UNIVERSIDADE FEEVALE

RAFAEL CARCUCHINSKI VARGAS

ANÁLISE COMPARATIVA DO SGBD SQL SERVER BASEADO EM DISCO E EM MEMÓRIA

Novo Hamburgo

2017

RAFAEL CARCUCHINSKI VARGAS

ANÁLISE COMPARATIVA DO SGBD SQL SERVER BASEADO EM DISCO E EM MEMÓRIA

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação pela Universidade Feevale

Orientador: Me. Edvar Bergmann Araujo

Novo Hamburgo

2017

AGRADECIMENTOS

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

As pessoas que convivem comigo diariamente, família e amigos principalmente, por compreender a necessidade de utilizar meu tempo livre para o desenvolvimento deste trabalho.

Ao meu professor orientador que sempre me apoiou e me orientou em qual caminho deveria seguir.

RESUMO

Com o crescimento da utilização de softwares para todas as áreas do conhecimento, a quantidade de dados gerados aumenta de forma exponencial. Com isso a necessidade de encontrar formas de armazenamento mais eficazes e de menor custo se tornou uma prioridade. O interesse na busca pelos dados gerados se tornou algo de extrema importância, principalmente por empresas que tem a necessidade de processamento em tempo real. Tem-se a necessidade de obter os dados da forma mais rápida possível e sem prejudicar os dados que estão sendo inseridos no banco de dados. A Microsoft recentemente disponibilizou uma versão com recursos em memória (otimização na memória) para o seu SGBD, o SQL Server. A partir de então, surgiram diversas dúvidas sobre o seu real ganho de desempenho no processamento de dados. A Microsoft apresenta resultados que indicam melhoras de desempenho de vinte vezes com a utilização exclusiva de tabelas *In-Memory* e de até noventa e nove com a utilização de *Compiled Stored Procedures* (procedimentos armazenados compilados), em comparação ao armazenamento em disco do próprio SQL Server. O presente trabalho tem como objetivo a comparação entre o desempenho do banco de dados Microsoft SQL Server em disco e em memória, bem como com os resultados obtidos por Horn (2016) ao utilizar o Oracle TimesTen, de forma que seja possível constatar qual a opção com melhores ganhos de desempenho e utilização de recursos. Os ganhos em processamento obtidos com a utilização do armazenamento em memória se mostraram grandiosos, principalmente aplicando a nova tecnologia de procedures compiladas, o que pode resultar em grandes benefícios para operações que envolvam a inserção e exclusão de dados.

Palavras-chave: Banco de dados em memória. *In-Memory Database*. Sistemas Gerenciadores de Banco de Dados. Desempenho. Computação em memória.

ABSTRACT

With the increase of software utilization by all knowledge areas, the amount of generated data increases exponentially. With this, the need to find more effective and lower cost storage forms has become a priority. The interest in the search for the generated data became something of extreme importance, mainly by companies that have the need for real-time processing. There is a need to obtain the data as quickly as possible and without losing any data while they are being inserted into the database. Microsoft has recently released a version with memory features (memory optimization) for your DBMS, SQL Server. Since then, several doubts have arisen about its real performance gain in the data processing. Microsoft presents results that indicate improvements in Twenty-fold performance with the exclusive use of In-Memory tables and up to ninety and nine with the use of Compiled Stored Procedures. The present work has the objective of comparing the performance of the Microsoft SQL Server database on disk and in memory, so that it is possible to verify the option with better performance gains and resource utilization. Those results obtained from the experiments were excellent, mainly for the compiled stored procedure utilization, this is what can be used to get the better results from inserted and deleted data.

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
CPU	*Central Processing Unit*
C#	*C Sharp*
DLL	*Dynamic Link Library*
DRAM	*Dynamic Random-Access Memory*
IMDB	*In-Memory Database*
IMDS	*In-Memory Database System*
I/O	*Input / Output*
OLAP	*Online Analytical Processing*
OLTP	*Online Transaction Processing*
RAM	*Random-Access Memory*
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	*Structured Query Language*
XML	*EXtensible Markup Language*
WAN	*Wide Area Network*

SUMÁRIO

INTRODUÇÃO

Com o crescimento exponencial da utilização de dispositivos eletrônicos no mundo, estão sendo gerados imensos volumes de dados por diversas interações de usuários com seus sistemas. A geração e, principalmente, o consumo dos dados gerados por todos os dispositivos/sistemas, se tornaram fontes de extrema importância para a tomada de decisões das equipes de planejamento estratégico. Com os avanços do mercado, a necessidade da obtenção dos dados em tempo real se tornou ainda mais imprescindível para que se saia na frente dos concorrentes.

Os bancos de dados relacionais tradicionais foram e continuam sendo uma ótima opção para as empresas. Contudo, com o constante crescimento de dados armazenados e analisados, estes SGBDs tradicionais têm demonstrado algumas limitações, como escalabilidade, armazenamento, lentidão no processamento e gerenciamento de grandes volumes de dados, o que se tornou algo desafiador. (ABRAMOVA et al., 2014)

Os Sistemas de Gerenciamento de Banco de Dados tradicionais, que tem seus dados armazenados em disco, ainda são muito utilizados devido ao custo do armazenamento em memória ainda ser superior. Contudo, essa pressuposição em relação ao custo da memória tem-se modificado. Nos últimos anos os valores para se manter dados na memória caíram em um fator de 10 a cada cinco anos (DIACONU et al., 2013). Para dar suporte a imensa quantidade de dados gerados, o conceito de banco de dados em memória vem sendo mais difundido, algo que veio para mudar todo o paradigma dos sistemas de gerenciamento de banco de dados.

Conforme Garcia-Molina e Salem (1992), em um banco de dados em memória (IMDB – *In-memory database*) os dados ficam permanentemente na memória principal, enquanto em um banco de dados convencional eles permanecem no disco rígido. Percebe-se que o conceito não é recente, mas somente nos últimos anos esta tecnologia tem recebido mais atenção e investimentos por parte dos grandes fornecedores de SGBDs do mercado. 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.

Seguindo a analogia de Plattner e Zeier (2012), pode-se imaginar que toda vez que se queira um copo d'água, ao invés de buscar na cozinha, se tenha que dirigir até o aeroporto, pegar um voo até a Alemanha e pegar o copo d'água lá. Na perspectiva de uma CPU moderna, acessar dados que estão na memória é como pegar água da cozinha. Enquanto, acessar dados do disco rígido é como voar até a Alemanha para buscar água.

Utilizando-se da tecnologia de banco de dados em memória, as operações de leitura e escrita de dados sofrem grandes melhoras, uma vez que são eliminadas e/ou minimizadas as operações de *input*/*output*, mais conhecidas como I/O. Obtém-se assim o processamento de dados em tempo real e, portanto, o gerenciamento e processamento de grandes quantidades de dados em um curto espaço de tempo. Além disso, considerando 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.

A sobrecarga para gerenciar as transações simultâneas será menor, isso porque o acesso aos dados é muito mais rápido e, portanto, os bloqueios são liberados muito 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 em uma rede WAN de alta velocidade para que se permita virtualmente um local de dados e acesso a esses dados, portanto, mais rápido, mesmo em caso de nós distribuídos (GUPTA; VERMA; VERMA, 2013).

Quando se trata do processamento de grandes quantidades de dados em tempo real, banco de dados em memória parecem ser a resposta ideal para o problema em questão. A performance obtida pelo IMDB é o objetivo principal de quem adere a esta tecnologia. Grandes fornecedores de Sistemas e SGBDs tem investido muito em soluções utilizando-se dos benefícios do armazenamento em memória, dentre as quais, SAP, Oracle e a Microsoft.

A Oracle introduziu seu banco de dados em memória em meados de 2014, com a premissa da entrega de dados em tempo real para a tomada de decisão quando for necessário. Horn (2016) realizou um trabalho prático cujo objetivo foi a comparação da solução de armazenamento em memória da Oracle, o produto Timesten, comparando-o com o armazenamento em disco. O trabalho demonstrou resultados satisfatórios da solução em memória do Oracle.

Enquanto isso, a Microsoft trabalhava para adicionar esta opção de armazenamento ao seu produto, SQL Server, desde a versão 2014, mas só foi disponibilizado na versão 2016. Segundo a Microsoft (2016a), com a utilização da tecnologia *In-memory* em conjunto com o MS SQL Server, podem ser obtidas grandes melhoras na taxa de transferência e latência. A ferramenta foi desenvolvida para atender as necessidades das transações mais custosas, e a Microsoft tem trabalhado com outras empresas para comprovar estes ganhos. Os dois principais

recursos do *In-Memory OLTP* (*Online Transaction Processing* ou Processamento de Transações em Tempo Real) são:

- Tabelas otimizadas em memória (*Memory-optimized table*): possuem dois tipos, as duráveis, que permanecem mesmo com quedas do servidor, e as não duráveis, que se perdem com reinicializações, mais utilizadas para processamentos em que os dados não são necessários após a consulta.
- Procedimento armazenado nativo compilado (*Natively compiled stored procedure*s): SQL Server pode nativamente compilar procedimentos armazenados que acessam tabelas otimizadas em memória. Tal compilado faz com que o acesso aos dados seja mais rápido e mais eficiente do que tradicionais consultas.

Todas as tabelas armazenadas em memória precisam de pelo menos um índice. Cada *primary key* implica na criação de um índice. Assim sendo, se uma tabela possui uma *primary key*, ela possui ao menos um índice. Os índices devem ser criados juntamente com a tabela. A criação de um índice no decorrer da utilização da tabela pode causar o não aproveitamento de todo o seu desempenho em potencial. Índices criados em memória não são replicados para o disco, a estrutura de indexação é criada juntamente com a tabela, com a alteração ou com a inicialização do banco de dados (MICROSOFT, 2015).

Este trabalho tem como objetivo apresentar técnicas e propor soluções para problemas encontrados na utilização de banco de dados em memória, utilizando-se da ferramenta da Microsoft SQL Server 2016 para demonstrar comparativos de melhorias no armazenamento e processamento de grandes volumes de dados entre SGDB’s que tem seu armazenamento em disco e em memória.

Este trabalho está dividido em cinco capítulos, onde o Capítulo 1 apresenta as definições de banco de dados em memória com algumas de suas características. O Capítulo 2 abrange a ferramenta alvo do estudo, com algumas particularidades das funções agregadas na versão 2016, principalmente a questão de armazenamento em memória.

Nos capítulos seguintes serão realizadas as validações técnicas na aplicação, onde no Capítulo 3 constará a forma com que os dados foram obtidos e tratados para a inserção nas bases de dados do estudo.

O Capítulo 4 conterá todos os experimentos realizados exclusivamente no SQL Server 2016, com execuções de diversas operações nas bases com armazenamento em disco e em

memória, com a apresentação do passo a passo das operações e os resultados obtidos em forma de gráficos.

O quinto capítulo será de comparações com o trabalho realizado anteriormente por Horn (2016), que teve objetivos semelhantes com o estudo da ferramenta TimesTen, que tem como fornecedora a Oracle, concorrente da Microsoft na área de banco de dados.

A adição da opção em memória na ferramenta da Microsoft é recente, o que ocasiona na existência de poucos estudos científicos que realizam comparações independentes da empresa fornecedora do produto.

1. BANCO DE DADOS EM MEMÓRIA

Com o constante surgimento de novas tecnologias, mais opções de dispositivos e atualizações para os já existentes, a geração de dados tem tido grande crescimento pela demanda de usuários comuns em diversos sistemas. Uma destas tecnologias, é a utilização de banco de dados em memória principal, que tem sido o carro chefe de grandes empresas para a realização de busca e tratamento de dados para as equipes de *Business Inteligence* (BI). Com o passar dos anos, diversas empresas estão adotando a utilização de banco de dados em memória pela velocidade que a solução entrega, agradando os usuários que buscam principalmente isso.

Há muito tempo, quando os maiores SGBDs foram criados, o hardware tinha seu valor muito elevado. Assim, os servidores costumavam ter apenas uma ou poucas CPUs e uma pequena quantia de memória disponível. Os servidores de banco de dados tinham de lidar com dados armazenados em disco e carregá-los na memória somente quando tais dados fossem solicitados. Esta situação tem mudado drasticamente com o passar dos anos. Nos últimos 30 anos o preço da memória tem caído em um fator de dez a cada 5 anos, hoje é possível obter servidores com 32 *cores* e 1TB de RAM por menos de cinquenta mil dólares. Contudo, se sabe também que a quantidade de dados gerados cresce rapidamente. (KOROTKEVITCH, 2016)

Conforme Najam (2016), um grande número de empresas de tecnologia da informação tem oferecido soluções de banco de dados em memória para diversos ramos da indústria. Com essa adesão, também estão em ascendência o desenvolvimento de ferramentas para controle de cache, com o objetivo de fazer com que companhias mantenham o conteúdo de seus bancos de dados relacionais na memória principal, como Memcache e Redis. Como exemplo, o Facebook utiliza o MySql para armazenamento, mas utiliza Memcache para realizar consultas de forma mais veloz. Microsoft, Oracle, SAP e Altibase são consideradas as maiores empresas provedoras do serviço *in-memory*, mantendo contínuas melhoras e desenvolvimento de novas soluções.

A utilização dos bancos de dados em memória vem se tornando rapidamente o favorito entre as grandes empresas, onde a principal questão vem sendo modificada de "O que é isso?" para "Como eu posso fazer isso?". Está muito claro que empreendedores estão de forma constante procurando maneiras de melhorar a eficiência de seus serviços. A capacidade de um banco de dados em memória não é somente melhorar o tempo de resposta, mas também de possibilitar novas linhas de negócio, podendo até mesmo alterar valores de produtos para ganhar a competição com outras empresas. Tais alterações de valores podem ser feitas com a utilização

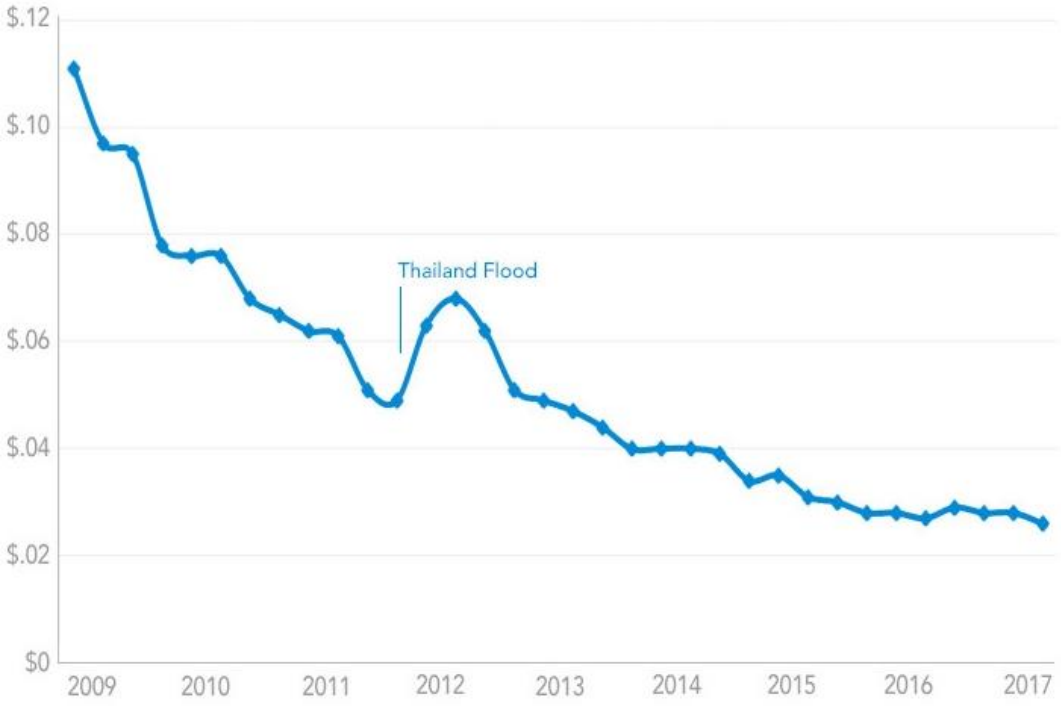
de banco de dados tradicionais, mas quando é realizado em memória, o processo se torna muito mais rápido e o atraso no tempo resposta é eliminado.

