontradas informações referentes a linguagens de programação e configurações de ambientes de desenvolvimento, por exemplo.

3.1 METODOLOGIA

Este trabalho iniciou com estudos acerca de bancos de dados em memória através de pesquisa bibliográfica sobre o assunto, buscando principalmente informações em livros, artigos científicos e manuais de fabricantes. Continuou com uma análise técnica de um SGBD que utiliza a abordagem *in-memory*. Tudo isto embasou o que se inicia neste capítulo, o uso de um método prático de experimentação. Esta pesquisa possui natureza aplicada e tem como foco de estudos uma comparação entre o banco de dados Oracle em disco e o banco de dados em memória Oracle TimesTen.

Após o levantamento das informações necessárias, foi construído um ambiente para a realização de testes e medições propostos nesse trabalho. Este ambiente é composto por um SGDB relacional e o outro com um SGDB em memória principal. Ambos utilizarão bases de dados idênticas, permitindo assim uma comparação fidedigna. O método utilizado para atingir o objetivo principal foi o experimental que, segundo Gil (2008), “consiste, especialmente, em submeter os objetos de estudo à influência de certas variáveis, em condições controladas e conhecidas pelo investigador, para observar os resultados que a variável produz no objeto” (apud PRODANOV; FREITAS, 2009, p. 37).

3.2 CONFIGURAÇÃO E INSTALAÇÃO DO AMBIENTE

O ambiente utilizado neste trabalho foi um servidor não virtualizado, com a seguinte configuração:

- Processador: Intel (R) Xeon (R) CPU E5-2690 v3 2,6 GHz;
- Memória RAM: 256 GB;
- Núcleos físicos: 24;
- Sistema operacional: Windows Server 2012 R2.

O levantamento de custos de implantar uma arquitetura em memória, incluindo hardware e licenciamentos, não foi realizada neste trabalho. Entretanto, para fins de estudo, o servidor citado tem um custo que varia de R$ 40.000,00 a R$ 50.000,00.

Após instalado o sistema operacional, alguns softwares foram necessários para que fossem preparados os cenários para a execução dos experimentos. Foi feito o download e instalados os seguintes softwares da Oracle, disponíveis no site https://www.oracle.com:

- Oracle Database 12c Release 1;
- Oracle Timesten DataBase 11.2.2.8;
- Oracle SQL Developer;
- Oracle Data Modeler.

A instalação do Oracle Database 12c Release 1, garantiu a criação da instância de banco denominada ORCL, configurada para armazenamento em disco, na qual foi realizada a criação das tabelas, chaves e índices utilizados nos experimentos.

A instalação do Oracle TimesTen Database demandou mais trabalho. Houve dificuldades na configuração, uma vez que para sua utilização é exigida a configuração de fonte de dados ODBC. Ao contrário da instalação do Oracle Database, o TimesTen exige a criação de um DNS (*Data Source Name*) de forma manual, e que para o presente trabalho, foi denominado “Data_Memory” e teve sua alocação máxima de memória determinada em 125 *Gigabytes*.

O Oracle SQL Developer foi utilizado para a realização dos testes e experimentos referidos no próximo capítulo deste trabalho, bem como para a criação das tabelas e índices referidos na seção anterior e definidos no modelo lógico-relacional.

O Oracle Data Modeler foi utilizado para a criação do modelo lógico-relacional do banco de dados utilizado neste trabalho e que é apresentado na seção 3.3.

3.3 BASE DE DADOS

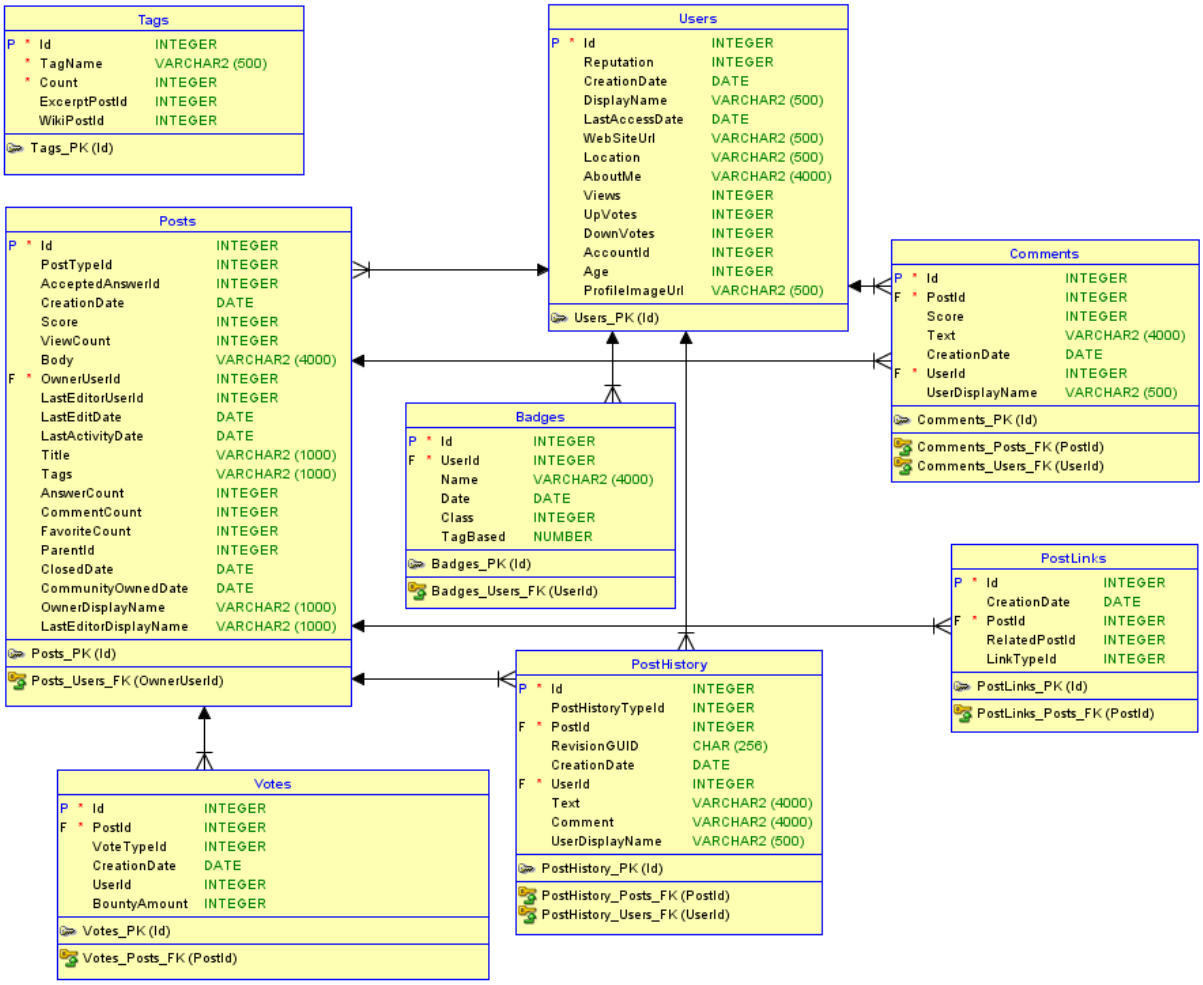
Para garantir consistência nas informações obtidas através dos experimentos, buscou-se uma base de dados com uma grande quantidade de informação, que possibilite a manipulação dos dados e que ocupe um grande espaço em disco e/ou memória. Para este fim, foi necessário realizar uma exploração online na procura de um *database* aberto e que estivesse disponível para download, permitindo assim a experimentação das funcionalidades citadas no capítulo anterior.

Após alguns dias de pesquisa, foi localizada a base de dados da aplicação citada anteriormente, no local https://ia800500.us.archive.org/22/items/stackexchange, em formato XML (*eXtensible Markup Language*). Essa base de dados está separada em 8 arquivos do tipo XML, sendo cada arquivo determinado pelo conjunto de dados correspondentes a uma tabela, totalizando aproximadamente 142 *Gigabytes* de espaço ocupado no disco rígido após o download.

Algumas colunas, como por exemplo *VoteTypeId, PostTypeId, WikiPostId, AccountId*, aparentam ter referência com outras tabelas que não foram citadas neste trabalho. É importante frisar que estas tabelas não estão disponíveis para download, não influenciando o objetivo proposto neste trabalho, nem comprometendo os testes e experimentos realizados.

O modelo visto na Figura 10, foi construído com o Oracle Data Modeler, e sua estrutura foi determinada através da observação e entendimento dos dados que compõe as tabelas, bem como do estudo realizado no site http://stackoverflow.com/.

Figura 10 – Modelo lógico-relacional



Fonte: do autor, 2016

A base de dados é composta por 8 tabelas, conforme descrição abaixo:

- Tabela ***Users***: contém o cadastro dos usuários vinculados na aplicação para utilização do fórum;
- Tabela ***Badges***: contém as medalhas/classificações dos usuários ao executarem determinadas ações no fórum da aplicação;
- Tabela ***Posts***: contém as postagens feitas pelos usuários;
- Tabela ***PostHistory***: contém o histórico das alterações dos *Posts* feitos pelos usuários;
- Tabela ***PostLinks***: contém os links que são relacionados a um *Post*;
- Tabela ***Comments***: contém os comentários feitos acerca de um *Post*;

- Tabela ***Tags***: contém as *tags* ou palavras-chave que são relacionadas a um *Post*;
- Tabela ***Votes***: contém os votos, positivos ou negativos, relacionados à um *Post*.

Para possibilitar uma análise mais apurada das informações, o Quadro 5 apresenta a relação da quantidade de linhas que compõe cada tabela da base de dados.