1.1 CONCEITOS

Conceitualmente, banco de dados em memória (IMDB) é um banco de dados onde os dados são armazenados na memória principal do servidor ao qual está vinculado, o que permite ganhos de performance ao ser comparado ao armazenamento tradicional realizado em disco rígido. Com os dados armazenados na memória principal, é possível que se obtenha uma resposta mais rápida para as diversas operações envolvendo o SGBD.

Conforme Kemper (2013), banco de dados em memória estão sendo utilizados desde os anos 1980. No entanto, apenas nos últimos anos o custo pelo uso de memória vem se tornando viável o suficiente para se ter o armazenamento completo de aplicações, conforme Figura 1. Isto despertou o interesse de grandes empresas da área da Tecnologia da Informação, tais como Microsoft, Oracle e SAP. O crescimento constante da capacidade de armazenamento em memória principal, possibilita maior capacidade computacional fazendo com que seja possível o processamento de grandes quantidades de dados.

Figura 1 – Custo por GB nos últimos anos



Fonte: Klein, 2017

Banco de dados em memória fornecem significativas melhoras de desempenho de duas formas. A primeira e mais importante, mantendo dados na memória principal ao invés de armazená-los no disco rígido, fazendo com que seja minimizada ou até mesmo eliminada a latência das *querys.* A segunda, arquiteturas alternativas de banco de dados fazem com que o acesso aos dados seja feito de forma mais eficiente aproveitando-se melhor do uso da memória disponível. Como exemplo, grande parte da utilização de IMDB utilizam-se do layout de armazenamento de tabelas em colunas, ao invés do tradicional armazenamento em linhas. Valores salvos em colunas têm maior facilidade para ser comprimidos, e o SGBD pode percorrer os dados assim armazenados de forma mais rápida durante a execução de *querys*. (LOSHIN et al. 2014)

1.2 ARMAZENAMENTO

De acordo com Evans (2014), existem dois itens obrigatórios para armazenamento utilizando banco de dados em memória:

- Mídia permanente para armazenamento de transações finalizadas, utilizada para manter a durabilidade e para quando for necessário recarregar o banco de dados para a memória principal;
- Armazenamento permanente para ter a cópia ou backup do banco de dados de forma completa.

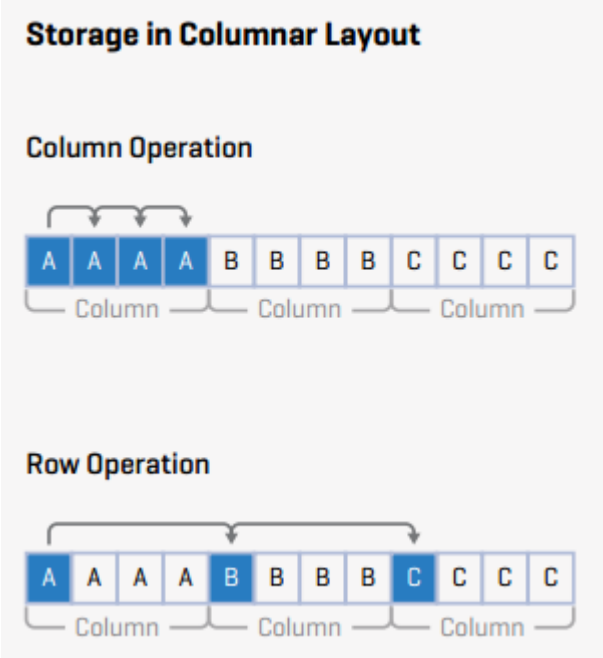
Enquanto são processadas as transações, a performance de disco I/O (*Input/Output*) é o maior gargalo para o desempenho, fazendo com que a diminuição de I/O seja algo significativo. Com isso, a melhor solução seria a utilização de memória *flash*, movendo a memória *flash* para mais próximo do processador reduzindo a latência. No que diz respeito a velocidade para recarregar o banco de dados, claramente a utilização de *flash* tem melhor aproveitamento de tempo. Realizar a leitura de um banco de dados inteiramente armazenado na memória *flash* sempre será muito mais rápido se comparado à leitura do disco rígido. O problema, claro, é o custo da memória *flash* ser maior do que a do disco rígido, e no caso da utilização do IMDB, será acessado de forma esporádica.

1.2.1 Armazenamento em colunas

Conforme Plattner e Leukert (2015), à primeira vista a utilização do armazenamento em colunas (*columnstore*) parece ser muito complicado. Mas, como os dados são sempre armazenados em blocos de certo tamanho, pode-se perceber que para tal função serão acessados muito menos blocos ao utilizar o método de colunas. Os valores são armazenados próximos dos outros como um vetor, permitindo a leitura constante, principalmente se os dados forem codificados, fazendo com que muitos valores fiquem dentro de um mesmo bloco. A CPU reconhece que o processo está em uma sequência lógica, carregada os próximos blocos na memória e os mantém em cache mais próximo da CPU.

Tem-se registro de estudos sobre a utilização de banco de dados orientados por colunas desde o início da década de 1970. Mas somente após os anos 2000 que pesquisas mais profundas e aplicações começaram a utilizar tal método. O mesmo é conhecido pelo ganho de desempenho notável em relação ao modo convencional, que tem sua orientação por linhas. Para melhor entendimento pode-se observar a Figura 2, com um exemplo de armazenamento de dados em coluna. (YAMAN, 2012)

Figura 2 – Layout de armazenamento em coluna



Fonte: Plattner; Leukert, 2015

Na imagem anterior, percebe-se a distância que deve ser percorrida para a localização da informação que está sendo buscada, onde a operação realizada em nível de busca por coluna conterá a informação lado a lado, ocasionando em um retorno muito mais veloz.

Em um banco de dados que tem seu armazenamento em colunas, cada coluna é armazenada em local separado do disco. Os valores armazenados nas colunas são compactados de forma densa, a fim de aprimorar a eficiência para a realização da leitura. Bancos de dados que utilizam o armazenamento em colunas possuem melhor desempenho para a realização de consultas se comparados aos bancos de dados tradicionais, pois são mais eficientes na utilização de I/O para esta operação. Além disso, estes sistemas incluem otimizações para a realização de operações diretas em dados compactados.

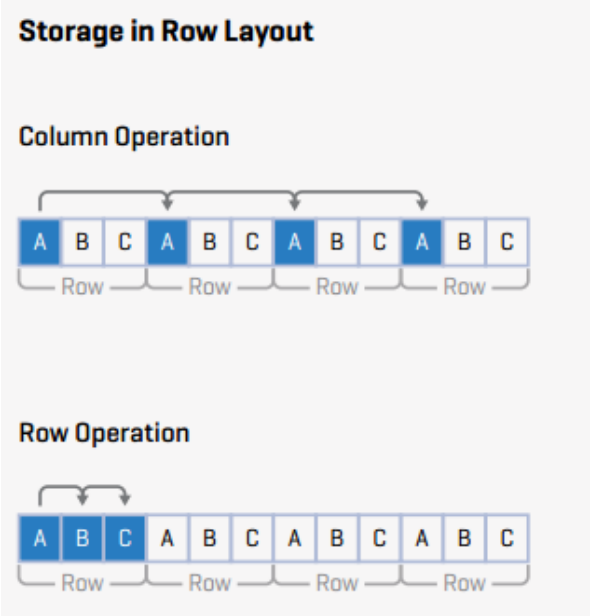
Um dos benefícios mais citados na utilização de armazenamento por colunas é a compressão de dados, que possibilita maiores taxas de compressão se comparadas com os dados armazenados em linha. Esse fato reduz o espaço ocupado no disco e diminui a quantidade e consequentemente o tempo das operações de I/O enquanto os dados estão sendo carregados do disco para a memória. A maior diferença entre a compressão de dados em linha e de coluna pode ser observada quando um mesmo valor se repete na coluna. Neste armazenamento se tem a possibilidade de resumir este dado repetido, enquanto no armazenamento em linha esta opção tem maior dificuldade de ser implementada.

1.2.2 Armazenamento em linhas

De acordo com Plattner e Leukert (2015), o armazenamento em linha é mais eficiente quando se busca um registro por completo, ou seja, uma linha de informações sobre determinado item, conforme Figura 3. Armazenando o conjunto de dados em único bloco no disco, o sistema pode consultar os dados utilizando quantidade menor de operações em disco. Sistemas que utilizam este armazenamento são menos eficientes quando há necessidade de busca em tabelas com muitos registros, ao contrário de buscas em registros específicos.

O inverso do *layout* de armazenamento em coluna acontece para o armazenamento em linha, onde as operações que procuram por registros completos terão grande vantagem por este método, como por exemplo, no caso da existência de uma tabela Pessoa, se a operação executada tiver como objetivo o retorno de todos os dados de uma determinada pessoa, certamente este método apresentará melhor desempenho.

Figura 3 – Armazenamento utilizando o layout de linha



Fonte: Plattner; Leukert, 2015

Armazenamento em coluna tem se mostrado muito superior ao método tradicional se o objetivo for realizar análise sobre os dados em tempo real para a tomada de decisões e aplicações de *Business Intelligence*. Armazenamento em colunas e linhas são imensamente diferentes, onde este armazenamento não pode obter os benefícios de performance do armazenamento em coluna nem mesmo com o particionamento vertical do *schema* ou com a indexação de todas as colunas, assim poderiam ser acessadas de forma independente. (MIAN, 2013)

1.3 INDEXAÇÃO

Conforme Nevarez (2016), apesar de índices serem utilizados para diversos fins, o mais importante, sem dúvidas, é para consultas, isto é, para encontrar um ou mais registros da forma mais veloz possível. Consultas baseadas em índices, utilizam quantidade aleatória de I/O e são extremamente efetivas para um pequeno ou limitado número de registros. Contudo, cada uma destas quantidades aleatórias de I/O normalmente requer que sejam lidas uma boa quantidade de registros.

Então, conforme a base de dados cresce, a quantidade de I/O utilizada aumenta e o processamento tem seu desempenho mais lento. Apesar de tais definições, o desempenho de

consultas em tabelas que utilizam índices sempre será mais veloz, já que o SGBD encontrará o dado mais rápido do que operações com *full join*, que dependem de varredura completa em toda a tabela a fim de obter seu resultado.

Conforme dados apresentados por Microsoft (2016b), a criação de índices (*index*) para o armazenamento em colunas pode proporcionar ganhos de performance na execução de *querys* de até dez vezes em relação a indexação em tabelas tradicionais, e de até dez vezes mais com a compressão de dados. Para altas taxas de compressão e performance, os índices de *columnstore* quebram a tabela em grupos de linhas, que é um conjunto de linhas que são comprimidas no formato de *columnstore*, a quantidade de linhas do agrupamento precisa ter quantidade significativa de dados para obter taxas de compressão melhores e ao mesmo tempo pequenas para se beneficiar da utilização da memória. Os motivos pelos quais a indexação de *columnstores* tem melhores ganhos é que as colunas armazenam valores do mesmo domínio e muitas vezes contém valores similares, fazendo com que haja altas taxas de compressão, acarretando na diminuição do fluxo de I/O e utilizando menos memória.

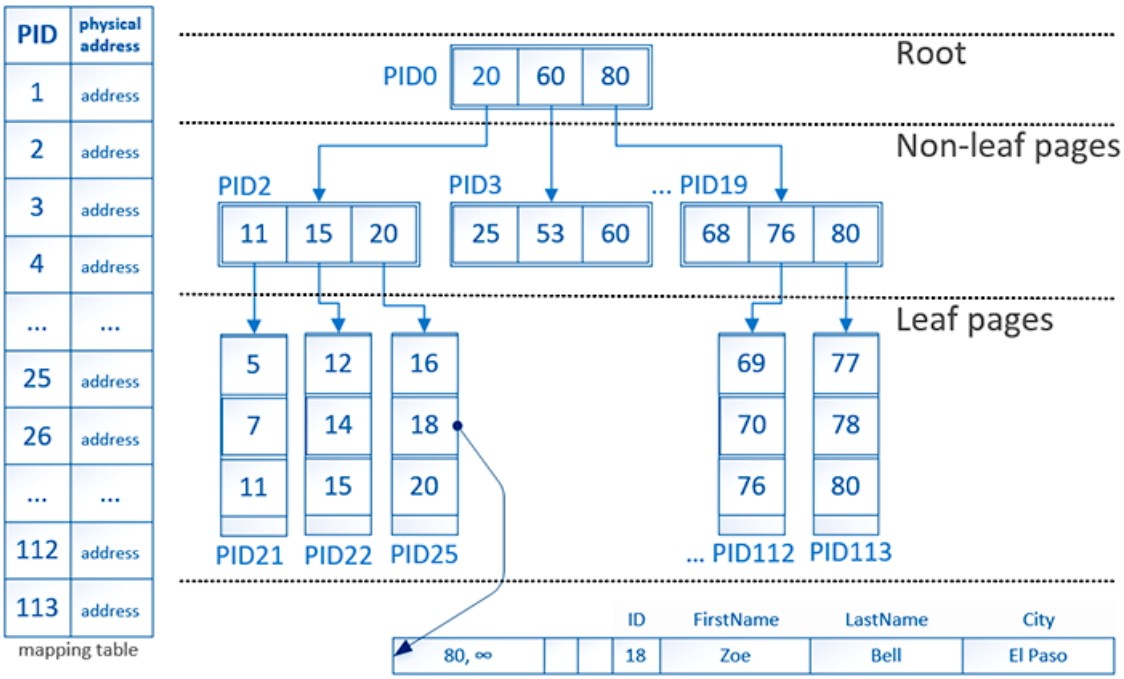
A ferramenta SQL Server, pode utilizar mais de um *nonclustered index* por vez se isso gerar ganho de performance para a *query*. Tal operação é denominada *index intersection* e a utilização desta opção se dará partindo do custo de processamento. O uso de *index* é recomendado para ajudar na performance de *querys* muito utilizadas ou que tem seu desempenho crítico, não sendo recomendada a adição de index de forma descontrolada, o que acarretará em novos problemas de performance e de manutenções no futuro.

Com a utilização do armazenamento em memória, foram introduzidos dois novos tipos de indexação, *memory-optimized nonclustered ind*exes e *hash indexes*, cada um com propósitos diferentes, ambos persistem apenas na memória, não sendo persistentes em disco.

*Memory-optimized nonclustered ind*ex existem apenas em memória e são gerados sempre que o banco de dados em memória fica disponível, ou seja, caso haja a reinicialização do *database* os índices serão recriados. Estes índices são responsáveis por armazenar o endereço de memória para a localização de registros em uma tabela.

Este tipo de índice funciona baseado em um novo conceito denominado BW-Tree, nome dado se distinguir do utilizado em armazenamento em disco, B-Tree. No motor por trás do *In-Memory* OLTP, estes índices são uma estrutura em árvore onde a localização de uma página fica armazenada e quando há a necessidade de alteração desta página ela não é modificada, é criada uma nova página ao invés disso, conforme Figura 4.

Figura 4 – Estrutura de BW-Tree



Fonte: Kosev, 2016

Percebe-se que a estrutura de BW-Tree é muito similar a B-Tree, onde a diferença principal é que uma tabela mapeada é utilizada para referenciar páginas e que o índice das páginas não é alterável após sua criação. Esta é a chave para a performance obtida com o uso de *nonclustered indexes* pois ajuda a isolar os efeitos da divisão de um nodo atualizando só o nodo necessário ao invés de atualizar toda a estrutura do índice. (KOSEV, 2016)

Hash index são os responsáveis por prover a melhor performance possível por serem capazes de encontrar registros únicos mais rápido do que qualquer outro tipo de index do SQL Server. Has*h index* utilizam um novo parâmetro, sendo assim necessário estabelecer o valor inicial para o tamanho do *bucket*. *Buckets* são basicamente agrupamentos de linhas, onde são aplicados os *hash* para mapear os itens presentes no *bucket*. (NEVAREZ, 2016)

Como exemplo, pode ser observado na Figura 5 que a mesma contém o *hash index* na coluna de Cidades que possui *bucket* de 8 valores. Utilizando o index para encontrar a linha de valor San Francisco, será utilizado o *bucket* de valor 2.

Figura 5 – Utilização de *Hash Index*

Fonte: Nevarez, 2016

Contudo, colisões podem ocorrer com a utilização de *Hash index*, se o objetivo for consultar o local de valor igual a Prague, a função do *hash* retornará corretamente o *bucket* de número 5. Entretanto, para o mesmo *bucket* existe mais de um valor, o banco de dados terá de procurar dentro da lista o valor solicitado. Não serão observados problemas enquanto a quantidade de valores para o *bucket* não for grande, porém a performance pode sofrer impactos a medida que a quantidade de valores para o *bucket* aumente.

O valor do *bucket* deve ser criado juntamente com a tabela, sendo necessário estimar e planejar o melhor valor de *buckets* a ser utilizado. A melhor performance é obtida quando a quantidade de valores por *bucket* é menor, assim é recomendado que a quantidade de valores para o *bucket* seja igual ou o dobro de valores únicos possíveis para o index. (NEVAREZ, 2016)

1.4 DURABILIDADE

Uma transação é considerada durável uma vez que finalizada, todas as alterações realizadas por ela são gravadas de forma permanente no sistema. Existem configurações para que exista a proteção da operação em questão, prevenindo assim a possível perda de algum dado, até mesmo em caso de falha no sistema. Armazenando os registros de *log* que a transação realizou, o sistema pode utilizar este para a realização de *backup* até mesmo se o *hardware* apresentar algum problema. O conceito de durabilidade permite que se tenha o conhecimento de que toda transação concluída fará parte do sistema, independente do que acontecer com o sistema no futuro. (MICROSOFT, 2016c)

1.4.1 Durabilidade das transações

Com a utilização do SQL Server, as transações podem ser tanto completamente duráveis, o padrão do SGBD, como duráveis com atraso (*delayed durable*). Transações completamente duráveis são síncronas, fazendo com que o retorno de sucesso seja retornado após o registro ser gravado em disco. Já as duráveis com atraso são assíncronas, retornando sucesso até mesmo antes do dado estar registrado em disco. Ambos os controles têm suas vantagens e desvantagens, onde uma mesma aplicação pode se utilizar de ambas as soluções, sendo necessária análise de qual a necessidade da aplicação. Tais atributos de controle de transações são aplicados na criação ou alteração do banco de dados, conforme *script* que pode ser observado na Figura 6.

Figura 6 – Atribuição de durabilidade de transações

```
ALTER DATABASE … SET DELAYED_DURABILITY = { DISABLED | ALLOWED | FORCED }
```

Fonte: Microsoft, 2016a

O uso de transações completamente duráveis é utilizado quando não se pode perder nenhum dado e o gargalo do sistema não é devido a latência de I/O. Garante que quando a transação for finalizada com sucesso, as alterações ficarão visíveis para as demais transações que estão em andamento, tem sua durabilidade garantida no momento do *commit*.

Transações duráveis com atraso garantem que o retorno para o cliente será anterior ao registro dos dados em disco, não tendo de aguardar I/O para ser finalizado. Pode ser utilizado quando são toleráveis perdas de dados, como exemplo, quando alguns dados de indivíduos não são críticos enquanto se tem dados mais importantes. Se a geração de dados I/O está causando gargalos ao sistema acarretando em atrasos para o cliente, a aplicação pode liberar a utilização enquanto realiza as operações de forma que o cliente não perceba tais execuções.

1.4.2 Durabilidade dos objetos

Para a utilização da persistência dos dados no SGBD, se tem a possibilidade de manter somente o *schema* ou o conjunto de *schema* e dados. O padrão do SGBD é manter ambos, o

que impacta nos dados é a definição da base de dados, se está com a configuração de completamente duráveis ou duráveis com atraso. Caso haja a reinicialização do servidor, as tabelas em memória são instanciadas novamente, para recuperar os dados para o estado anterior a reinicialização.

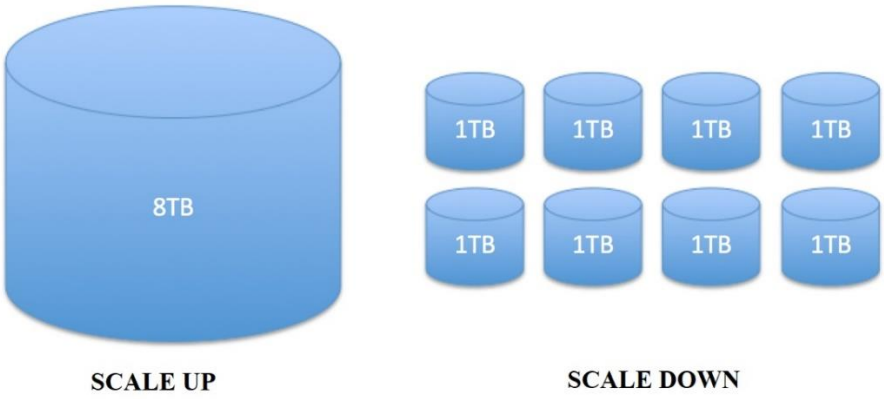
A opção de manter somente o *schema*, entretanto, faz com que somente as tabelas fiquem salvas, acarretando na perda de todos os dados caso ocorra a reinicialização do servidor, o mesmo que ocorre com a criação de tabelas temporárias. Esta opção é utilizada quando não se tem necessidade de mantes os dados, onde os mesmos serão utilizados para alguma operação rápida e somente uma vez. (MICROSOFT, 2017a)

1.5 ESCALABILIDADE

Segundo Microsoft (2012), escalabilidade é a habilidade da aplicação utilizar de forma eficiente os recursos disponíveis para assim realizar mais operações. Como exemplo, uma aplicação que pode atender a quatro usuários em um único processador e pode atender a quinze usuários utilizando quatro processadores é considerada escalável. Se a adição de processadores não aumentar a quantidade de usuários atendidos, então a aplicação não é escalável.

Conforme a Figura 7, existem dois tipos de escalabilidade: *scale-up* e *scale-down,* que também são conhecidas como escalabilidade vertical, sendo o aumento e a diminuição do tamanho do servidor, respectivamente. Assim, passando de um servidor de 4 processadores para 64 ou até mesmo 128 processadores, por exemplo. Isto é o que normalmente acontece quando se é desejado o aumento, pois quando o banco de dados fica sem recursos por conta do hardware, são adquiridos mais processadores e memória, não sendo necessárias alterações significativas do *database*.

Figura 7 – Comparação entre *Scale-up* e *Scale-down*



Fonte: Appleby, 2014

Scale-out significa que será realizada a expansão para múltiplos servidores ao invés de se manter em um único e maior. A vantagem inicial é de que o custo é menor em relação a verticalização, pois 8 conjuntos de 4 processadores é geralmente mais barato do que um único de 32 processadores, contudo, este valor é anulado quando o custo de licenciamento e manutenção é incluso.

O fator crítico para a escolha da melhor estratégia de escalabilidade é reconhecer que aplicações devem atender a diversos tipos de dados, e que cada dado necessita diferentes requisitos em uma arquitetura escalável. Quando se tem somente um *database*, é mais simples fazer com que o tratamento de dados seja o mesmo para todos os tipos, mas quando são separados os *databases* e se tem a necessidade de replicações, é importante saber como este dado será usado, assim pode-se definir qual solução deve ser escolhida.

1.6 MS SQL SERVER X CONCORRÊNCIA

Com as melhorias apresentadas na ferramenta SQL SERVER 2016, o mesmo ganhou grande destaque em diversas publicações especializadas em SGBDs. Segundo a Solid IT (2017a), o SGBD da Microsoft encontra-se na terceira posição no ranking dos bancos de dados mais utilizados em maio de 2017, conforme mostra o Quadro 1. O ranking é baseado em quantas menções a ferramenta tem em diversas situações, como a quantidade de menções ao sistema, interesse do público alvo, quantidade de discussões técnicas e vagas de emprego que buscam profissionais especialistas nas ferramentas.

Quadro 1 – SGBDS mais utilizados

Ranking	SGBD	Pontuação
1	Oracle	1354.31
2	MySQL	1340.03
3	**SQL SERVER**	**1213.80**
4	PostgreSQL	365.91
5	MongoDB	331.58

Fonte: Solid IT, 2017b

De acordo com a documentação de algumas das principais fabricantes de SGBDs, foi realizada a comparação de algumas das principais características, as quais estão apresentadas no Quadro 2 que compara as soluções Oracle, Microsoft SQL Server e também a solução da HANA, da empresa SAP. As informações gerais técnicas foram obtidas de SOLID IT (2017b), que possui compilado de características dos produtos;

Quadro 2 – Comparação entre SGBDs

SGBD	TimesTen	HANA	SQL Server
Versão	12.1 – 2014	SPS11 - 2015	2016
Desenvolvedor	Oracle	SAP	Microsoft
OLTP em memória	X	X	X
Armazenamento em coluna	X	X	X
Compressão em memória	X	X	-
Espaço não limitado	X	X	X
Persistência no armazenamento colunar	X	X	X
Coluna de armazenamento exclusivo em memória	-	X	-
Integração com a linguagem R	X	X	X
Linguagem SQL	PL/SQL	SQL Script	T-SQL
Suporte XML	X	-	X
Conceito de consistência	ACID	ACID	ACID

Ranking menos vulneráveis[1]	3º	2º	1º
Custo da licença Enterprise anual[2]	U$ 6.650	-	U$ 2.850
Custo da licença banco de dados em memória[2]	U$ 3.220	-	Incluso

Fonte: do Autor, 2017

Com a coleta das principais características, pode-se observar que os SGBDs citados no quadro anterior possuem grande semelhança entre suas configurações, o que pode acarretar em que uma das principais características consideradas por grande parte das empresas que as utilizam seja o preço, que a solução da Microsoft tem vantagem.

[1] Segundo o National Institute of Standards and Technology Comprehensive Vulnerability Database, que recebe relatórios de vulnerabilidades encontradas em diversos sistemas.
[2] *Website* que compila custos de licenciamento entre alguns fornecedores de SGBD.

2. ESTUDO DA FERRAMENTA - SQL SERVER

A ferramenta a ser utilizada para estudo e experimentos deste trabalho de conclusão de curso é a versão mais recente do SGBD da Microsoft, o SQL Server 2016. A versão 2016 recebeu novos recursos e otimizações de recursos já existentes para melhorar a performance e fazer com que as funções em memória iniciadas, mas não difundidas, no SQL Server 2014 fossem mais exploradas. Com a necessidade cada vez mais rápida de se obter dados por parte dos usuários, esta versão do SQL Server incluiu diversas opções para aperfeiçoar a execução de *queries*. Tabelas otimizadas em memória passaram a suportar cargas maiores de *online transaction processing*, com melhores taxas de transferência como resultado das melhorias de tratamento de operações paralelas. (MICROSOFT, 2016e)

De acordo com a Microsoft (2016f), o único requisito necessário para a utilização de tabelas com armazenamento em memória é de que se tenha disponível a quantidade de memória suficiente para o armazenamento de seus dados, não havendo limite na quantidade de dados em uma única tabela. O tamanho utilizado para o armazenamento em memória corresponde ao tamanho total dos dados mais uma quantidade extra para armazenamento dos cabeçalhos das linhas de registro.