Quadro 5 – Volume de dados

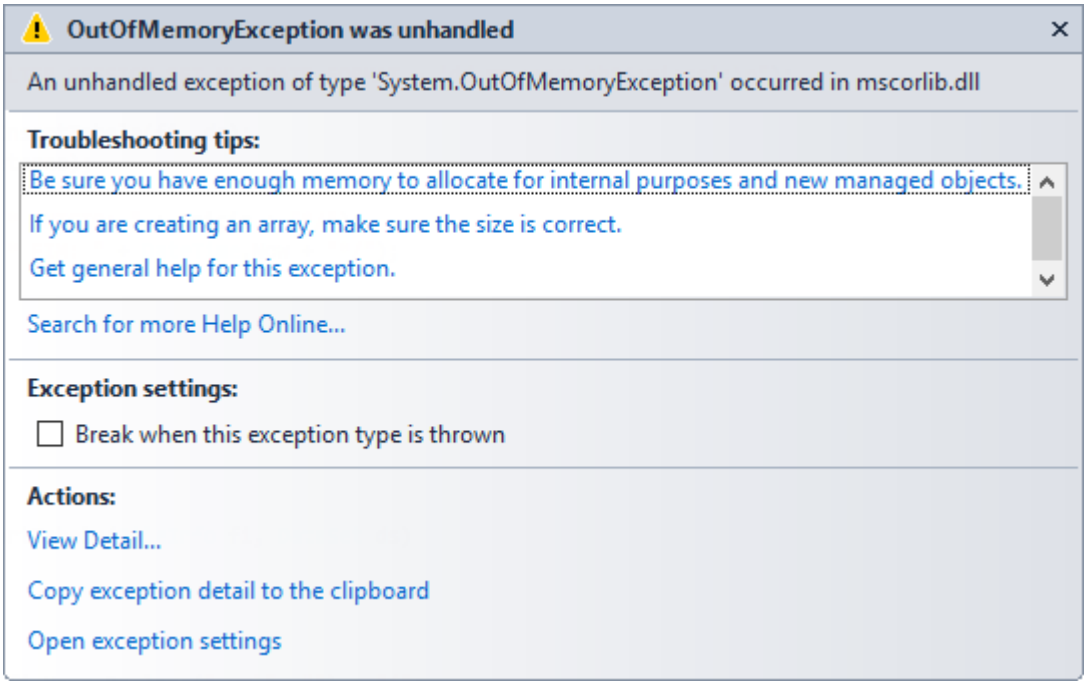
TABELA	VOLUME DE DADOS (EM LINHAS)
Users	5.848.756
Badges	17.999.999
PostHistory	17.561.139
Posts	31.739.346
PostLinks	2.254.426
Comments	48.299.299
Tags	45.425
Votes	104.499.999
Total	**228.248.389**

Fonte: do autor, 2016

3.4 CARGA DOS DADOS

Para realizar a conversão dos arquivos XMLs para a linguagem SQL, foi desenvolvida uma aplicação em C#, utilizando o Microsoft Visual Studio 2015. O processo da criação acabou tendo quatro tentativas, visto que as primeiras estratégias acabaram apresentando erros de execução devido ao elevado volume de dados, conforme pode ser observado na Figura 11. Para gerar a carga, foi necessário ler o arquivo XML linha a linha, criando a respectiva instrução de *INSERT* e gerando um novo arquivo SQL no disco, conforme a Figura 12, que posteriormente foi executado no banco de dados Oracle.

Figura 11 – Erro de estouro de memória em C#



Fonte: do Autor, 2016

Figura 12 – Instrução de *Insert* gerada a partir do XML

```
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('1','.net','233796','3624959','3607476');
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('2','html','545013','3673183','3673182');
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('3','javascript','1143897','3624960','3607052');
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('4','css','395577','3644670','3644669');
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('5','php','933625','3624936','3607050');
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('8','c','220080','3624961','3607013');
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('9','c#','962767','3624962','3607007');
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('10','c++','452918','3624963','3606997');
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('12','ruoy','161786','3624964','3607043');
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('14','lisp','4532','3656743','3656742');
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('16','python','587704','3624965','3607014');
INSERT INTO Tags(Id, TagName, Count, ExcerptPostId, WikiPostId) Values ('17','java','1086539','3624966','3607018');
```

Fonte: do Autor, 2016

Primeira tentativa: foi utilizado o objeto do tipo *DataTable* do C# que fazia a leitura do XML e carregava todos os seus dados em uma tabela em memória da aplicação. Para arquivos com até 100 *Megabytes*, não foi observado nenhum problema. Porém, arquivos com tamanho superior apresentavam problemas de estouro de memória ao criar e escrever o arquivo no formato SQL.

Segunda tentativa: foi utilizada a mesma aplicação da primeira tentativa, mas desta vez, dividindo o arquivo escrito em partes menores. Essa divisão resolveu o problema inicial

de estouro de memória ao escrever o arquivo. Porém, foi limitada à leitura de arquivos XML com até 350 *Megabytes*, apresentando novamente o problema de estouro de memória, mas desta vez, ao carregar o arquivo XML para o objeto *DataTable*.

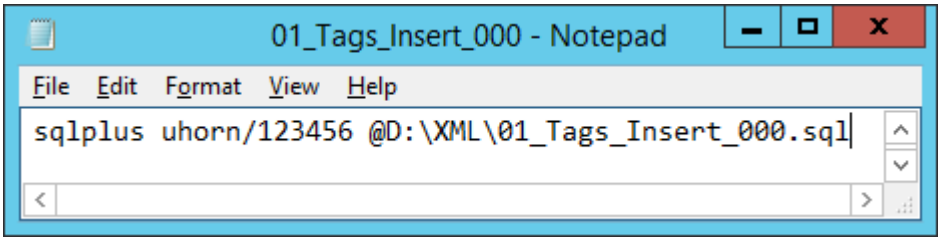
Terceira tentativa: o Visual Studio conta com bibliotecas nativas de leitura XML para o C#. A fim de utilizá-las para o trabalho, foi utilizado o objeto *XmlDocument*, que se mostrou muito promissor em questões de processamento dos dados e tempo de leitura do arquivo XML, produzindo resultados de maneira mais eficaz do que utilizando a biblioteca *DataTable*. Ao realizar os primeiros procedimentos de geração de arquivos maiores, os problemas de estouro de memória se mostraram novamente, limitando a leitura de arquivos com o tamanho até 500 *Megabytes*.

Quarta tentativa: após estudos sobre maneiras de alocar mais memória no C# ou resolver os problemas de leitura de arquivos com tamanhos medidos em *Gigabytes*, foram realizados testes com o objeto *XmlTextReader*. Esse novo objeto, também disponível para utilização no Visual Studio e aplicado ao C#, permitiu a leitura de arquivos com tamanhos extremamente grandes. Como exemplo, o XML que contém os dados da tabela de *Posts* possui cerca de 45,5 *Gigabytes*. Mesmo sendo possível realizar a leitura, houve problemas de estouro de memória na escrita dos arquivos SQL. A solução adotada foi segmentar a escrita em arquivos menores.

Resumindo, devido às limitações da alocação de memória presentes nas bibliotecas do C# utilizadas no Visual Studio e também da necessidade de dividir a geração de cada arquivo XML de origem em arquivos menores, foram gerados 1.486 arquivos com os comandos de *INSERT* de todas as tabelas relacionadas. O processo de geração desses arquivos levou aproximadamente 5 horas para ser concluído.

Com o propósito de inserir os dados na base de dados criada anteriormente, foi criada uma segunda aplicação em C#, que gera um arquivo *BATCH* com a instrução de chamada para cada arquivo SQL gerado, conforme a Figura 13. Posteriormente, a aplicação executa blocos de 30 arquivos do tipo *BATCH* em segundo plano, gerando a carga de dados necessária para a realização dos testes e experimentos.

Figura 13 – Arquivo *BATCH* com chamada para execução de arquivo SQL



Fonte: do Autor, 2016

A carga dos dados foi feita por uma aplicação própria, executando os arquivos do tipo SQL já gerados. A base de dados atingiu um tamanho de 49,668 *Gigabytes* de espaço ocupado em disco, sendo contabilizadas mais de 228 milhões de linhas. Foram necessárias aproximadamente 35 horas de execução de aplicação para realizar a inserção de todos os dados que foram propostos neste trabalho.

Após a carga dos dados na base de dados em disco, todas as tabelas foram carregadas para a base de dados em memória, a fim de permitir a realização dos experimentos de forma igualitária entre ambos tipos de armazenamento. Estes experimentos estão descritos no próximo capítulo e servirão como base para que se obtenha uma conclusão final do que foi tratado neste trabalho.

4. EXPERIMENTOS

A fim de atingir os objetivos propostos neste trabalho e com foco em realizar testes práticos comparativos de forma igualitária entre um banco de dados tradicional e um banco de dados em memória, foram realizados três experimentos utilizando a base de dados relacionada na seção 3.3. No intuito de garantir um maior esclarecimento e entendimento dos resultados, todos os experimentos foram divididos em etapas, sendo elas: descrição do experimento, resultados obtidos e discussão dos resultados. Para realização e execução destes experimentos, foi utilizada a ferramenta Oracle SQL Developer 4.2. O SQL Developer foi utilizado tanto no Oracle quanto no TimesTen, a fim de manter a isonomia entre os testes realizados.

O primeiro experimento consiste na execução e comparação de tempo de processamento para operações de consulta, inserção, alteração e exclusão de dados, determinando os ganhos e perdas de desempenho ao utilizar a mesma operação nos ambientes. O segundo experimento tem uma abordagem relacionada à compressão de dados, onde tabelas foram pré-determinadas para a realização da compressão, possibilitando a mensuração de espaço economizado com este recurso e os impactos de performance que atingem a consulta de informações em colunas com os dados comprimidos. O terceiro experimento consiste em uma análise das mudanças no plano de execução gerado por consultas de dados em disco em contraste com consultas de dados em memória.

4.1 METODOLOGIA DOS EXPERIMENTOS

Para realizar os experimentos, foram observados alguns critérios de coleta de dados, com base na observação e estudo realizado na base de dados e por meio de uma análise da utilização desta estrutura de dados no site referenciado. Para todos os experimentos deste capítulo, todas as operações foram executadas três vezes consecutivas, alternando a execução entre o ambiente com armazenamento em memória e em disco, utilizando como métrica a média de tempo das execuções. Para o experimento III, buscou-se elaborar um artefato que gerasse o processamento necessário a fim de analisar e ressaltar os resultados obtidos entre os diferentes ambientes estudados, tornando possível a observação das semelhanças e diferenças do objeto proposto no experimento.