Para o processamento de grandes cargas de trabalho, existe a necessidade de memória adicional para o processamento das operações e criação do versionamento das linhas. A quantidade de memória depende da carga de trabalho, contudo, a recomendação é que seja o dobro da memória ocupada pelas tabelas envolvidas e seus índices. (MICROSOFT, 2016f)

2.1 ARQUITETURA

O SQL Server tem a capacidade de utilizar e liberar recursos de memória conforme sua necessidade, de forma dinâmica, onde o motivo para o gerenciamento de tal recurso consiste na não necessidade de limitar no próprio sistema a quantidade a ser utilizada. O espaço utilizado pelo SQL Server pode ser dividido em duas regiões, a região do *buffer pool* e o restante. O *buffer pool* é utilizado como o principal repositório de recursos. (MICROSOFT, 2016e)

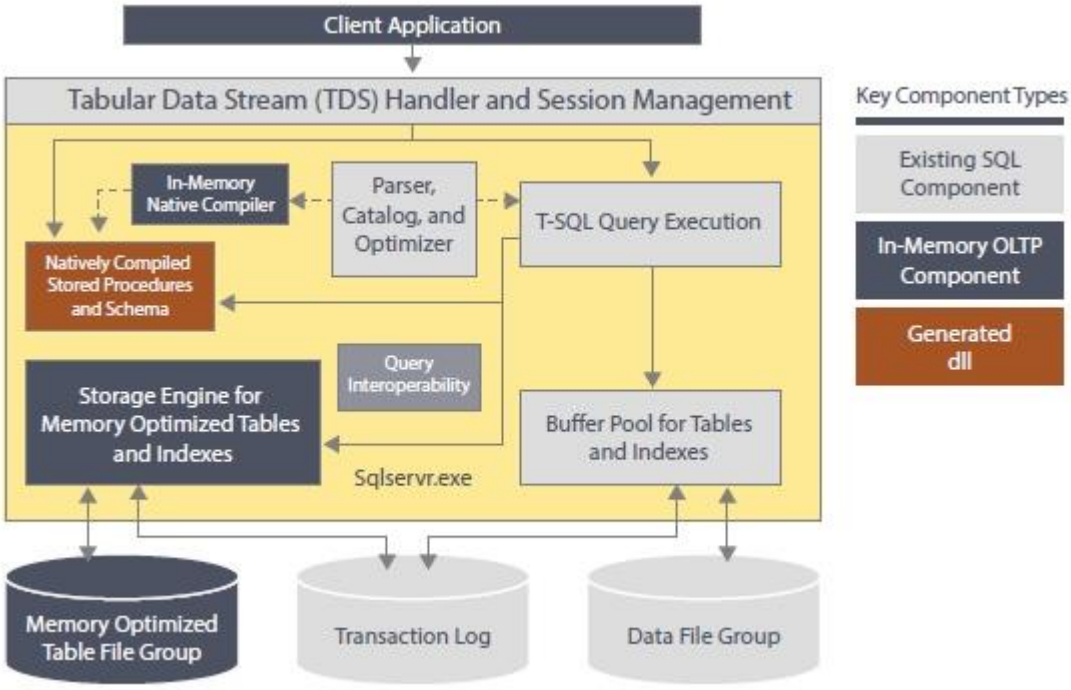
De acordo com Microsoft (2016e), o objetivo principal da utilização de um banco de dados é para o armazenamento e consulta de dados. Portanto, lidar com o intenso volume de dados é uma das principais características do motor do banco de dados. Por conta do alto consumo de recursos das operações de I/O, o SQL Server foca na utilização de recursos de

forma mais eficiente, onde o gerenciamento do *buffer* é a chave principal para tal eficiência. O gerenciamento do *buffer* faz com que se tenha o acesso e atualização das páginas de dados, enquanto o *buffer pool* reduz o consumo de I/O.

Quando o SGBD é inicializado, é computado o espaço de endereçamento virtual para o *buffer pool*, baseado em uma significativa quantidade de parâmetros, como a quantidade de memória física do sistema, número de *threads* do servidor e vários parâmetros de inicialização. É reservado o valor computado para o *buffer pool*, mas a quantidade de memória física utilizada é somente o necessário para atender a requisição atual. Conforme novos usuários se conectam e executam novas *queries*, o SGBD libera mais recursos para utilizar nas demandas até que se chegue ao valor máximo de memória disponível no servidor.

Conforme Otey (2014), o novo motor por trás da tecnologia *In-Memory* do SQL SERVER, apresentado na Figura 8, utiliza um novo mecanismo de controle de concorrência em conjunto com algoritmos otimizados para dados residentes em memória. Quando um registro armazenado em um buffer compartilhado é alterado, o motor do *In-Memory* cria uma nova versão de todo esse registro com a data de quando foi realizado. Tal processo é realizado todo em memória, sendo de rápida conclusão.

Figura 8 – Arquitetura do SQL Server



Fonte: Otey, 2014

Após a realização da operação, o motor analisa e valida todas as linhas alteradas e realiza o *commit*. Este método é mais rápido e mais escalável do que o tradicional mecanismo de bloqueio utilizado pelo SQL Server quando em disco, pois, não são realizados bloqueios ou quaisquer estados de espera que façam o processador não utilizar a sua capacidade máxima.

Com a criação das novas linhas, as antigas se tornam inúteis para a utilização de *queries* e ocupam espaço em memória. Para lidar com isso foi desenvolvida uma rotina que remove todas as linhas que foram inutilizadas, liberando memória para o restante do banco de dados.

Partindo da análise da figura anterior, todo o conjunto em cinza mais claro pertence a arquitetura tradicional do banco de dados em disco, sendo todo o restante para a utilização das operações em memória. Após a entrada da instrução no SGBD, é realizada a análise e transcrita para o formato que o banco de dados interpreta, com a análise concluída é possível saber se o procedimento referência a estrutura em memória ou disco, levando assim para a sequência de passos correspondente.

Para a utilização do *In-Memory*, foi disponibilizada a função *In-Memory Native Compiler* que interpreta a instrução T-SQL e compila, gerando uma DLL. O objetivo é a redução do número de instruções que a CPU executa para o processamento da query. O resultado disto é que o conjunto da nova estrutura de processamento de *querys* com as procedures compiladas são o fator principal da alta performance gerada com a tecnologia *In-Memory* do SQL Server.

2.2 OLTP NA MEMÓRIA

In-Memory OnLine Transaction Processing, também conhecido como OLTP na memória, é a tecnologia desenvolvida pela Microsoft para aumentar a performance de aplicações que utilizam como banco de dados o SQL Server. Em diversas publicações é chamado de Hekaton, que foi o nome dado ao projeto de desenvolvimento. Utiliza-se principalmente de tabelas otimizadas em memória (*memory-optimized table*) e de procedimentos armazenados compilados nativamente (*natively compiled stored procedures*).

De acordo com Korotkevitch (2016), o motor do *In-Memory OLTP* é completamente integrado com o motor do SQL Server, que é a chave principal para o sucesso da solução em comparação com seus concorrentes que possuem soluções de armazenamento em memória.

No SQL Server não é necessária a refatoração de sistemas complexos, nem separar dados entre servidores de memória principal e convencionais, nem mesmo ter que mover todos os dados para a memória principal.

Pode-se separar dados armazenados em memória e disco individualmente entre as tabelas, assim sendo possível mover os dados mais utilizados para a memória principal e mantendo dados pouco acessados em disco.

2.2.1 Tabelas otimizadas em memória

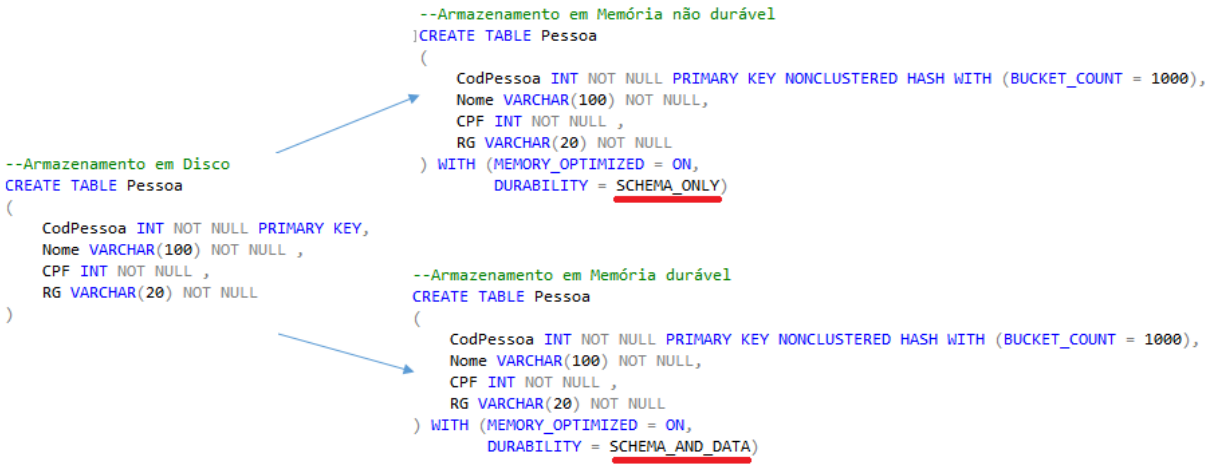
Segundo Arumilli (2016), as tabelas otimizadas em memória têm seus dados armazenados em memória principal utilizando múltiplas versões de cada linha de dados. Esta técnica é caracterizada como "*non-blocking multi-version optimistic concurrency control*", eliminando qualquer bloqueio, obtendo assim significantes vantagens de desempenho. As características principais das tabelas otimizadas em memória são:

- Linhas da tabela são lidas e escritas na memória;
- A tabela está completamente na memória;
- As opções de dados duráveis e não duráveis para a tabela;
- Uma cópia é mantida em disco para fins de durabilidade;
- Dados são lidos diretamente do disco apenas em caso de recuperação da base de dados;

As tabelas otimizadas em memória, são criadas como duráveis, por padrão. Assim sendo, em caso de falhas no servidor, os dados serão recuperados do log de transações. Contudo, pode-se atribuir a característica de não durável de acordo com a Figura 9.

Assim, as transações nessa tabela não terão necessidade de I/O em disco. Porém, em caso de falhas no servidor, os dados desta tabela não serão recuperados.

Figura 9 – Atribuição de característica de durabilidade



Fonte: Do autor, 2017

2.2.2 Procedimentos armazenados compilados

Segundo Arumilli (2016), um procedimento armazenado compilado nativamente é um objeto do SQL Server que somente pode acessar estruturas de dados com memória otimizada, demonstrado na Figura 10. As suas características principais são:

- São compilados para código nativo (DLL) após a sua criação;
- Otimizações agressivas consomem tempo enquanto são compiladas;
- Somente pode interagir com tabelas otimizadas em memória;
- A chamada desta *procedure* é uma chamada para o ponto de entrada da DLL.

Figura 10 – Exemplo de procedimento compilado

```
CREATE PROCEDURE spAtualizaNome
    with native_compilation, schemabinding
as
begin
atomic with (Transaction Isolation Level = SNAPSHOT, Language = N'us_english')

    Update Pessoa
    set Nome = Nome + 'Teste'

end
```

Fonte: Do autor, 2017

2.2.3 Benefícios da utilização de OLTP em memória

De acordo com Artemiou (2015), o desempenho obtido com a utilização de OLTP apresenta melhores resultados dependendo da necessidade de operações a serem realizadas. Há recomendações da Microsoft que especificam as principais cargas de trabalho e os maiores benefícios que terão com o uso de OLTP. O Quadro 3 sumariza estes tipos de cargas de trabalho.

Quadro 3 – Relação de benefícios por carga de trabalho com OLTP em memória

Tipo de carga de trabalho	Exemplos	Principais benefícios da utilização de OLTP em memória
Altas taxas de inserção de dados	Medição inteligente Sistema de telemetria	Elimina contenção Minimiza I/O
Performance de leitura	Navegação em rede social	Elimina contenção Eficiente recuperação de dados Minimiza tempo de execução de códigos Uso de CPU mais eficiente
Processamento massivo de registros	Cadeia de fornecimento de manufatura ou varejistas	Elimina contenção Minimiza tempo de execução de códigos Eficiente processamento de dados
	Plataformas de jogo online	Minimiza tempo de execução de códigos Eficiente recuperação de dados

Fonte: Artemiou, 2015

2.3 COMPRESSÃO DOS DADOS

A compressão de dados é utilizada para reduzir o tamanho das tabelas, tendo a possibilidade de escolher entre dois métodos. O primeiro é por linha, que reduz o tamanho das linhas utilizando um outro formato de linha que elimina o espaço de armazenamento que não está sendo utilizado, baseado no tamanho especificado na criação. O segundo método é por página, que trabalha com os dados armazenados em uma página, removendo sequências de bytes duplicados. (KOROTKEVITCH, 2016)

Além do objetivo principal, que é a redução de espaço, a compressão de dados pode ajudar a melhorar a performance de I/O quando trabalhando com grandes cargas de dados, pois os dados são armazenados em menos páginas e as *queries* precisam ler menos páginas do disco. Contudo, são necessários mais recursos da CPU para que o servidor consiga comprimir e descomprimir os dados.

2.3.1 Compressões por linha

As tabelas com colunas de tamanho fixo, sempre utilizam o mesmo espaço de armazenamento, baseado no maior valor possível do tipo de dado utilizado. Por exemplo, uma coluna INT sempre usará 4 *bytes*, até mesmo quando o valor armazenado for 0 ou NULL. Isto pode ocasionar um rápido crescimento no tamanho dos dados. Por exemplo um *byte* não utilizado leva a geração de 350MB de espaço não utilizado em 1 ano, em uma tabela que registra 1 milhão de linhas por dia. A Figura 11 apresenta a aplicação da compressão por linha, fazendo com o espaço em disco vazio seja aproveitado de forma eficiente.

Figura 11 – Aplicação de compressão em linha

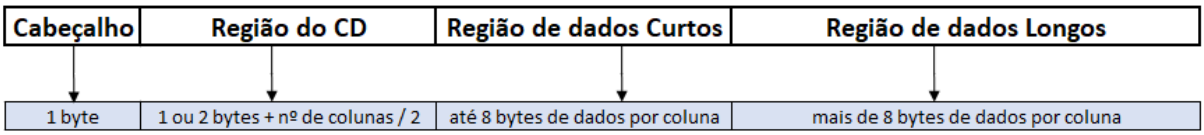
Fonte: Do autor, 2017

No exemplo do valor 125, o *Int* pode ser convertido para *TinyInt* gerando economia de 3 bytes, o mesmo ocorre com o valor “SQL” armazenado na coluna *Char* (20) que pode ser compresso no tipo *Char* (3).

A tabela utiliza mais espaço em disco e no *buffer*, o que acarreta na necessidade de mais operações de I/O, tornando o sistema menos eficiente. A compressão por linha elimina este problema com a implementação de outro formato de linha, denominado CD (do inglês *Column Descriptor*). Com este formato, todas as linhas armazenam a coluna e a descrição dos dados da linha, utilizando o espaço exato necessário para o valor. A Figura 12 representa o formato CD de armazenamento. (KOROTKEVITCH, 2016)

Ainda de acordo com Korotkevitch (2016), dados em linhas de formato CD, são separados em dois conjuntos, Região de Dados Curtos (*Short Data Region*) e Região de Dados Longos (*Long Data Region*). Tal separação é mediante a análise do tamanho do dado, e não no tipo de dado armazenado. A região curta armazena dados com tamanho até 8 bytes, dados maiores que 8 bytes são armazenados na longa.

Figura 12 – Formato de linha *Column Descriptor*



Fonte: Do autor, 2017

O *byte* do cabeçalho (*header*) é um *bitmask*, que consiste em um conjunto de bits que representam as propriedades da linha, se é um *index*, se tem controle de versão, se foi deletada e diversos outros atributos. A região CD armazena as informações dos dados nas colunas da linha, iniciando com a informação em 1 ou 2 bytes indicando quantas colunas estão presentes no *array* do CD. O primeiro *bit* indica se existe mais de 127 colunas, assim sendo, teria a necessidade de utilizar 2 *bytes* para o armazenamento do número de colunas. Cada elemento do *array* armazena informações de uma das colunas. (KOROTKEVITCH, 2016)

2.3.2 Compressões por página

Esta compressão é aplicada na página inteira, mas somente após a mesma estar completamente preenchida e a compressão resultar em ganhos significativos de espaço. A compressão por página consiste em aplicar a compressão por prefixos em cada coluna, onde é localizado e reutilizado o prefixo mais comum, o que causa a redução de tamanho dos valores da coluna, e após é aplicada a compressão por dicionário, removendo todos os valores duplicados de todas as colunas.

Para comprimir os prefixos identificados na Figura 13, o SQL Server define um registro âncora, que é um registro da tabela como outro qualquer, com a característica de ser o maior valor que contém prefixo duplicado. O registro âncora é utilizado posteriormente para criar novamente a representação dos valores da coluna quando forem acessados, este tipo de registro só pode ser acessado internamente pelo SGBD, não retornando em resultado de *queries*. A compressão do prefixo substitui o prefixo de cada valor, que possa ser comprimido pelo valor âncora, com um indicador de quantos *bytes* podem ser "copiados" do valor âncora e se caso for o valor exato da ancora o mesmo retornará nulo. (CEBOLLERO, 2015)

Figura 13 – Página com registros duplicados e após comprimidos

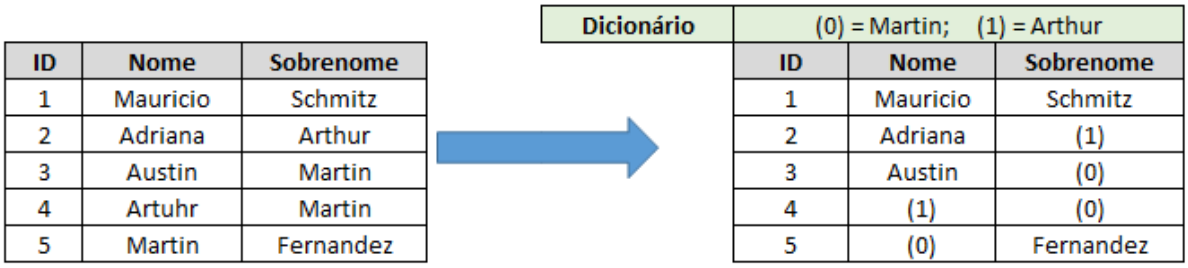
ID	Nome	Sobrenome
123	Alexander	Smith
789	Alexandra	Smith
1234	Alexis	Smith
1287	Andrew	Smith
1789	Austin	Smith

	ID	Nome	Sobrenome
Registro Ancora	1234	Alexander	Smith
	[3]	[NULL]	[NULL]
	789	[7]ra	[NULL]
	[NULL]	[4]is	[NULL]
	[2]87	[1]ndrew	[NULL]
	[1]789	[1]ustin	[NULL]

Fonte: Cebollero, 2015

Conforme Cebollero (2015), a compressão por dicionário é o segundo tipo de compressão utilizado na página, onde é criado um dicionário de valores que ocorrem múltiplas vezes entre todas as colunas e linhas da página. Então estes valores duplicados são substituídos por índices do dicionário, conforme Figura 14 onde os valores Arthur e Martin são adicionados ao dicionário e substituídos na página por índices do dicionário. O valor Martin é substituído pelo valor de índice (0) em todos os locais da página em que o valor é igual, e da mesma forma o valor Arthur é substituído pelo índice (1).

Figura 14 – Aplicação de dicionário de dados



ID	Nome	Sobrenome
1	Mauricio	Schmitz
2	Adriana	Arthur
3	Austin	Martin
4	Artuhr	Martin
5	Martin	Fernandez

Dicionário	(0) = Martin; (1) = Arthur		
	ID	Nome	Sobrenome
	1	Mauricio	Schmitz
	2	Adriana	(1)
	3	Austin	(0)
	4	(1)	(0)
	5	(0)	Fernandez

Fonte: Cebollero, 2015

O Capítulo 4 conterá uma seção destinada a realização de experimento com a utilização de consultas em tabelas com armazenamento em disco sem compressão e com compressão de dados, a fim de verificar a diferença no tempo de processamento quando os dados são comprimidos.

3. PREPARAÇÃO DOS DADOS

Neste capítulo será apresentada a metodologia utilizada para o estudo da ferramenta SQL Server, a aplicação utilizada para o desmembramento dos arquivos e sua posterior importação e a base de dados utilizada para os estudos de desempenho. Um dos objetivos na utilização do SQL Server foi a possibilidade de comparar com os resultados previamente obtidos no estudo realizado por Horn (2016) sobre a ferramenta da Oracle, o TimesTen.

Para maior integridade dos resultados, a base de dados utilizada foi a mesma utilizada por Horn (2016) e cedida pelo mesmo. Tal base é disponibilizada livremente no fórum do *StackOverflow*, que se trata de um fórum de perguntas e respostas destinado a interessados na área de desenvolvimento de software. A aplicação para a divisão dos arquivos também foi utilizada a base da aplicação desenvolvida por Horn (2016). Contudo, foram criados novos métodos para a realização dos *scripts* de inserção.

3.1 METODOLOGIA

A metodologia iniciou-se com estudos sobre a parte teórica dos bancos de dados relacionais e em memória, através de pesquisas bibliográficas, nas quais foram adquiridas informações em diversos artigos científicos, livros e dos manuais da Microsoft, fabricante da ferramenta de estudo. Teve sequência a análise técnica e prática do SGBD SQL Server, que tem disponibilidade da utilização de memória e disco em uma única ferramenta.

Esta pesquisa possui natureza aplicada e tem como objetivo de estudo a comparação de armazenamento em disco e em memória do SQL Server 2016, bem como da comparação com os resultados obtidos no estudo do aluno Horn (2016) com a utilização da ferramenta TimesTen, da Oracle.

Com as informações necessárias para a inicialização dos experimentos, criou-se duas bases de dados, uma para a realização do estudo em memória principal, e a outra para os testes com armazenamento em disco rígido. Ambas as bases de dados serão criadas no mesmo servidor e serão preenchidas com os mesmos registros, para que assim se tenha uma comparação equivalente.

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 CONFIGURAÇÕES DO SERVIDOR

Para a realização dos experimentos, foi solicitado junto a Universidade Feevale a disponibilização de um servidor para os testes deste estudo. Um dos propósitos deste trabalho é produzir conhecimentos para a futura migração do SQL Server 2014 atualmente utilizado pela Universidade Feevale para a versão em estudo, 2016. O Servidor possui as características mencionadas no Quadro 4:

Quadro 4 – Características do servidor

Sistema Operacional	*Windows Server 2016 Standard*
Processador	Intel (R) Xeon (R) CPU E5506 2.13Ghz
Memória RAM	144 GB
Núcleos físicos	8
Unidade de Disco	2 Unidades, D: 418 GB e E: 557 GB
Valor Aproximado	R$ 50 mil

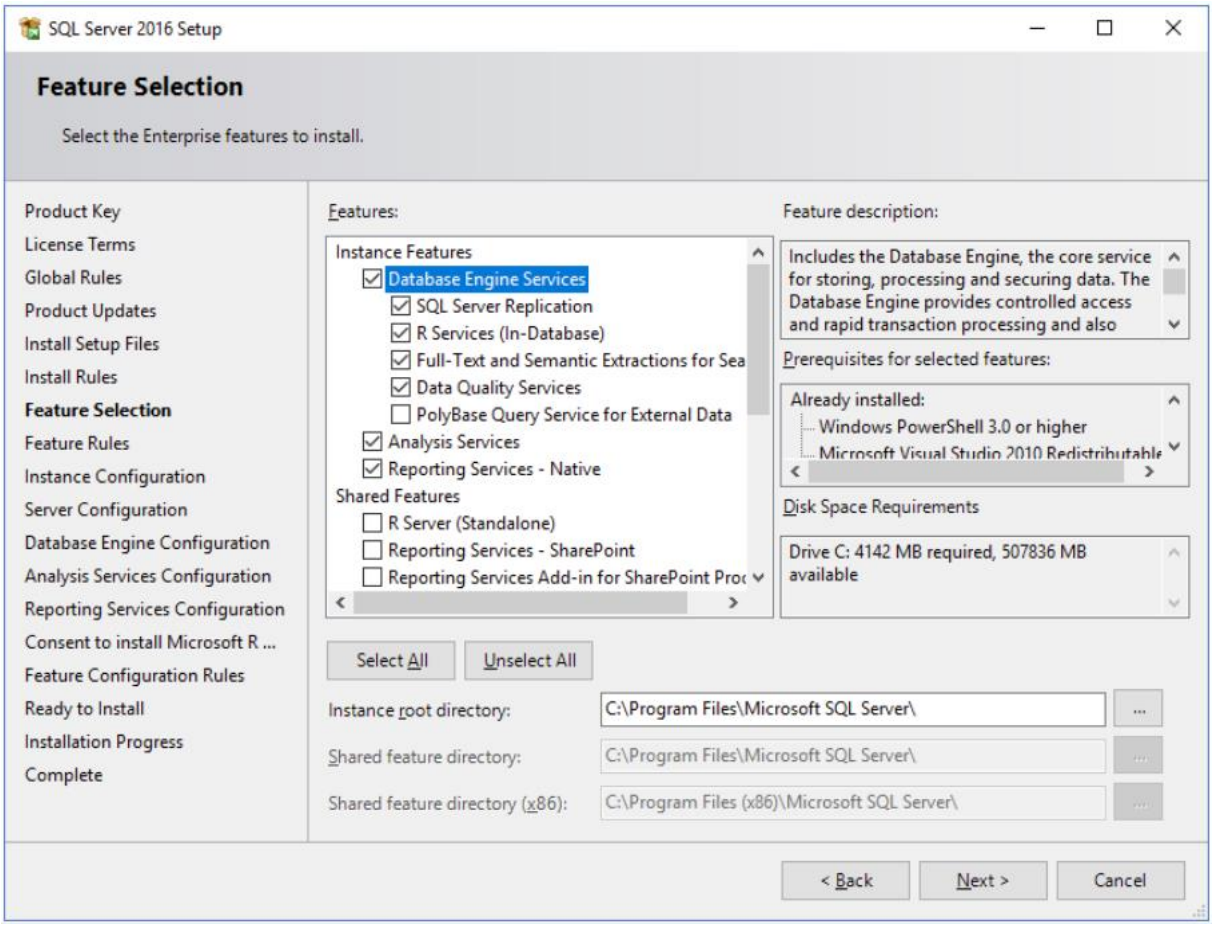
Fonte: do Autor, 2017

Aproveitando que o servidor possui dois discos, optou-se por distribuir as duas bases de dados entre eles, então a base de dados em memória teve seus arquivos vinculados a unidade E, enquanto a versão de armazenamento em disco ficou na unidade D. Isto também foi necessário pois não seria possível colocar as duas bases de dados em um único disco, considerando os tamanhos disponíveis, pois o somatório do espaço necessário ultrapassa o tamanho individual, mesmo da maior unidade.

Considerando que as duas unidades possuem as mesmas características de desempenho, esta divisão, além de melhorar a distribuição dos dados não causa diferença no desempenho.

Realizou-se a instalação do SQL Server através do *SQL Server Installation Center*, ferramenta que facilita a instalação de grande parte dos componentes e centraliza as funções disponíveis, conforme Figura 15.

Figura 15 – Guia de instalação



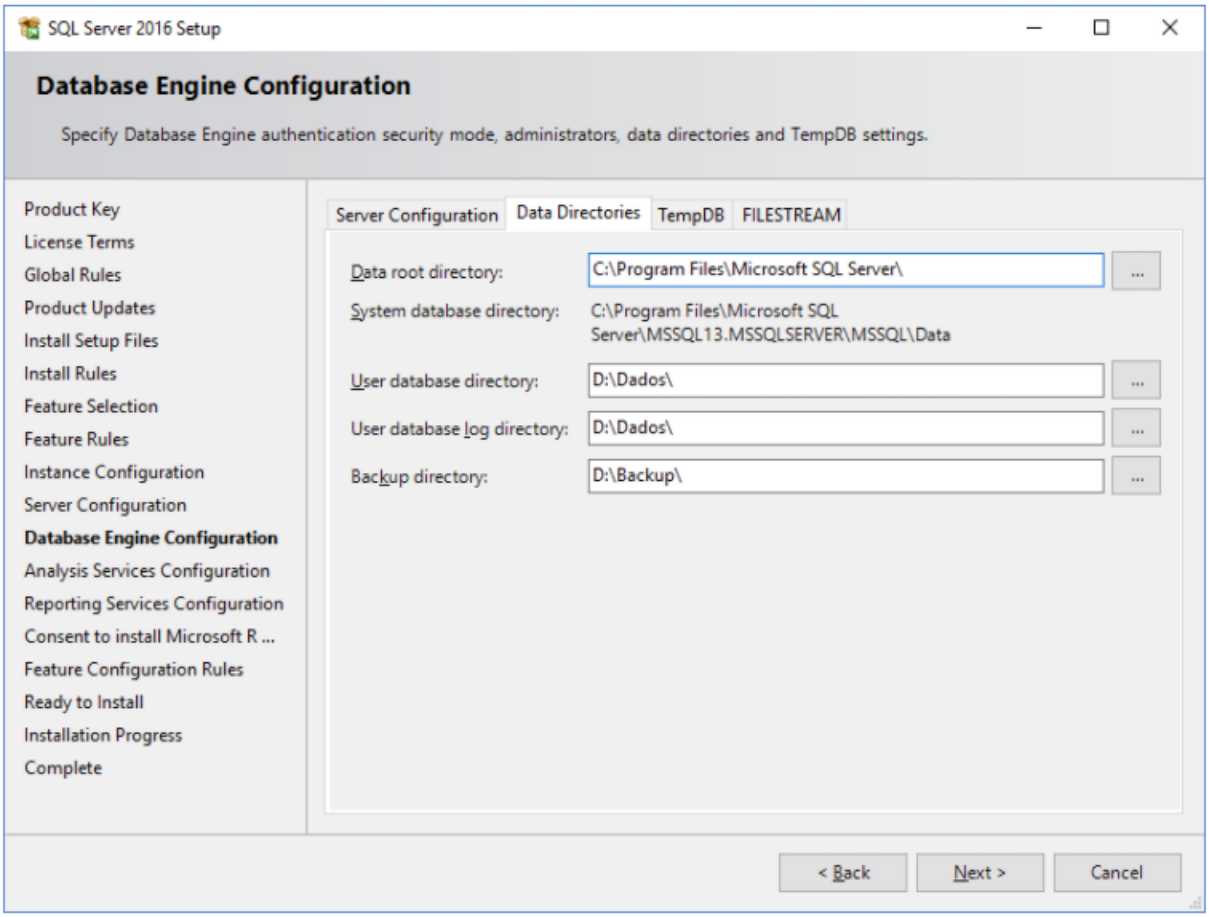
Fonte: do autor, 2017

Foram selecionados os componentes a serem instalados e estipulada a quantidade de instâncias que seriam utilizadas para a realização dos experimentos (utilizada somente uma instância).

Depois disto, foi realizada a configuração do banco de dados, sendo definidos os usuários e suas permissões de acesso iniciais, bem como os métodos que serão aceitos para a

realização da autenticação com a instância. Foi configurada também a localização onde os arquivos físicos foram armazenados dentre eles, os arquivos de definições do banco de dados, *logs* de alterações para toda e qualquer alteração da configuração e arquivos de *backup* para caso haja algum dano na base de dados, configurações estas apresentadas na Figura 16.