O critério que determinou a definição de execução em três vezes consecutivas para cada ambiente, foi feito com base no histórico de execuções reproduzidas durante cada experimento, onde foi percebido que o resultado obtido com três execuções em cada simulação foi suficiente

para determinar o tempo médio necessário para cada experimento, não havendo uma variação de tempo considerável quando houve mais execuções. O tempo coletado e registrado de cada situação, foi feito através do tempo de processamento gerado pelo Oracle SQL Developer, utilizando resultados que tivessem diferenças de tempo desconsideráveis para cada experimento. Ou seja, quando observado um valor não condizente com a realidade dos demais valores, este experimento era refeito, de modo a evitar possíveis agentes externos que interferissem nos testes. Além disso, para cada execução, foi realizada uma limpeza de cache na base de dados experimentada, garantindo assim uma maior precisão dos resultados.

4.2 EXPERIMENTO I – DESEMPENHO

Esse experimento tem como objetivo, verificar e mensurar a diferença de desempenho em operações de consulta, inserção, alteração e exclusão de dados, entre um banco de dados tradicional com armazenamento em disco e um banco de dados em memória. As duas instâncias estão em execução no mesmo servidor e podem sofrer interferências externas de outros processos do servidor, de forma igualitária, não comprometendo a execução e os resultados deste experimento.

Baseando-se na bibliografia apresentada sobre banco de dados em memória, espera-se que ao final deste experimento, seja observado um ganho de desempenho no processamento dos dados que estão armazenados em memória em relação aos que estão armazenados em disco. O experimento abrangerá também queries com *joins*, forçando um processamento mais elevado por parte de ambas formas de armazenamento.

4.2.1 Descrição do Experimento

Uma vez que todas as tabelas e registros estão disponíveis para manipulação de suas informações, se tem a autonomia de utilizar qualquer tabela para a realização das queries, tanto em disco quanto em memória. Dessa forma, um dos objetivos é construir e executar uma *query* similar à utilizada no site http://www.stackoferflow.com ao consultar um *post* de um usuário qualquer, a fim de comparar a eficiência do desempenho de um caso real da utilização dos dados. Para atingir este objetivo, o experimento seguirá as seguintes etapas:

1) Elaborar o comando *SELECT* para a operação de leitura nas tabelas para os dois cenários propostos, buscando uma única linha de registro, com um predicado definido para um identificador da tabela;
2) Executar o comando;
3) Registrar os tempos de resposta;
4) Elaborar o comando *SELECT* para a operação de leitura nas tabelas para os dois cenários propostos, com um predicado definido para vários identificadores, simulando a consulta de todos os posts de um determinado usuário;
5) Executar o comando;
6) Registrar os tempos de resposta;
7) Elaborar o comando *SELECT* para a operação de leitura nas tabelas para os dois cenários propostos, utilizando um predicado associado à sintaxe “contém” na tabela de *Posts*;
8) Executar o comando;
9) Registrar os tempos de resposta;

Para validar o tempo de resposta de inserção e exclusão de dados, serão criados 10 mil registros randômicos que serão inseridos e então excluídos, possibilitando a comparação do tempo de processamento de cada tipo de operação. Para realizar este procedimento, as seguintes etapas serão seguidas:

1) Escolher a tabela para a realização das operações;
2) Gerar um script para carga aleatória de dados;
3) Executar a operação de *INSERT* em ambos ambientes;
4) Registrar os tempos de resposta;
5) Executar a operação de *DELETE* em ambos ambientes;
6) Registrar os tempos de resposta;

Para validar o tempo de resposta na manipulação de dados, serão realizadas alterações nos dados de tabelas já existentes para ambos ambientes em duas tabelas distintas, a fim de mensurar o tempo de processamento para alterações de uma grande quantidade de registros, e para isso, as seguintes etapas serão seguidas:

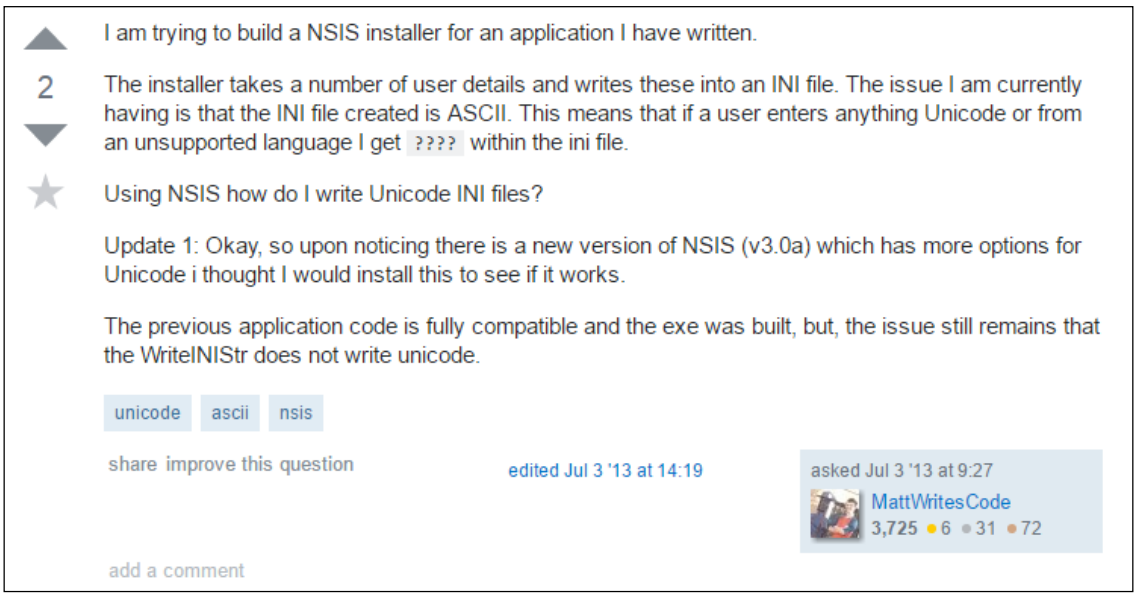
1) Escolher duas tabelas para a realização das operações;
2) Elaborar os comandos de *UPDATE* para alteração de dados;
3) Executar os comandos;

4) Registrar os tempos de resposta;

4.2.2 Resultados Obtidos

Para o processamento das operações de consulta, foi construída uma *query* que busca as informações contidas em uma consulta de um *post* no site do StackOverflow. Para exemplificar o retorno que se deseja alcançar através da *query* construída, pode-se observar a Figura 14. Foram necessários realizar *joins* entre as tabelas *Posts* e *User*, sendo que essa *query* pode ser vista na Figura 15.

Figura 14 – Exemplo real de *post*



Fonte: StackOverflow, 2016

Figura 15 – *Query* de consulta às informações de um *post* específico

```
select
  P.Id as PostId
  , p.body as Corpo
  , p.Title as Titulo
  , p.Tags as Tags
  , u.DisplayName as NomeUsuario
  , p.ViewCount as QtVisualizacoes
  , p.Score as Votos
  , p.CreationDate as DataCriacaoPost
  , p.LastEditDate as DataUltimaAtualizacao
from
  Posts P
Inner join Users u
  on u.Id = p.ownerUserId
where P.Id = 17444363;
```

Fonte: do autor, 2016

Ao executar a *query* acima, foi observado que o tempo médio de processamento computado para ambos os ambientes foi de 1 segundo, trazendo como resultado somente um registro selecionado.

Para realizar a busca de mais informações com a utilização de *joins*, foi construída uma *query* que visa consultar todos os posts realizados por um determinado usuário, novamente objetivando a busca das informações necessárias que viabilizem a utilização da informação no site referido neste trabalho. Para isso, foi utilizado a *query* da Figura 16, a qual resultou em 186 linhas selecionadas. O tempo médio de execução da mesma foi de 44,411 segundos em disco e 1,781 segundos para processamento em memória.

Figura 16 – Query de consulta às informações de todos os posts de um usuário

```
select
  P.Id as PostId
  , p.body as Corpo
  , p.Title as Titulo
  , p.Tags as Tags
  , u.DisplayName as NomeUsuario
  , p.ViewCount as QtVisualizacoes
  , p.Score as Votos
  , p.CreationDate as DataCriacaoPost
  , p.LastEditDate as DataUltimaAtualizacao
from
  Posts P
Inner join Users u
  on u.Id = p.ownerUserId
where u.id = 601245;
```

Fonte: do autor, 2016

Além das queries citadas acima, foi também construída uma terceira consulta mais genérica, com o objetivo de buscar dados na tabela de *Posts*, que contém 31.739.346 milhões de registros. Foi realizado um filtro usando *like*, no campo *body*, o qual descreve o conteúdo do *post*, conforme a Figura 17. Neste caso, a consulta em disco levou em média 45,817 segundos de processamento. Para fins de conhecimento, a utilização real dos dados do campo *body* pode ser observado na Figura 18. Utilizando os mesmos parâmetros em uma consulta nos dados armazenados em memória, os resultados tiveram uma média de tempo de processamento de 24,150 segundos. O resultado da consulta utilizada foi de 404.248 registros.

Figura 17 – *Query* de consulta às informações de posts que contém a palavra "*layout*"

```
SELECT COUNT(*)
FROM
  Posts P
INNER JOIN Users U  ON u.Id = P.ownerUserId
WHERE  body LIKE '%layout%';
```

Fonte: do autor, 2016

Figura 18 – Informações contidas da coluna "*body*" da tabela POSTS

The installer takes a number of user details and writes these into an INI file. The issue I am currently having is that the INI file created is ASCII. This means that if a user enters anything Unicode or from an unsupported language I get `????` within the ini file.

Using NSIS how do I write Unicode INI files?

Update 1: Okay, so upon noticing there is a new version of NSIS (v3.0a) which has more options for Unicode i thought I would install this to see if it works.

The previous application code is fully compatible and the exe was built, but, the issue still remains that the WriteINIStr does not write unicode.

Fonte: StackOverflow

Para realizar o teste de inserção, foi escolhida a tabela *Tags*. Foram gerados 10 mil registros com dados aleatórios, contemplando a estrutura da tabela conforme o exemplo que segue na Figura 19. A inserção dos registros em disco foi feita de forma singular, ou seja, cada linha de instrução de *INSERT* foi executada separadamente, em sequência, o que produziu um tempo de processamento com média de 161,150 segundos em média para ser finalizada. A inserção em memória foi realizada da mesma forma, e custou ao banco de dados um tempo médio de processamento médio de 158,270 segundos.

Figura 19 – Instrução de inserção de dados

```
INSERT INTO Tags (ID, TagName, Count, ExcerptPostId,WikiPostId) values (120337,'biporttitorvolutpatm',5270050,7938428,8442458);
INSERT INTO Tags (ID, TagName, Count, ExcerptPostId,WikiPostId) values (120338,'.Pr',2495142,2344903,6446237);
INSERT INTO Tags (ID, TagName, Count, ExcerptPostId,WikiPostId) values (120339,'set',2079416,9567690,442836);
INSERT INTO Tags (ID, TagName, Count, ExcerptPostId,WikiPostId) values (120340,'dictumlacus',7491264,2373789,7525643);
INSERT INTO Tags (ID, TagName, Count, ExcerptPostId,WikiPostId) values (120341,'ortami.Donecsce',6146706,2022839,5245301);
INSERT INTO Tags (ID, TagName, Count, ExcerptPostId,WikiPostId) values (120342,'er',3375775,9535045,8768277);
INSERT INTO Tags (ID, TagName, Count, ExcerptPostId,WikiPostId) values (120343,'risusetvelitv',5941138,1800013,5451678);
INSERT INTO Tags (ID, TagName, Count, ExcerptPostId,WikiPostId) values (120344,'hiculaquamn',3134076,7655392,5264223);
```

Fonte: do autor, 2016

A instrução de exclusão dos registros, tanto em disco quanto em memória, foi feita com o uso do identificador da tabela, através da instrução **“Delete tags where id >= 120326”**, atingindo os 10 mil registros inseridos anteriormente e tendo um tempo médio de processamento de 0,039 segundos em ambos ambientes.

Foram realizadas duas operações de *update* dos dados. A primeira operação foi realizada na tabela *PostsHistory*, que possui 48.299.999 milhões de registros. A clausula *where* da atualização destes registros utilizou a sintaxe “contém”, a fim de validar e forçar a busca de dados através de um predicado que não seja o identificador da tabela, conforme a Figura 20, resultando na alteração de 10.218.725 registros. Essa operação foi realizada no banco de dados em disco com um tempo médio de 575,147 segundos. Já no armazenamento em memória, a mesma operação com o mesmo predicado, obteve-se um tempo médio de processamento de 215,365 segundos.

Figura 20 – Query de alteração de dados da tabela POSTHISTORY

```
UPDATE
  PostHistory
SET Text = replace(text,'se','se123')
WHERE
  Text like '%se%';
```

Fonte: do autor, 2016

Uma segunda operação de *update* foi realizada sobre os dados da tabela *Comments*, como pode ser visto na Figura 21. A operação levou em média 585,184 segundos e 161,906 segundos para o término do processamento da operação, para os dados armazenados em disco e em memória, respectivamente. Foram alteradas 2.814.005 milhões de linhas nesta operação, de um total de 19.049.999 milhões contidas na tabela.

Figura 21 – Query de alteração de dados da tabela COMMENTS

```
UPDATE
  Comments
SET Text = replace(text,'sed','sed123')
WHERE
  Text like '%sed%';
```

Fonte: do autor, 2016

4.2.3 Discussão

O Quadro 6 apresenta um resumo de todas as operações realizadas neste experimento, fazendo referência as operações de consulta, inserção, exclusão e alteração definidas anteriormente.

Quadro 6 – Comparativo de desempenho - disco X memória

Operação/Resultado	Disco - Tempo (s)	Memória - Tempo (s)	Registros afetados	Percentual de ganho
Consulta 1	1,000	1,000	1	0,00%
Consulta 2	44,411	1,781	186	2393,60%
Consulta 3	45,817	24,150	404248	89,72%
Alteração 1	575,147	215,365	10218725	167,06%
Alteração 2	585,184	161,906	2814005	261,43%
Inserção	161,150	158,270	10000	1,82%
Excusão	0,051	0,051	10000	0,00%

Fonte: do autor, 2016

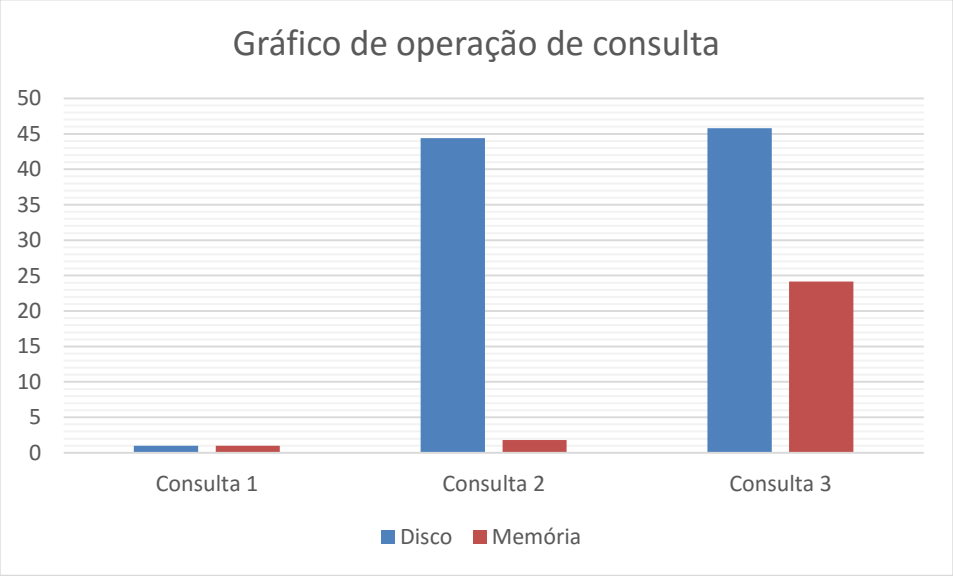
Pode ser observado que os procedimentos realizados neste experimento foram focados nas operações de consulta, inserção, exclusão e alteração, demonstrando que há um ganho considerável para operações de consulta e atualização de dados. Ao utilizar o uso do índice de chave primária na Consulta 1, pode ser observado que não há diferença entre o tempo de processamento de disco ou de memória. No entanto, ao utilizar predicados voltados diretamente sobre os dados de uma coluna da tabela que não contém índice, se observa uma constante melhora de performance. As operações de exclusão e inserção realizadas mostraram ter um comportamento e tempo de processamento muito similar nas execuções feitas em ambos ambientes.

Desta forma, pode se concluir que neste experimento, houve um ganho de performance observado quando foram executadas operações de *UPDATE* e *SELECT* em IMDB, principalmente quando envolviam filtros em colunas não indexadas. Não houve perda de processamento para operações de *INSERT* e *DELETE* de registros, mantendo o tempo de

processamento estável em ambos ambientes, demonstrando assim os benefícios que a tecnologia *in-memory* pode oferecer.

O Gráfico 1 evidencia novamente a magnitude das diferenças de desempenho para cada uma das situações em ambos ambientes de armazenamento, demonstrando que o tempo de processamento pode ser mais de 20x mais rápido no banco de dados em memória, considerando o melhor cenário deste experimento. A diferença de tempo entre as três operações realizadas, se dá em função da origem dos dados e da forma com que as consultas foram construídas. Mesmo assim, é evidenciado de uma forma bastante clara a diferença de performance entre ambos os tipos de armazenamento.

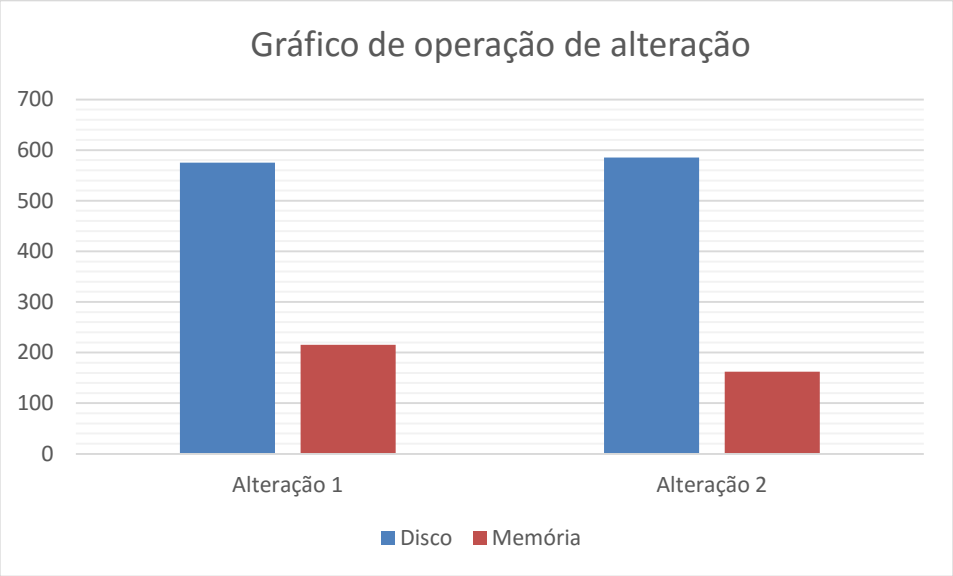
Gráfico 1 – Consulta de dados com armazenamento em disco X memória



Fonte: do autor, 2016

O Gráfico 2 demonstra a diferença em segundos na execução da operação de *UPDATE* realizado na seção anterior, entre o armazenamento em disco e o armazenamento em memória. Como pode ser observado, o processamento dos dados em memória para essas operações supera em mais de 2 vezes o mesmo processamento feito em disco.