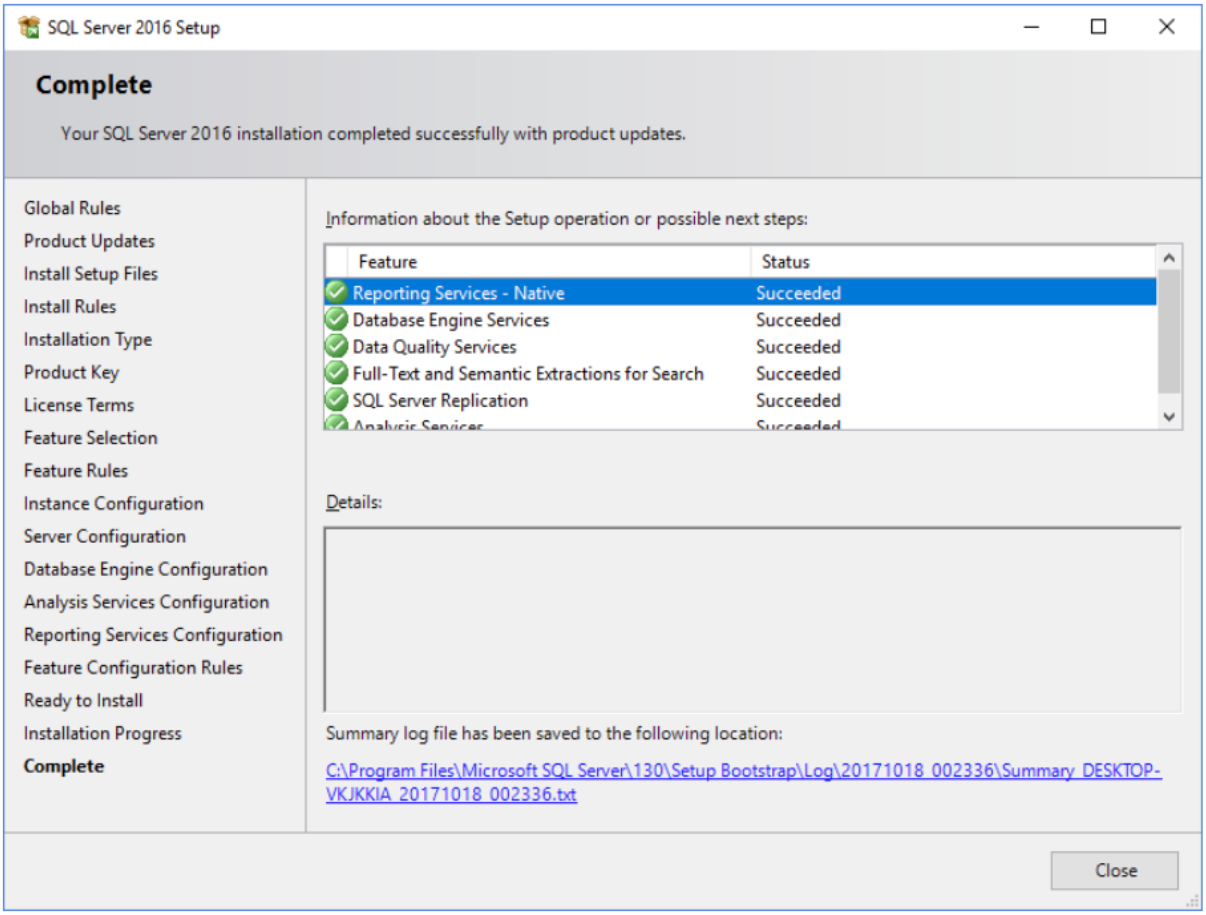
Figura 16 – Pastas de destino



Fonte: do autor, 2017

Após, foi realizada a instalação das *features* selecionadas anteriormente, que por fim, resultará em uma lista com os itens que foram instalados corretamente, de acordo com a Figura 17. Caso ocorra algum erro durante o processo de instalação, o mesmo ficará visível para que o administrador possa solucionar o problema, sendo necessário que a aplicação seja reiniciada percorrendo todo o fluxo novamente.

Figura 17 – Log de instalação



Fonte: do autor, 2017

A administração do banco de dados foi realizada através do *SQL Server Management Studio*, que se encontra na versão 17.3 e é utilizado para configurar, gerenciar e monitorar o banco de dados SQL Server.

Não será apresentado o passo-a-passo para a instalação pois não possui nenhuma configuração especifica para o seu correto funcionamento, contudo, pode ser realizado o *download* da aplicação diretamente no site da Microsoft (2017b).

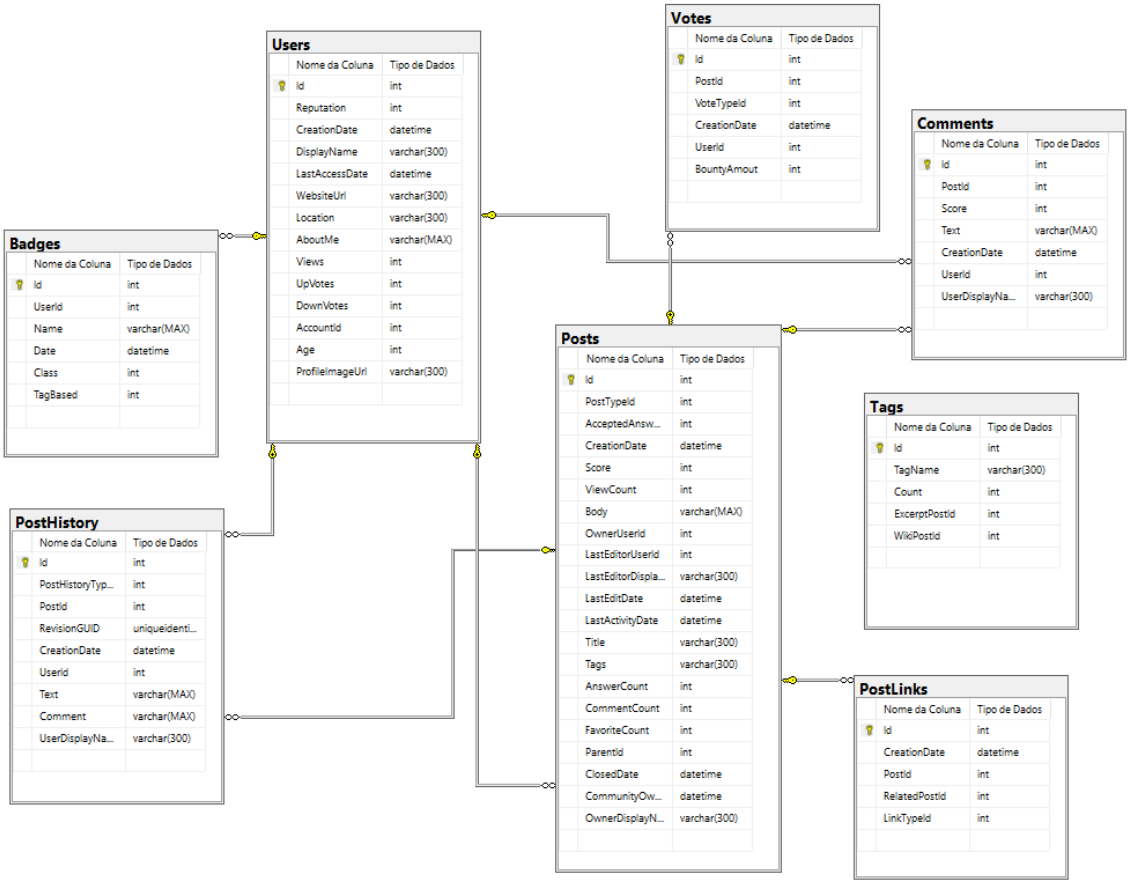
3.3 BASE DE DADOS

Para a obtenção de resultados íntegros na comparação dos diferentes métodos de armazenamento e entre os SGBDS, buscou-se a utilização da mesma base de dados utilizada no estudo realizado por Horn (2016). Com o objetivo de simular o processamento, houve a necessidade de que a base de dados possuísse grande quantidade de registros, com isso ocupando grande espaço em disco e/ou memória.

A base de dados utilizada foi obtida do estudo de Horn (2016). Com a mesma quantidade de registros para melhor equivalência dos resultados. A base atualizada, pode ser encontrada no StackOverflow (2017), 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. (HORN, 2016)

Conforme análise dos dados armazenados nos 8 arquivos e de fontes do próprio *StackOverflow* (2014), foi criado o esquema lógico do banco de dados, conforme Figura 18.

Figura 18 – Esquema Lógico Relacional



Fonte: do autor, 2017

Os índices criados inicialmente foram apenas os vinculados a *Primary Key* de cada tabela, que precisam ser declarados no momento de criação das tabelas com armazenamento em memória, a adição de novos índices será realizada nos experimentos do Capítulo 4.

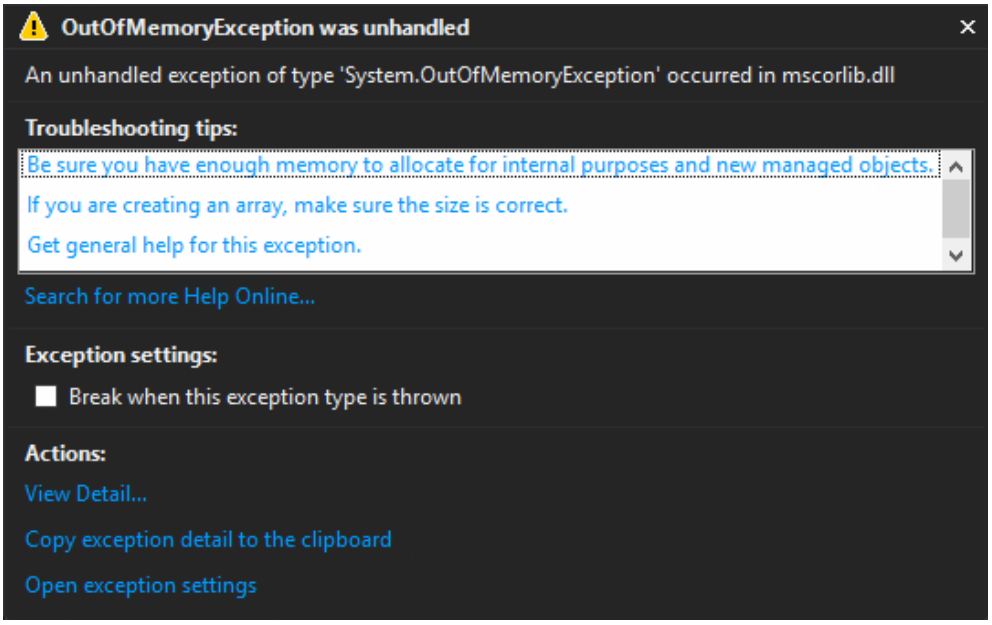
3.4 IMPORTAÇÃO DOS DADOS

Após a realização de análise da aplicação desenvolvida por Horn (2016), foi verificado que houve muitos problemas com o tratamento dos dados, devido ao tamanho dos arquivos. Para a realização do experimento deste trabalho, foram realizadas tentativas diferentes com o intuito de melhorar o desempenho da aplicação de divisão dos arquivos, o que poderia beneficiar o modelo gerado para a posterior inserção dos dados no SGBD.

Após verificar como o trabalho anterior tratava os dados de inserção, foi constatado que o método não atendia plenamente o grupo de dados presentes no XML. O arquivo era lido com limites de dados, sendo delimitado que a cada X linhas, geraria um novo XML. Porém, todas as linhas posteriores ao limite estavam sendo desconsideradas, não sendo inseridas no banco de dados. Como exemplo, caso o delimitador para quebra, estivesse definido como 10 linhas e a quantidade de linhas do arquivo fosse de 25, as últimas 5 linhas de registro seriam ignoradas pela aplicação, o que poderia causar inconsistência no vínculo entre as tabelas. Foram realizadas alterações para a correção do problema e de melhor aproveitamento do código.

Na primeira tentativa, foi utilizada a classe C# *ReadLines*, que é utilizada principalmente para a leitura de dados de arquivos. Esta foi efetiva para a leitura de pequenos arquivos XML, gerando erro de Excesso de Memória para arquivos com mais de 100 MB, conforme Figura 19.

Figura 19 – Estouro de memória



Fonte: do autor, 2017

A segunda tentativa foi mais efetiva, sendo utilizada a classe C# *XmlTextReader*, que permite uma rápida leitura de arquivos XML independente de seu tamanho. Com isso, foi obtido o número de linhas de cada arquivo XML, dividido pela quantidade de linhas limite previamente estipulado para cada arquivo XML principal.

A quantidade de linhas por XML foi definida após análise do tamanho do arquivo principal e do conjunto de dados presentes. Como exemplo, o arquivo XML da tabela *Comments* possui 12.5 GB de espaço, e tal tabela possui a coluna *Text*, que armazena o comentário de cada usuário em um *VARCHAR (MAX).* Com tais informações foi definido que a quebra ocorreria a cada 150 mil linhas, gerando arquivos entre 50 MB e 60 MB, tamanho máximo para que a aplicação não gerasse erro de exceção de memória.

Contudo, a tabela *Badges*, que possui 2.1 GB, foi separada em arquivos de 500 mil linhas, pois seu conteúdo é na maior parte de colunas numéricas *(INT)* e sem grandes informações na coluna *Name*. Assim, cada arquivo gerado ficou com cerca de 55 MB, não ultrapassando os 60 MB que resultariam no erro de falta de memória para o processamento.

Com a definição de qual método utilizar para a divisão do XML, foi aplicada a solução para os 8 arquivos principais, os quais geraram os dados do Quadro 5.

Quadro 5 – Volume de dados utilizado para realização de testes

Tabela	Linhas	Tamanho Arquivo	Ponto de Quebra	Arquivos Gerados	Tempo
Users	5.987.285	1.7 GB	200.000	30	00:02:34
Badges	18.395.121	2.1 GB	500.000	37	00:04:22
PostHistory	77.910.283	74.8 GB	80.000	974	01:29:30
Posts	32.209.817	47.7 GB	50.000	645	00:51:21
PostLinks	3.335.637	0.4 GB	100.000	34	00:00:42
Comments	48.998.958	12.5 GB	200.000	245	00:18:55
Tags	45.426	0.039 GB	30.000	2	00:00:01
Votes	105.301.745	10 GB	500.000	211	00:14:49
Total	**292.184.272**	**149.239 GB**	**-**	**2.178**	**03:02:14**

Fonte: do autor, 2017

No mesmo processo de divisão dos arquivos, foi criada a instrução de inserção no banco de dados, esta instrução foi gerada em arquivos no formato .BAT, para que após seja executado outro comando para que percorra todos os arquivos *batch* e os insira na base dados, com a estrutura de tabela conforme o XML apresentava.