Gráfico 2 – Alterações de dados com armazenamento em disco X memória



Fonte: do autor, 2016

Ao ser observado os resultados deste experimento, pode se contatar que o ganho médio para operações de consulta foi de 414% ao realizar a sumarização dos dados de uma tabela e de 84% ao ser realizada uma consulta específica em uma tabela utilizando o identificador como predicado e agregando mais tabelas à consulta. Já para o processamento de alteração, o ganho percebido foi superior a 167% no primeiro teste e superior a 261% no segundo teste e essa diferença de percentual pode ser justificada pela quantidade de registros afetados na operação. Esses dados comprovam as informações dispostas nos capítulos anteriores, demonstrando um ganho ao utilizar banco de dados em memória, principalmente no que diz respeito a consulta de informações. Os dados também mostram que operações de inclusão e exclusão de dados não sofreram alterações consideráveis no que diz respeito à performance das operações executadas no banco de dados, comprovando as afirmações identificadas na bibliografia da Oracle que foi utilizada neste trabalho.

4.3 EXPERIMENTO II – COMPRESSÃO DE DADOS

Este experimento visa comparar a quantidade de espaço ocupado por determinadas tabelas da base de dados em memória. Como ponto de partida, será considerado o espaço utilizado após a carga inicial dos dados. A partir daí, será realizada a compressão de dados de determinadas colunas destas tabelas. Com isto, será possível comparar o tamanho de

armazenamento entre esses objetos e também a execução de queries que validem o ganho ou prejuízo de performance ao realizar sua execução.

4.3.1 Descrição do Experimento

Conforme documentação da ORACLE (2015b), a compressão dos dados de uma tabela pode proporcionar uma economia significativa de espaço de armazenamento, principalmente quando se trata de bancos de dados em memória, uma vez que o recurso de memória pode ser limitado para algumas organizações. Em termos práticos no caso estudado, o TimesTen é capaz de realizar consultas diretamente sobre colunas comprimidas, podendo assim proporcionar uma possível melhora no desempenho de processamento nas operações de consulta.

Este experimento consiste em realizar a compressão de algumas colunas de determinadas tabelas, a fim de realizar uma comparação da economia de espaço proporcionada por tal recurso. Além disto, avaliar o impacto gerado em queries com agregações antes e após a aplicação da compressão dos dados. Para que a compressão surta efeito e seja possível uma comparação mais apurada, as tabelas utilizadas para a compressão de dados serão duplicadas. Para isso, as seguintes etapas serão seguidas:

1) Escolher duas tabelas para a realizar a compressão;
2) Definir as colunas que terão os dados comprimidos;
3) Desenvolver uma *query* que contemple as tabelas escolhidas na etapa anterior;
4) Executar a operação de *SELECT* em memória com a base não comprimida;
5) Recriar as tabelas escolhidas com compressão colunar;
6) Executar a operação de *SELECT* em memória com a base comprimida;
7) Comparar o tamanho dos dados armazenado em cada tabela e o tempo de processamento da *query* antes e após a compressão;

4.3.2 Resultados Obtidos

A fim de analisar o uso do recurso de compressão e para fins de estudo da base de dados abordada neste capítulo, a compressão foi aplicada sobre as tabelas: *Posts* e *Users*. A tabela *Posts* conta inicialmente com 24,931 *Gigabytes* de espaço ocupado e a tabela *Users* ocupa um espaço total de 1,987 *Gigabytes* como pode ser observado na Figura 22.

Figura 22 – Tamanho total ocupado pelas tabelas USERS e POSTS

```
SELECT
  Table_Name as Tabela, ROUND(SUM(TOTAL_BYTES)/1024/1024/1024,3) as Total_Posts
FROM SYS.all_tab_sizes
WHERE table_name IN ('POSTS','USERS')
GROUP BY Table_Name;
```

TABELA	TAMANHO(GB)
USERS	1.987
POSTS	24.931

Fonte: do autor, 2016

As tabelas escolhidas para a compressão de dados foram selecionadas desta forma, pois os dados contidos nas mesmas apresentavam a melhor possibilidade de se obterem resultados de forma satisfatória em relação à economia de espaço. Isso porque estas tabelas contêm colunas com informações que tem uma maior probabilidade de se repetirem, possibilitando o uso de compressão por dicionário.

Segundo recomendações constantes na documentação da Oracle (2016), cabe ao DBA avaliar e decidir quais são as colunas de cada tabela que tem potencial para compressão. Isso quer dizer que as colunas definidas para a compressão neste experimento não foram selecionadas de forma aleatória e sim justificadas por haver uma possibilidade de que informações distintas podem ter menos ocorrências nestas colunas do que nas demais que fazem parte do respectivo objeto.

Para fins de comparação, a tabela *Posts* foi recriada com o nome *Posts2*. Sobre ela, foi aplicada uma compressão do tipo dicionário, otimizado para operações de leitura, nas colunas *PostTypeId* e *Score*. Foi determinado que a coluna *PostTypeId* contasse com 8 possíveis valores distintos e a coluna *Score* com 7000 possíveis valores distintos, conforme demonstrado na Figura 23.

Figura 23 – Criação de tabela POSTS2 para compressão

```
CREATE TABLE UHORN.POSTS2 (
        "ID"                NUMBER(38) NOT NULL,
        POSTTYPEID          NUMBER(38),
        ACCEPTEDANSWERID    NUMBER(38),
        CREATIONDATE        DATE,
        SCORE               NUMBER(38),
        VIEWCOUNT           NUMBER(38),
        "BODY"              VARCHAR2(4000 BYTE),
        OWNERUSERID         NUMBER(38),
        LASTEDITORUSERID    NUMBER(38),
        LASTEDITDATE        DATE,
        LASTACTIVITYDATE    DATE,
        TITLE               VARCHAR2(1000 BYTE),
        TAGS                VARCHAR2(1000 BYTE),
        ANSWERCOUNT         NUMBER(38),
        COMMENTCOUNT        NUMBER(38),
        FAVORITECOUNT       NUMBER(38),
        PARENTID            NUMBER(38),
        CLOSEDDATE          DATE,
        COMMUNITYOWNEDDATE  DATE,
        OWNERDISPLAYNAME    VARCHAR2(1000 BYTE),
        LASTEDITORDISPLAYNAME VARCHAR2(1000 BYTE))
COMPRESS (PostTypeId BY DICTIONARY MaxValues = 8
          , Score BY DICTIONARY MaxValues = 7000) OPTIMIZED FOR READ;
ALTER TABLE UHORN.POSTS2 ADD CONSTRAINT SQLD_POSTS2_PK PRIMARY KEY ("ID");
```

Fonte: do autor, 2016

Após realizada a criação da tabela e carga dos dados, foi observada uma diminuição do tamanho de espaço de armazenamento para 23,832 *Gigabytes*, conforme demostrado na Figura 24.

Figura 24 – Tabela de POSTS2 após compressão

```
SELECT
  Table_Name as Tabela, ROUND(SUM(TOTAL_BYTES)/1024/1024/1024,3) as "Tamanho(GB)"
FROM SYS.all_tab_sizes
WHERE table_name = 'POSTS2'
GROUP BY Table_Name;
```

TABELA	TAMANHO(GB)
POSTS2	23.832

Fonte: do autor, 2016

O mesmo procedimento foi realizado para a tabela *Users*, que possui 14 colunas e um total de 5.848.756 registros. Para efetuar a compressão de dados e minimizar o espaço total ocupado, a tabela *Users* foi recriada com o nome *Users2*. Foi aplicada sobre a coluna *Location*,

uma compressão de dados por dicionário otimizada para leitura com limitação de 70.000 mil valores distintos possíveis, conforme observado na Figura 25. Após a realização da carga de dados na tabela *Users2*, observou-se uma diferença no tamanho da tabela, conforme demonstrado na Figura 26.

Figura 25 – Criação da tabela USERS2 para compressão

```
CREATE TABLE UHORN.USERS2 (
        "ID"            NUMBER(38) NOT NULL,
        REPUTATION      NUMBER(38),
        CREATIONDATE    DATE,
        DISPLAYNAME     VARCHAR2(500 BYTE),
        LASTACCESSDATE  DATE,
        WEBSITEURL      VARCHAR2(500 BYTE),
        LOCATION        VARCHAR2(500 BYTE),
        ABOUTME         VARCHAR2(4000 BYTE),
        VIEWS           NUMBER(38),
        UPVOTES         NUMBER(38),
        DOWNVOTES       NUMBER(38),
        ACCOUNTID       NUMBER(38),
        AGE             NUMBER(38),
        PROFILEIMAGEURL VARCHAR2(500 BYTE))
COMPRESS ("LOCATION" BY DICTIONARY MaxValues = 70000) OPTIMIZED FOR READ;
ALTER TABLE UHORN.USERS2 ADD CONSTRAINT SQLD_USERS2_PK PRIMARY KEY ("ID");
```

Fonte: do autor, 2016

Figura 26 – Tabela de USERS2 após a compressão

```
SELECT
  Table_Name as Tabela, ROUND(SUM(TOTAL_BYTES)/1024/1024/1024,3) as "Tamanho(GB)"
FROM SYS.all_tab_sizes
WHERE table_name = 'USERS2'
GROUP BY Table_Name;
```

TABELA	TAMANHO(GB)
USERS2	1.965

Fonte: do autor, 2016

Para fins de comparação em termos de performance, duas queries idênticas, uma relacionando tabelas não comprimidas e a outra com tabelas comprimidas, demonstradas nas figuras 27 e 28 respectivamente, foram desenvolvidas para utilizar os dados que foram referenciados nas colunas comprimidas, qualificando assim a informação do resultado para um cenário real de processamento de dados e possibilitando uma comparação entre queries de colunas comprimidas e não comprimidas.

Figura 27 – *Query* de consulta com colunas não comprimidas

```
SELECT
  count(p.postTypeId) as QTTipoPostPorLocalizacao
  ,sum(Score) as SomaPostuacaoPorLocalizacao
  ,Location
FROM POSTS P
INNER JOIN USERS U
on P.OwnerUserId = U.Id
group by Location;
```

Fonte: do autor, 2016

Figura 28 – *Query* de consulta com colunas comprimidas

```
SELECT
  count(p.postTypeId) as QTTipoPostPorLocalizacao
  ,sum(Score) as SomaPostuacaoPorLocalizacao
  ,Location
FROM POSTS2 P
INNER JOIN USERS2 U
on P.OwnerUserId = U.Id
group by Location;
```

Fonte: do autor, 2016

Para a execução da *query* em tabelas sem compressão, o tempo médio de processamento foi de 28,165 segundos, apresentando um resultado de 47.293 linhas selecionadas. Já a *query* em tabelas com colunas comprimidas apresentou um tempo de processamento médio de 29,226 segundos, retornando o mesmo número de linhas.

4.3.3 Discussão

Ao analisar os dados obtidos através deste experimento, pode se observar que a compressão de dados da tabela *Posts*, promoveu uma redução de espaço armazenado de aproximadamente 1,099 *Gigabytes* em relação à tabela original, ou seja, houve uma economia com cerca de 4,41% após a compressão. A tabela *Users* reduziu o espaço necessário para armazenamento em cerca de 22,528 *Megabytes*¸ logo, gerando uma economia de aproximadamente 1,11%.

A compressão de dados colunar por dicionário atua em tabelas que possuem um número elevado de informações repetidas em uma mesma coluna, criando um dicionário de dados no cabeçalho da tabela, a fim de que se possa realizar o apontamento da informação apenas uma vez para cada registro distinto no momento da execução da consulta ao banco de dados. No

caso do experimento com a tabela *Posts*, a tabela possui 31.739.346 registros e existem apenas 8 valores possíveis para a coluna *PostTypeId* e 1816 valores possíveis para a coluna *Score*, o que permitiu que as colunas fossem comprimidas de forma a economizar espaço no banco de dados.

Também é observado que o processamento da *query* escolhida para este experimento teve uma pequena degradação de tempo um pouco superior a 1 segundo, quando executada sobre as tabelas que tiveram colunas comprimidas, ou seja, apesar da economia de espaço apresentada pela compressão, o processamento desta *query* em específico ao consultar dados comprimidos, foi ligeiramente afetada de forma negativa. Ainda é observado que, para a consulta realizada com a junção destas tabelas comprimidas, houve uma perda de performance de 3,77% em relação à consulta nas tabelas originais.

Ao analisar tanto a informação de economia de espaço quanto a de impacto nas operações de consulta, deve se remeter novamente à afirmação da Oracle (2016) que recomenda o DBA a analisar o conjunto de dados antes da compressão, a fim de determinar se a mesma trará benefícios ou prejuízos para o banco de dados como um todo, levando em consideração a necessidade do uso de tal recurso.

4.4 EXPERIMENTO III – ANÁLISE DO PLANO DE EXECUÇÃO

Este experimento objetiva realizar uma análise e comparação do plano de execução de queries ao serem executadas em um banco de dados com armazenamento em disco e as mesmas queries sendo executadas em um ambiente com armazenamento em memória. Para determinar o plano de execução das operações realizadas, foi utilizado o Oracle SQL Developer, que conta com um recurso da visualização de plano de execução.

4.4.1 Descrição do Experimento

Segundo a Oracle (2015b), uma vez que todos os dados estão armazenados na memória, não há mais necessidade de percorrer os índices das tabelas para realizar queries de consultas. A fim de comparar e comprovar essas afirmações no banco de dados proposto neste trabalho, este experimento seguirá as seguintes etapas:

1) Elaborar o comando *SELECT* para a operação de leitura em uma tabela com o índice somente na chave primária, sendo realizado filtro em uma coluna não indexada;
2) Executar o comando em ambos ambientes;
3) Registrar plano de execução;
4) Criar índice na coluna usada como filtro no comando do item 1;
5) Executar o comando em ambos ambientes;
6) Registrar plano de execução;
7) Analisar e comparar os planos de execução.;
8) Definir uma *query* de consulta utilizando sumarizadores como COUNT e SUM, também utilizando a cláusula WHERE e por consequência da sumarização, a cláusula GROUP BY;
9) Executar o comando em ambos ambientes;
10) Registrar plano de execução;
11) Analisar e comparar os planos de execução.

4.4.2 Resultados Obtidos

Primeiramente, foi criada uma *query* simplificada para este experimento, conforme visto na Figura 29. Esta *query* realiza uma contagem de todos os registros da tabela *COMMENTS*, somando os valores da coluna *SCORE* para todos os posts onde o identificador é menor de 100053. O plano de execução em disco para esta operação é demonstrado na Figura 30.

Figura 29 – *Query* de consulta para plano de execução

```
SELECT COUNT(*), SUM(Score) FROM Comments WHERE postid < 100053;
```

Fonte: do autor, 2016

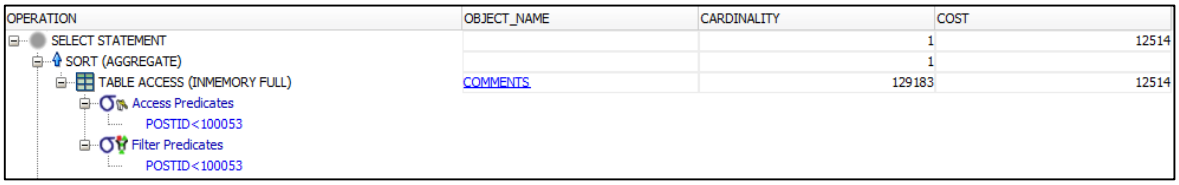
Figura 30 – Plano de execução em disco sem índice

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	329072
SORT (AGGREGATE)		1	
TABLE ACCESS (FULL)	COMMENTS	129183	329072
Filter Predicates			
POSTID<100053			

Fonte: do autor, 2016

O mesmo procedimento foi executado para o armazenamento em memória, podendo ser visto na Figura 31.

Figura 31 – Plano de execução em memória sem índice



Fonte: do autor, 2016

Após estas operações, foi elaborado um script de criação de índice para a coluna "*PostId"* conforme demonstra a Figura 32.

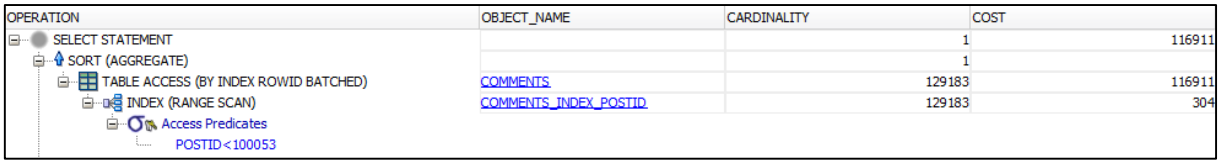
Figura 32 – *Query* de consulta para plano de execução

```
CREATE INDEX Comments_Index_PostId ON Comments(PostId);
```

Fonte: do autor, 2016

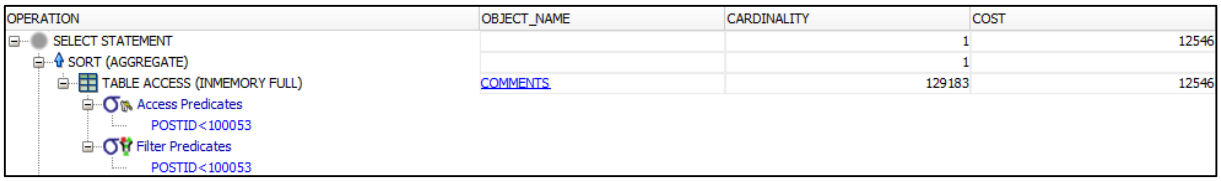
Posteriormente, novamente a *query* de consulta da Figura 29 foi executada em ambiente de disco e memória, sendo os resultados demonstrados nas figuras 33 e 34 respectivamente.

Figura 33 – Plano de execução em disco com índice



Fonte: do autor, 2016

Figura 34 – Plano de execução em memória com índice



Fonte: do autor, 2016

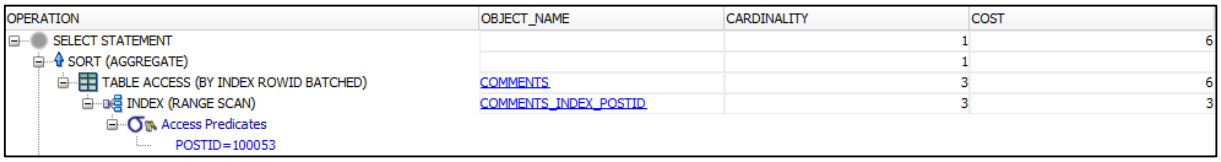
A *query* anterior foi alterada conforme a Figura 35, para utilizar a igualdade no predicado da consulta, buscando informações de um *PostId*, visando a utilização do índice criado na etapa anterior. O plano de execução gerado para esta consulta em disco pode ser visto na Figura 36.

Figura 35 – *Query* 2 de consulta utilizando busca pelo índice

```
SELECT COUNT(*), SUM(Score) FROM Comments WHERE postid = 100053;
```

Fonte: do autor, 2016

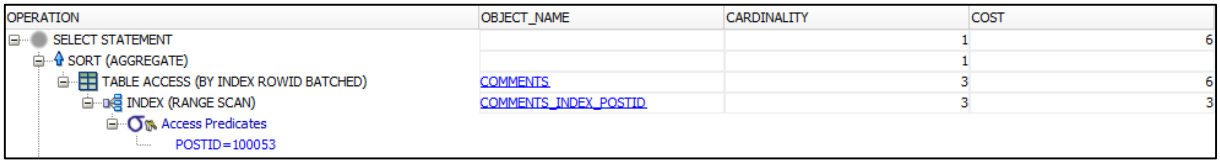
Figura 36 – Plano de execução em disco com busca pelo índice



Fonte: do autor, 2016

A figura 37 mostra o plano de execução com a operação de igualdade, realizado no armazenamento em memória.