Foi gerado um arquivo *batch* com a instrução de inserção para cada arquivo XML gerado. Foi utilizada a instrução de *OPENROWSET* para a leitura dos arquivos e após a leitura, a inserção dos dados em tabelas temporárias para a realização da transição dos dados para o seu destino final, conforme Figura 20.

Figura 20 – Arquivo Batch

```
INSERT INTO XmlPosts
SELECT 1,CONVERT(xml, BulkColumn, 2) as BulkColumn FROM
    OPENROWSET(Bulk 'C:\Users\0093362\Desktop\baseStack\stackoverflow.com-Tags\02_Tags_Insert_00Final.xml', SINGLE_BLOB)

declare @X1 xml = (select XmlCol from XmlPosts where linha = 1)

insert into Posts
SELECT
    [Id] = XTbl.TypeNode.value('@Id', 'Int')
    ,[PostTypeId] = XTbl.TypeNode.value('@PostTypeId', 'int')
    ,[AcceptedAnswerId] = XTbl.TypeNode.value('@AcceptedAnswerId', 'int')
    ,[CreationDate] = XTbl.TypeNode.value('@CreationDate', 'datetime')
    ,[Score] = XTbl.TypeNode.value('@Score', 'int')
    ,[ViewCount] = XTbl.TypeNode.value('@ViewCount', 'int')
    ,[Body] = XTbl.TypeNode.value('@Body', 'varchar(max)')
    ,[OwnerUserId] = XTbl.TypeNode.value('@OwnerUserId', 'int')
    ,[LastEditorUserId] = XTbl.TypeNode.value('@LastEditorUserId', 'int')
    ,[LastEditorDisplayName] = XTbl.TypeNode.value('@LastEditorDisplayName', 'varchar(300)')
    ,[LastEditDate] = XTbl.TypeNode.value('@LastEditDate', 'datetime')
    ,[LastActivityDate] = XTbl.TypeNode.value('@LastActivityDate', 'datetime')
    ,[Title] = XTbl.TypeNode.value('@Title', 'varchar(300)')
    ,[Tags] = XTbl.TypeNode.value('@Tags', 'varchar(300)')
    ,[AnswerCount] = XTbl.TypeNode.value('@AnswerCount', 'int')
    ,[CommentCount] = XTbl.TypeNode.value('@CommentCount', 'int')
    ,[FavoriteCount] = XTbl.TypeNode.value('@FavoriteCount', 'int')
    ,[CommunityOwnedDate] = XTbl.TypeNode.value('@CommunityOwnedDate', 'datetime')
FROM
    @X1.nodes('/row') AS XTbl(TypeNode)
```

Fonte: do autor, 2017

Com a execução de todos os arquivos .BAT, foram inseridos todos os registros de todas as tabelas no banco de dados em disco para a primeira carga de dados do estudo. Para a conclusão desta operação foram cerca de 28 horas de processamento contínuo.

Para a base de dados em memória, a maior parte das tabelas pode ser preenchida diretamente através de inserção em tabelas temporárias e posterior inserção na tabela destino. Nas tabelas maiores não foi possível realizar a migração de dados deste modo, pois acarretava em estouro de memória durante a fase de inserção dos dados, sendo realizada a execução de .BAT e inserção direta na tabela destino.

No Capítulo 5 serão abordados os experimentos realizados com ambas as bases de dados, demonstrando os *scripts* que foram criados para a realização dos experimentos e seus resultados através de gráficos representando o tempo decorrido para cada operação em ambas as bases de estudo.

4. EXPERIMENTOS

O objetivo do trabalho visou a obtenção de conhecimento com a utilização de banco de dados em memória para a comparação, através de testes práticos, entre banco de dados tradicional e banco de dados em memória. Foram criadas duas bases de dados, utilizando a base de dados apresentada no capítulo 3. Onde, uma das bases foi criada através do método tradicional e outra totalmente em memória.

Ambas as bases apresentam os mesmos registros e estrutura de tabelas, podendo sofrer pequenas mudanças nos *scripts* de criação das tabelas, se adequando a cada caso e não influenciando no resultado final. Para os testes práticos, tanto tradicional quanto em memória, foi utilizada a ferramenta Microsoft SQL Server Management Studio 17, que é uma interface para criação de instruções do banco de dados.

Os experimentos foram executados 5 vezes cada instrução, em ambas as bases de dados. Com o objetivo de não ter influência de qualquer dado previamente consultado, cada execução foi feita individualmente. Após a coleta do tempo foi executado o comando *DROPCLEANBUFFERS*, que elimina qualquer dado que possa ter ficado na memória da consulta, além disso, o tempo foi obtido em milissegundos através do comando *SET STATISTICS TIME ON*.

4.1 EXPERIMENTO I – OPERAÇÕES BÁSICAS

Estes primeiros experimentos têm como objetivo a execução e comparação de desempenho com operações básicas. Foram realizados os mesmos experimentos em ambas as bases de dados, inserção, alteração, leitura e exclusão.

Cada instrução foi executada de forma individual, para evitar conflitos que causassem divergências inesperadas para o estudo. Espera-se que o resultado obtido com este experimento comprove a teoria de que operações em memória sempre terão melhor desempenho se comparadas com o armazenamento em disco.

Para a realização dos testes foi utilizado como base a questão de número 2334712, conforme a Figura 19, que é a pergunta com mais interação entre os usuários da categoria SQL Server. Deste, foi utilizado o próprio identificador e também o identificador de seu criador, que pode ser encontrado no próprio fórum, conforme a Figura 21.

Figura 21 – Exemplo de *post* utilizado no estudo

Fonte: do autor, 2017

Com a definição do *post* a ser utilizado, os experimentos que utilizam *WHERE* se basearam no número deste *post*, no identificador do usuário que criou o *post* e no conteúdo descrito no questionamento do usuário.

4.1.1 Consulta com filtro pela *Primary Key*

Uma das operações mais utilizadas do banco de dados, que consiste na obtenção de dados diversos sobre registros específicos. Para a realização deste teste, foram seguidos os seguintes passos:

1) Elaborar o comando *SELECT* para a operação de leitura nas tabelas para os cenários propostos, buscando um registro específico na tabela *Posts* (Figura 20);
2) Executar o comando;
3) Registrar os tempos de resposta;
4) Elaborar o comando *SELECT* para a operação de leitura nas tabelas para os cenários propostos, buscando todos os registros da tabela *Posts* sem *WHERE* (Figura 23);
5) Executar o comando;
6) Registrar os tempos de resposta;

7) Elaborar o comando *SELECT* para a operação de leitura nas tabelas para cenários propostos, buscando todos os registros da tabela *Posts* com *WHERE* que retorne todos os registros (Figura 23);
8) Executar o comando;
9) Registrar os tempos de resposta;

Com a execução da *query* da Figura 22 em ambas as bases de dados, foi constatado que o tempo para o retorno de um *post* especifico é muito baixo, por ser utilizado o índice da *Primary Key*, a diferença fica menor do que 1 centésimo de segundo, conforme exibido na Figura 23.

Figura 22 – Consulta por *post* especifico

```
DBCC DROPCLEANBUFFERS
SET STATISTICS TIME ON

select * from Posts where Id = 2334712
```

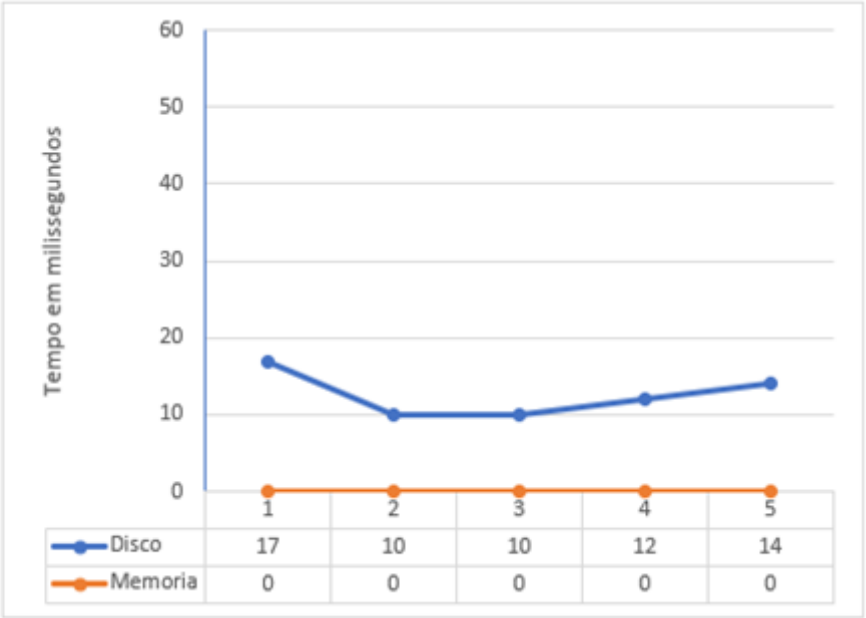
Fonte: do autor, 2017

Figura 23 – Comparação consulta por *post* especifico

Disco	Memória
00:00:00.017	00:00:00.000
00:00:00.010	00:00:00.000
00:00:00.010	00:00:00.000
00:00:00.012	00:00:00.000
00:00:00.014	00:00:00.000

Média	
00:00:00.013	00:00:00.000

Diferença
1300%



Fonte: do autor, 2017

Apesar de ambos terem o retorno muito rápido, não se pode desconsiderar o fato de que a execução em memória teve seu retorno instantâneo, sendo que se considerarmos o valor mínimo de 1 milissegundo para o processamento em memória, ainda assim será 13 vezes mais eficiente se comparado ao processo em disco.

O segundo experimento realizado foi uma consulta simples na tabela *Posts*, uma sem *WHERE* e outra com *WHERE* buscando todos os identificadores maiores que zero, fazendo com que ambas consultas retornassem todos os registros da tabela em questão, de acordo com a Figura 24.

Figura 24 – *Script* com e sem restrição

```
DBCC DROPCLEANBUFFERS
SET STATISTICS TIME ON

Select * from Posts
Select * from Posts where Id > 0
```

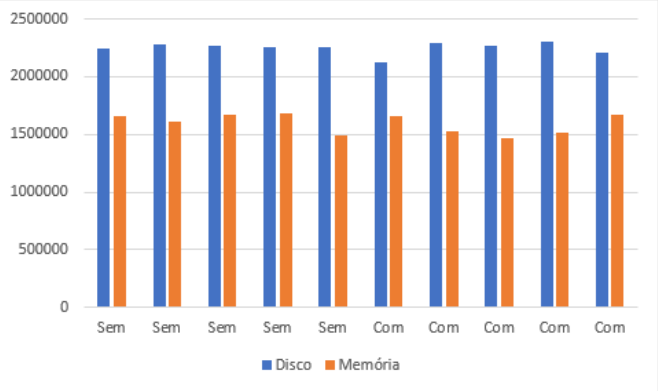
Fonte: do autor, 2017

Os resultados exibidos na Figura 25, demonstram que a diferença e a variação existem, contudo, ambas têm tempo de retorno muito semelhante, não sendo possível concluir que uma é melhor que outra.

Figura 25 – Comparação resultados sem e com restrição

Sem WHERE		Com WHERE	
Disco	Memória	Disco	Memória
00:37:26:820	00:27:38:013	00:35:21:829	00:27:41:487
00:38:00:287	00:26:48:553	00:38:10:460	00:25:21:603
00:37:52:700	00:27:50:197	00:37:52:403	00:24:28:473
00:37:31:482	00:28:06:877	00:38:26:830	00:25:19:480
00:37:36:684	00:24:45:473	00:36:49:763	00:27:48:727
Média		Média	
00:37:41:597	00:27:01:823	00:37:20:257	00:26:07:953

Diferença	Diferença
39%	43%



Fonte: do autor, 2017

As *queries* deste experimento se utilizam do *Full Table Scan*, o que gera um grande custo de processamento para o banco de dados, retornando todos os 32.209.817 registros da tabela *Posts*.

4.1.2 Consulta com filtro pela *Foreign Key*

Operação que consiste na filtragem de dados que referenciam outra tabela através de ligação denominada chave estrangeira. Para a realização deste teste, foram seguidos os seguintes passos:

1) Elaborar o comando *SELECT* para a operação de leitura nas tabelas para os dois cenários propostos, buscando todos os *posts* de determinado usuário;
2) Executar o comando;
3) Registrar os tempos de resposta;

Com a busca na tabela *Posts* para todos os *posts* de determinado usuário, utilizando a consulta da Figura 26, a diferença se torna visivelmente desproporcional, chegando a uma diferença de quase 70 vezes menos tempo na consulta em memória.

A coluna utilizada para o filtro não possui índice, fazendo com que ocorra a busca em toda a tabela, retornando o conjunto de 45 linhas de registros de *posts* do usuário especificado.

Figura 26 – *Script* consulta post especifico

```
DBCC DROPCLEANBUFFERS
SET STATISTICS TIME ON

Select * from Posts where OwnerUserId = 150939
```

Fonte: do autor, 2017

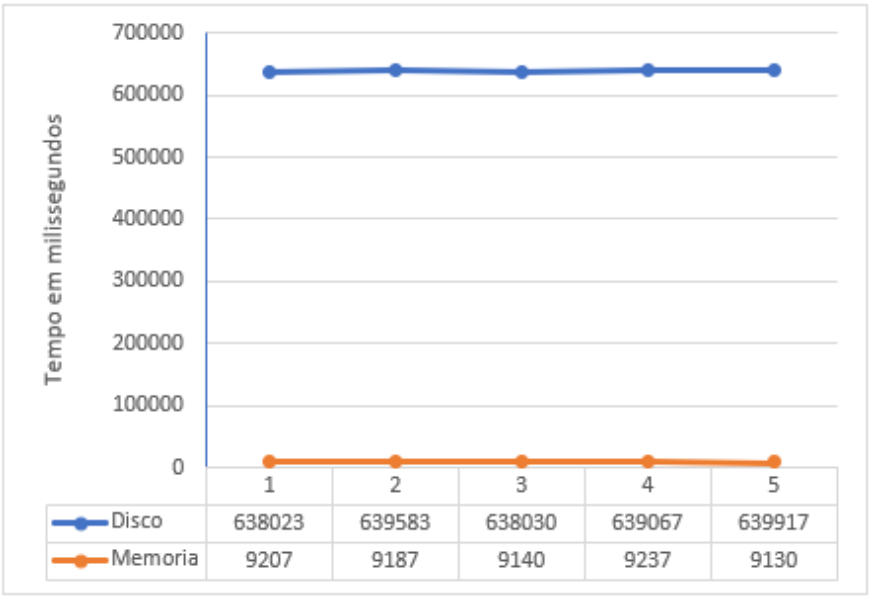
Pode-se observar na Figura 27 que há pouca oscilação nos tempos gerados em cada base de dados, concluindo-se que com mais execuções da mesma operação, a diferença permaneceria grande.