Figura 37 – Plano de execução em memória com busca pelo índice



Fonte: do autor, 2016

Conforme a Figura 38, a *query* determinada para este experimento utiliza os sumarizadores do tipo *COUNT* e *SUM* assim como faz uso de *JOIN* entre as tabelas *Posts* e *Users* e ainda conta com o predicado do tipo *BETWEEN* e *LIKE*, e tem seus dados ordenados pela cláusula *ORDER BY*. A utilização dos filtros e ordenação para a execução da consulta faz com que o plano de execução seja executado de uma maneira que contemple diversos graus de granularidade.

Figura 38 – *Query* 2 de consulta para plano de execução

```
SELECT
  COUNT(p.postTypeId) AS QtTipoPostPorLocalizacao
  ,SUM(Score) AS SomaPostuacaoPorLocalizacao
  ,Location
FROM
  POSTS P
INNER JOIN USERS U
  ON P.OwnerUserId = U.Id
WHERE
  p.CreationDate BETWEEN TO_DATE('01/01/2008','dd/mm/yyyy')
                     AND TO_DATE('01/01/2009','dd/mm/yyyy')
  AND P.BODY LIKE '%layout%'
GROUP BY Location
ORDER BY Location;
```

Fonte: do autor, 2016

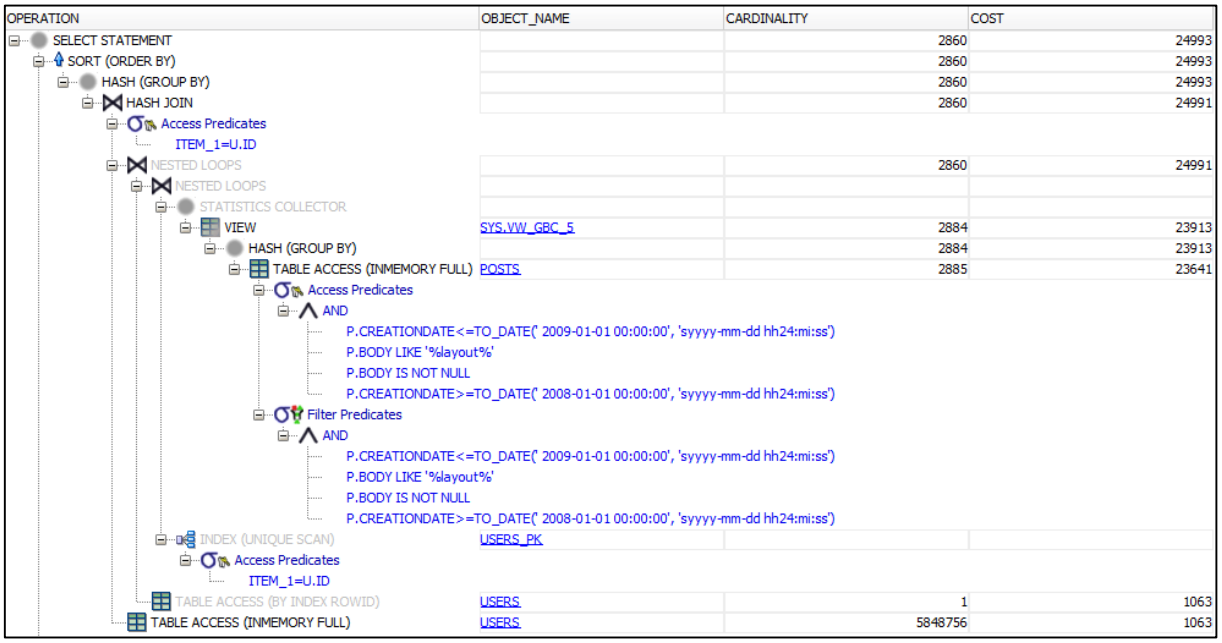
A consulta proposta foi executada em ambos tipos de armazenamento, gerando os resultados das figuras 39 para armazenamento em disco e 40 para armazenamento em memória.

Figura 39 – Plano de execução em disco

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		2860	628960
SORT (ORDER BY)		2860	628960
HASH (GROUP BY)		2860	628960
HASH JOIN		2860	628958
Access Predicates			
ITEM_1=U.ID			
NESTED LOOPS		2860	628958
NESTED LOOPS	COST=2	2884	628958
STATISTICS COLLECTOR			
VIEW	SYS.VW_GBC_5	2884	626061
HASH (GROUP BY)		2884	626061
TABLE ACCESS (FULL)	POSTS	2885	625790
Filter Predicates			
AND			
P.CREATIONDATE<=TO_DATE(' 2009-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss')			
P.BODY LIKE '%layout%'			
P.BODY IS NOT NULL			
P.CREATIONDATE>=TO_DATE(' 2008-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss')			
INDEX (UNIQUE SCAN)	USERS_PK	1	1
Access Predicates			
ITEM_1=U.ID			
TABLE ACCESS (BY INDEX ROWID)	USERS	1	2
TABLE ACCESS (FULL)	USERS	1	2

Fonte: do autor, 2016

Figura 40 – Plano de execução em memória



Fonte: do autor, 2016

4.4.3 Discussão

O primeiro teste realizado neste experimento, mostra que a consulta à tabela *COMMENTS* antes da criação do índice na coluna *PostId*, teve mudanças no plano de execução no que diz respeito ao custo gerado para a operação. No caso da execução em memória, o otimizador optou por utilizar *FullTableScan*, gerando um custo de 12546, comparado com o custo de 329072 para a execução em disco e utilizando também *FullTableScan*. Além disso, é observada a informação *INMEMORY FULL* no plano de execução para a consulta em memória, o que demonstra que toda a tabela estava em memória no momento da execução.

Após a criação do índice na coluna *PostId*, o otimizador do Oracle em disco optou por utilizar o índice ao realizar a busca de dados utilizando *RangeScan*, diminuindo o custo da operação para 116911. Já em memória, mesmo com a criação do índice, o otimizador optou por realizar a consulta através do *FullTableScan*, pois mesmo fazendo a varredura em toda a tabela para a busca dos resultados, o custo de operação manteve-se em 12546.

No próximo teste realizado, foi observada uma mudança na escolha do otimizador ao alterar o predicado da *query* originalmente utilizada. Neste segundo teste, foi alterado o operador de comparação, de “<” para “=”, fazendo com que o resultado contemplasse informações de apenas um *Post*. Essa mudança fez com que o custo no plano de execução fosse

reduzido para 6, tendo sido alterada a forma como o otimizador se comportou para o processamento em memória. Mesmo que os dados estejam em memória, o otimizador considerou que a utilização do índice para executar a consulta por *RangeScan* foi menos custosa do que utilizar a opção *FullTableScan*.

Na segunda operação de *SELECT*, apesar de conter uma maior complexidade no comando a ser executado, o otimizador optou em ambos os casos pela varredura completa da tabela *Posts*. O custo para a varredura da tabela *Posts* foi de 625790 no armazenamento em disco e de 23641 no armazenamento em memória. Já no caso da tabela *Users*¸ a opção em disco optou por realizar a busca através do índice do identificador da tabela, gerando um custo 2 para o processamento. Já em memória, o otimizador determinou que para esta consulta, era mais vantajoso utilizar a varredura completa da tabela, sendo que o custo para tal operação foi de 1063.

Em termos gerais de comparação, foi observado que o plano de execução busca resultados 25x mais rápido em memória do que disco. Ambos os testes apresentaram resultados de performance similares. Foi verificado também que, conforme consta na documentação da Oracle (2016b), o otimizador Oracle está aprimorado para definir qual a forma mais eficiente de realizar operações no banco de dados em memória, comprovando assim a eficácia dos resultados nas operações realizadas em memória quando comparadas com a execução em disco.

CONCLUSÃO

O processamento de dados em tempo real se tornou um recurso de extrema importância para as empresas que querem aproveitar o seu vasto conjunto de dados. Isso pode proporcionar a identificação de possíveis ganhos de mercado. A informação não tem validade e não tem preço, fazendo com que o valor de quem as possui, possibilite às organizações se tornarem mais competitivas.

Existem muitas empresas que trabalham com sistemas ineficientes de gerenciamento e armazenamento de dados, que são incapazes de fornecer informações atuais em tempo real. A maioria desses sistemas inaptos para esse tipo de atividade, sofrem com o gargalo de leitura de dados feito no disco rígido, por exemplo.

Uma vez que o custo de uma memória RAM vem diminuindo com o passar dos anos, a possibilidade de utilizar IMDBs em organizações de todos os tamanhos e segmentos está se tornando uma realidade para a maioria das empresas. Os problemas de desempenho de I/O encontrados em bancos de dados tradicionais estão sendo sanados em sistemas de bancos de dados em memória, sendo capazes de fornecer processamento em tempo real e possibilitando consultas de BI e *Analitycs*.

Ao fornecer informações de maneira mais rápida, os IMDBs aumentam a produtividade dos usuários finais e também dos tomadores de decisão. Além disso, a tendência é que o custo total de uma solução seja reduzido, quando comparado com outros meios de armazenamento, sendo essa redução atribuída ao fato de eliminar ou diminuir a necessidade de aplicar *tunning* no sistema.

Na pesquisa sobre um IMDB do mercado e suas principais funcionalidades, foi utilizado o Oracle TimesTen. Nele foram identificadas funcionalidades que são de extrema importância para o negócio e potencializam ainda mais o poder desse tipo de solução quando comparados aos conceitos mais abrangentes de IMDBs em geral. Funções de armazenamento orientado a linhas ou colunas, compressão de dados, replicação de dados, cache de tabelas, entre outras características como integridade transacional, persistência de dados e escalabilidade garantem o máximo desempenho das aplicações e tornam a mudança do local de armazenamento do disco para a memória possível.

O estudo realizado com a carga de dados e os experimentos realizados com o Oracle TimesTen, demonstram que o recurso de armazenamento em memória está preparado para trabalhar com grandes quantidades de dados e é capaz de proporcionar resultados de forma

potencialmente mais rápida. Foi comprovado por meio dos experimentos, que o ganho de desempenho no ambiente *in-memory* é superior ao realizar operações de consulta e alteração de registros no banco de dados em memória, e também não existe perda no processamento de operações de inserção e exclusão de registros. Também foi constatado que o otimizador da Oracle está aprimorado para optar pela melhor forma de consultar os dados em uma tabela, seja por consultas em índices ou pela varredura completa da tabela, optando sempre pela opção que representa um menor custo de processamento.

O objetivo deste trabalho: "Realizar uma análise técnica e prática de um banco de dados em memória, apresentando características e técnicas utilizadas por esse tipo de SGDB, relacionando seus benefícios e deficiências em relação ao armazenamento em banco de dados tradicional, validando sua aplicabilidade em situações reais e comparando os resultados obtidos com os de SGBD tradicionais", foi atingido através dos estudos e experimentos realizados com a ferramenta proposta, sendo possível a realização de estudos mais aprofundados em relação à outras funcionalidades do Oracle TimesTen não abordadas neste trabalho.

É possível constatar então, que já existem soluções em memória confiáveis e completas disponíveis no mercado, que oferecem todos os recursos já empregados em bancos de dados com armazenamento em disco. Estas soluções oferecem às organizações competitividade e produtividade, potencializando ainda mais o seu crescimento, graças à possibilidade a tecnologia em memória, que garante uma agilidade superior com que as informações são processadas, sendo que essas informações vinham se tornando um obstáculo para estas organizações devido as limitações performáticas citadas no escopo inicial deste trabalho.

Tendo em vista a quantidade de recursos oferecidos pela tecnologia *in-memory*, seria interessante como trabalho futuro, a exploração de experimentos abrangendo outras funcionalidades, como por exemplo, a replicação dos dados entre servidores mestres e assinantes, garantindo dessa forma a disponibilidade dos dados. Ou ainda, um estudo sobre o transporte dos dados entre ativo e passivo, possibilitando a realização de uma análise mais precisa da quantidade de memória necessária para o IMBD, evitando investimentos desnecessários em uma arquitetura em memória mais ampla. Visando também a evolução desta tecnologia e a nova disponibilidade de aplicações *in-memory*, um estudo abrangendo o mesmo foco deste trabalho, pode ser feito no Microsoft SQL Server, que atualmente disponibilizou uma versão de seu banco de dados em memória.

REFERÊNCIAS BIBLIOGRÁFICAS

ABADI, Daniel J.; MADDEN, Samuel R.; HACHEM, Nabil, 2008. **Column-Stores vs. Row-Stores: How Different Are They Really?** Acesso em: 07/09/2015. Disponível em: <http://db.lcs.mit.edu/projects/cstore/vldb.pdf>.

GARCIA-MOLINA, Hector; SALEM, Kenneth. **Main Memory Database Systems: An Overview**, IEEE Transactions on Knowledge And Data Engineering, Vol. 4, No. 6, pp. 509-516, Dec. 1992. Acesso em 10/09/2015. Disponível em: <http://pages.cs.wisc.edu/~jhuang/qual/main-memory-db-overview.pdf>.

GUPTA, Mohit Kumar; VERMA, Vishal; VERMA, Megha Singh, 2013. **In-Memory Database Systems - A Paradigm Shift**. International Journal of Engineering Trends and Technology (IJETT) – Volume 6 Number 6- Dec 2013. Acesso em: 26/08/2015. Disponível em: <http://arxiv.org/ftp/arxiv/papers/1402/1402.1258.pdf>.

HPI – Hasso Plattner Institute. **In-Memory Data Management**. 2015. Acesso em 19/09/2015. Disponível em <https://open.hpi.de/courses/imdb2015>.

IDC DIGITAL UNIVERSE STUDY, 2012. **The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East**. Acesso em 19/08/2015. Disponível em: <http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf>.

ISACA JOURNAL, 2013; VOLUME 5. **Computação In-Memory — Evolução, oportunidades e riscos**. Acesso em: 10/04/2016. Disponível em: <http://www.isaca.org/Journal/archives/2013/Volume-5/Documents/In-memory-Computing-Evolution-Opportunity-and-Risk_jrn_Portuguese_0913.pdf>.

KRUEGER, Jens; WUST, Johannes; LINKHORST, Martin; PLATTNER, Hasso; 2012; **Leveraging Compression in In-Memory Databases**. Acesso em: 12/04/2016. Disponível em: <https://www.thinkmind.org/download.php?articleid=dbkda_2012_6_20_30160>.

LAHIRI, Tirthankar; NEIMAT, Marie-Anne; FOLKMAN Steve, 2013. **Oracle TimesTen: An In-Memory Database for Enterprise Applications.** Acesso em: 31/05/2016. Disponível em: <http://sites.computer.org/debull/A13june/p6.pdf>.

LAHIRI, Tirthankar; KISSLING, Markus, 2015. **Oracle's In-Memory Database Strategy for OLTP and Analytics.** Acesso em: 07/07/2016. Disponível em: < https://www.doag.org/formes/pubfiles/7378967/docs/Konferenz/2015/vortraege/Oracle%20D

atenbanken/2015-K-DB-Tirthankar_Lahiri-Oracle_s_In-Memory_Database_Strategy_for_Analytics_and_OLTP-Manuskript.pdf>.

MCOBJECT LLC (2003). **Main Memory vs. RAM-Disk Databases**. Acesso em: 31/08/2015. Disponível em: <http://www.mcobject.com/in_memory_database>.

NARAYANAN, Dushyanth; HODSON, Orion, 2012. **Whole-System Persistence**. Acesso em 12/04/2016. Disponível em: <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/paper-updated.pdf>.

ORACLE (2014a). **Extreme Performance Using Oracle TimesTen In-Memory Database**. Acesso em: 12/04/2016. Disponível em: <http://www.oracle.com/technetwork/products/timesten/overview/wp-timesten-tech-132016.pdf>.

ORACLE (2014b) **Oracle TimesTen In-Memory Database**. Acesso em: 12/04/2016. Disponível em: <http://www.oracle.com/technetwork/products/timesten/overview/timesten-imdb-086887.html>.

ORACLE (2014c); **Oracle Database In-Memory Versus SAP HANA**. Acesso em 26/05/2016. Disponível em: <http://www.oracle.com/technetwork/database/options/dbim-vs-sap-hana-2215625.pdf>.

ORACLE (2015a). **Memory-Optimized Transactions and Analytics in One - Platform: Achieving Business Agility with Oracle Database**. Acesso em: 29/08/2015. Disponível em: <http://www.oracle.com/us/products/database/idc-business-agility-with-oracle-db-2641912.pdf>.

ORACLE (2015b); **Oracle Database In-Memory**. Acesso em: 10/05/2016. Disponível em: <http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html>.

ORACLE (2016); **Oracle TimesTen In-Memory Database and TimesTen Application-Tier Database Cache**. Acesso em: 10/05/2016. Disponível em: <https://docs.oracle.com/cd/E21901_01/timesten.1122/e21631/overview.htm#TTCIN119>.

ORACLE (2016a); **Oracle Database In-Memory - A Survey of In-Memory Databases from SAP, Microsoft, IBM, MemSQL and Oracle**. Acesso em 07/07/2016. Disponível em: <http://www.oracle.com/technetwork/database/in-memory/learnmore/dbim-competitors-2412331.pdf>.

PC MAGAZINE, **Definition of In-Memory Database**, 2013. Acesso em 02/03/2016. Disponível em <http://www.pcmag.com/encyclopedia/term/44861/in-memory-database>.

PLATTNER, Hasso; ZEIER, Alexander, 2011; **In-Memory Data Management - Technology and Application**, Second Edition, Springer. p.267.

PRICHETT, Dan.; **An Acid Alternative, Queue – Object-Relacional Mapping**, Nova York, NY, EUA, v. 6, n. 3, p. 50-55, Maio/Junho, 2008.

PRODANOV, Cleber Cristiano; FREITAS; Ernani Cesar, **METODOLOGIA DO TRABALHO CIENTÍFICO**: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico, 2ª edição. Novo Hamburgo, RS, 2013.

SAP (2016a); **SAP HANA™ Storage Requirements**. Acesso em 26/05/2016. Disponível em <http://www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/70c8e423-c8aa-3210-3fae-e043f5c1ca92?overridelayout=true&60125247188290>.

SAP (2016b); **Data Compression in the Column Store**. Acesso em 26/05/2016. Disponível em: <http://help.sap.com/saphelp_hanaplatform/helpdata/en/bd/9017c8bb571014ae79efaeb46940f3/content.htm>.

SAP (2016c); **Persistent Data Storage in the SAP HANA Database**. Acesso em 26/05/2016. Disponível em: <https://help.sap.com/saphelp_hanaplatform/helpdata/en/be/3e5310bb571014b3fbd51035bc2383/content.htm>.

SAP (2016d); **SAP HANA Tailored Data Center Integration**. Acesso em 27/05/2016. Disponível em:<http://www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/009bb21c-49b5-3210-afb5-d641f300d759?QuickLink=index&overridelayout=true&60730837565720>.

SAP (2016e); **Setting Session-Specific Client Information**. Acesso em 14/08/2016. Disponível em: < http://help.sap.com/saphelp_hanaplatform/helpdata/en/e9/0fa1f0e06e4840aa3ee2278afae16b/content.htm?frameset=/en/e9/0fa1f0e06e4840aa3ee2278afae16b/frameset.htm&current_toc=/en/34/29fc63a1de4cd6876ea211dc86ee54/plain.htm&node_id=482>.

StackOverflow, 2016; **NSIS WriteINIStr Unicode**. Acesso em 14/10/2016. Disponível em <http://stackoverflow.com/questions/17444363/nsis-writeinistr-unicode>.