Figura 27 – Comparação consulta todos *posts* de um usuário

Disco	Memória
00:10:38.023	00:00:09.207
00:10:39.583	00:00:09.187
00:10:38.030	00:00:09.140
00:10:39.067	00:00:09.237
00:10:39.917	00:00:09.130

Média	
00:10:38.923	00:00:09.180

Diferença
6860%



Fonte: do autor, 2017

4.1.3 Consulta com *Like*

Operação que aplica filtro em determinada coluna de uma tabela, procurando todos os registros que contenham determinado valor. Para a realização deste teste, foram seguidos os seguintes passos:

1) Elaborar o comando *SELECT* para a operação de leitura nas tabelas para os dois cenários propostos, utilizando a operação *LIKE* na tabela *Comments*;
2) Executar o comando;
3) Registrar os tempos de resposta;

Com a aplicação da cláusula *LIKE* na tabela *Comments*, conforme Figura 28, houve maior variação dos tempos de processamento, fazendo com que retornassem 1.008.654 registros.

Figura 28 – Consulta com cláusula *LIKE*

```
DBCC DROPCLEANBUFFERS
SET STATISTICS TIME ON

select * from Comments where Text Like '%SQL%'
```

Fonte: do autor, 2017

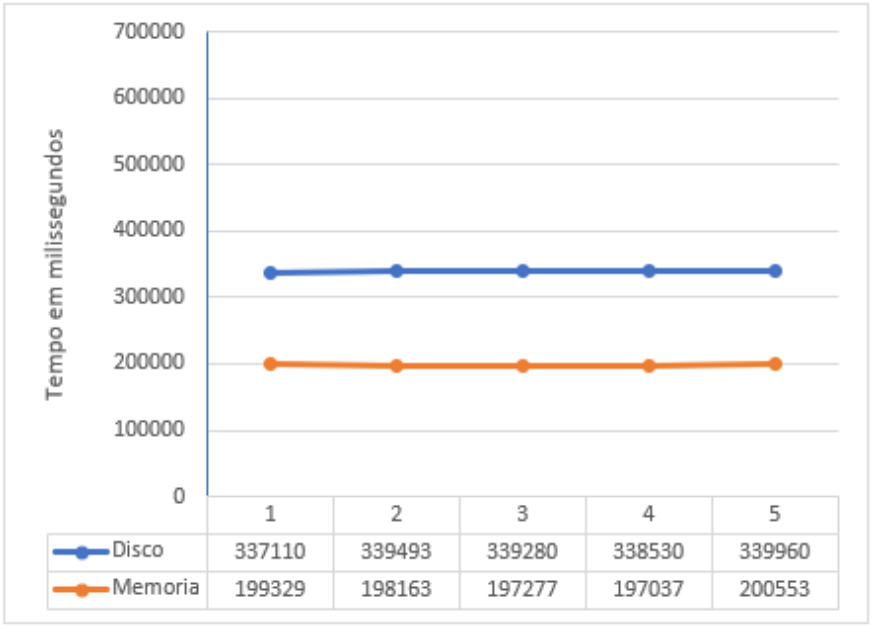
Pode-se observar que houve grande custo por parte da operação em disco, chegando a quantia de 71% mais do que a mesma operação em memória, conforme disponibilizado na Figura 29.

Figura 29 – Comparação consulta com cláusula *LIKE*

Disco	Memória
00:05:37:110	00:03:19:329
00:05:39:493	00:03:18:163
00:05:39:280	00:03:17:277
00:05:38:530	00:03:17:037
00:05:39:960	00:03:20:553

Média	
00:05:38:877	00:03:18:473

Diferença
71%



Fonte: do autor, 2017

4.1.4 Consulta com contador e agrupamento

Além das operações simples, foi realizada a comparação dos tempos de consultas com contador e agrupadas em um identificador.

1) Elaborar o comando *SELECT* para a operação de leitura nas tabelas para os dois cenários propostos, tendo o contador *COUNT* na tabela *Posts* e agrupados por usuário.
2) Executar a instrução em ambos ambientes;
3) Registrar os tempos de resposta;

Com a execução da consulta da Figura 30, foi obtida a quantidade de 2.936.843 registros distintos de usuários.

Figura 30 – Script consulta agrupada

```
DBCC DROPCLEANBUFFERS
SET STATISTICS TIME ON

Select count(ownerUserId) from Posts
group by ownerUserId
```

Fonte: do autor, 2017

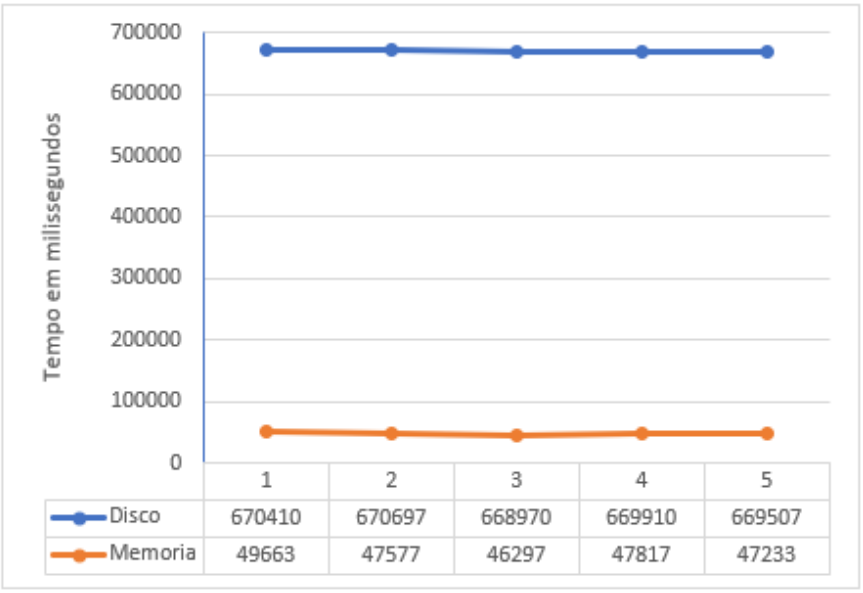
Os resultados apresentados na Figura 31 mostram que houve diferença de cerca de quatorze vezes menos tempo de processamento para a instrução na base de dados em memória, um ganho considerado grande para uma instrução simples.

Figura 31 – Comparação de consulta com contador e agrupamento

Disco	Memória
00:11:10.410	00:00:49.663
00:11:10.697	00:00:47.577
00:11:08.970	00:00:46.297
00:11:09.910	00:00:47.817
00:11:09.507	00:00:47.233

Média	
00:11:09.900	00:00:47.717

Diferença
1304%



Fonte: do autor, 2017

4.1.5 Operação de inserção

Para a validação do tempo de inserção, serão inseridos 1 milhão de registros reais em cada uma das bases do estudo, na tabela *Comments*. Para realizar este procedimento, as seguintes etapas serão seguidas:

1) Selecionar 1 milhão de registros;
2) Inserir os registros em uma tabela temporária;

3) Executar a operação de *DELETE* nos dados da tabela principal que foram transferidos para a tabela temporária
4) Executar a operação de *INSERT* em ambos ambientes;
5) Registrar os tempos de resposta;

Para a inserção dos dados foi executado o s*cript* apresentado na Figura 32, em ambas as bases de estudo. Embora a quantidade de dados seja significante, a diferença ficou na média de 2 segundos entre as operações, em algumas execuções da instrução chegou a ficar com diferença na casa dos centésimos de segundo.

Figura 32 – Processo de inserção de 1 milhão de registros

```
DBCC DROPCLEANBUFFERS
SET STATISTICS TIME ON

select * into ##Comments2 from Comments where id >= 61767110

delete Comments where id >= 61767110

insert into Comments
select * from ##Comments2
```

Fonte: do autor, 2017

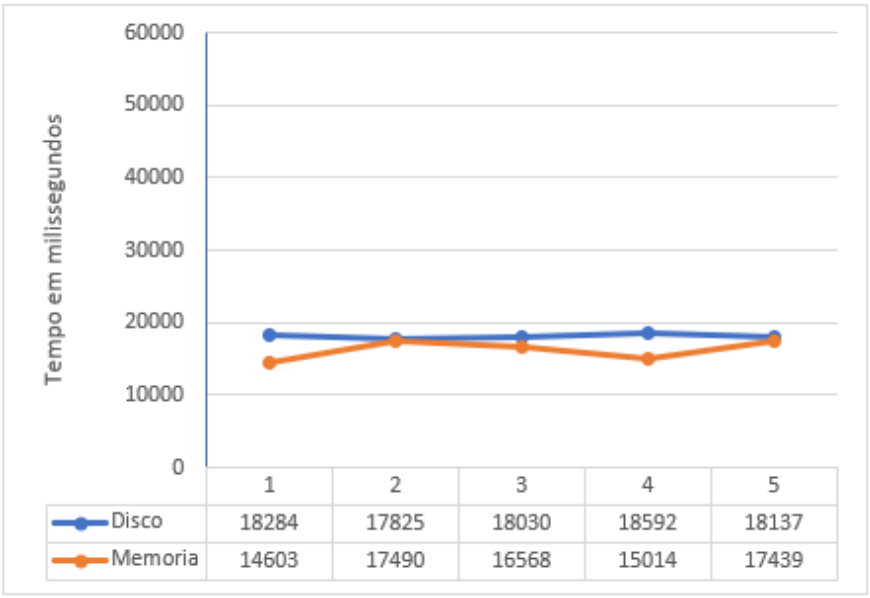
Na Figura 33 pode-se observar que houve maiores oscilações no tempo gerado na base de dados em memória em comparação com as operações em disco, podendo ter como causa instabilidade do servidor.

Figura 33 – Comparação instrução de *Insert*

Disco	Memória
00:00:18:284	00:00:14:603
00:00:17:825	00:00:17:490
00:00:18:030	00:00:16:568
00:00:18:592	00:00:15:014
00:00:18:137	00:00:17:439

Média	
00:00:18:173	00:00:16:223

Diferença
12%



Fonte: do autor, 2017

4.1.6 Operação de exclusão

Para as operações de exclusão, serão removidos 1 milhão de registros inseridos anteriormente e todos os comentários de um determinado usuário, ambos da tabela *Comments* e realizados em cada base do estudo. Para este procedimento, as seguintes etapas serão seguidas:

1) Limitar a exclusão para os dados inseridos no processo anterior;
2) Executar a operação de *DELETE* dos dados em ambos ambientes;
3) Registrar os tempos de resposta;
4) Limitar a exclusão para um determinado usuário;
5) Executar a operação de *DELETE* dos dados em ambos ambientes;
6) Registrar os tempos de resposta;

Na exclusão dos resultados, foi observada grande diferença impactada pela estrutura da tabela, onde se tem a *Primary Key* no identificador do comentário, gerando melhor processamento da operação de delimitação dos dados e remoção. A Figura 34 apresenta os *scripts* utilizados para este experimento.

Figura 34 – Instruções para testes de exclusão

```
DBCC DROPCLEANBUFFERS
SET STATISTICS TIME ON

delete Comments where Id >= 61767110

delete Comments where UserId = 150939
```

Fonte: do autor, 2017

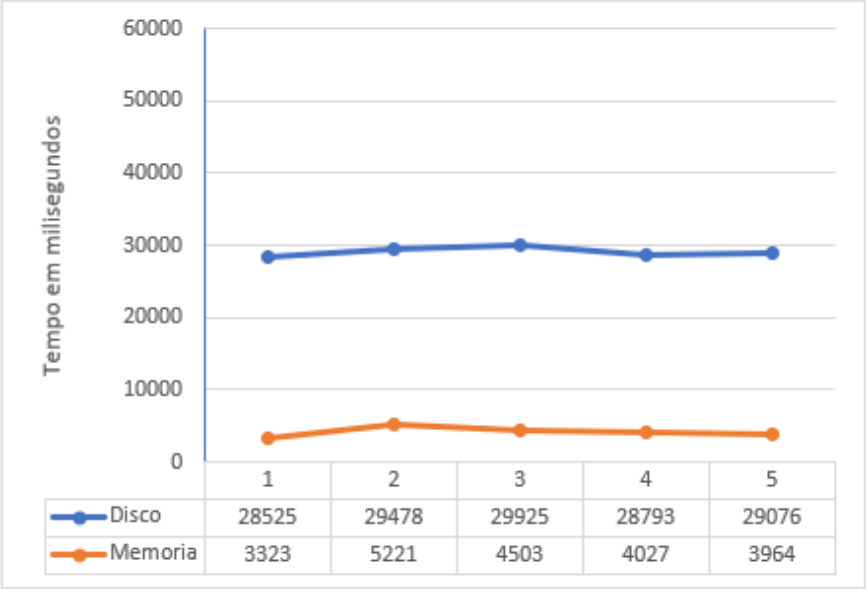
Em conformidade com a Figura 35, para 1 milhão de registros, a média de tempo em disco foi de 29 segundos, enquanto para a remoção de 65 registros encontrado pelo código do proprietário, foram gastos 5 minutos e 28 segundos, conforme Figura 36. Contudo, a média de tempo para as operações em memória, não tiveram oscilação tão grande entre as duas operações.

Figura 35 – Comparação da remoção de 1 milhão de registros

Disco	Memória
00:00:28:525	00:00:03:323
00:00:29:478	00:00:05:221
00:00:29:925	00:00:04:503
00:00:28:793	00:00:04:027
00:00:29:076	00:00:03:964

Média	
00:00:29:159	00:00:04:208

Diferença
593%



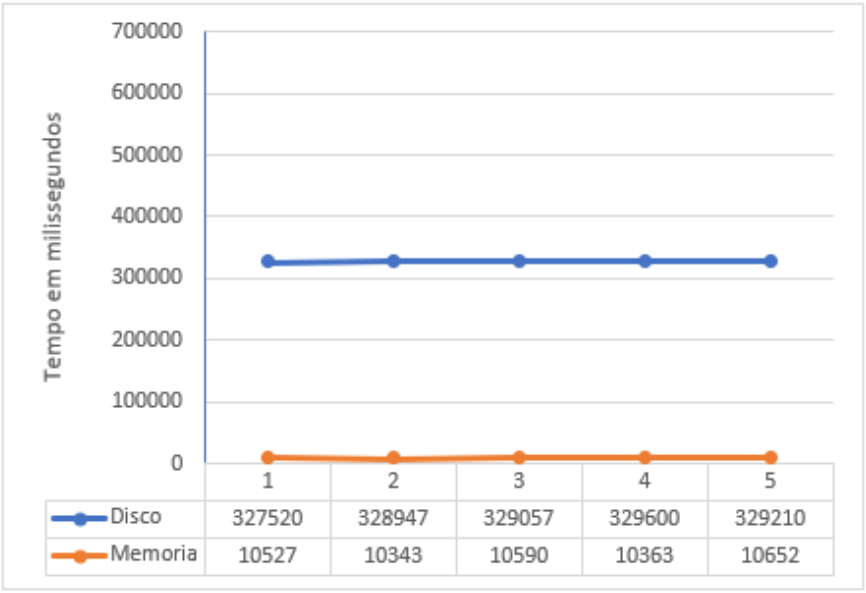
Fonte: do autor, 2017

Figura 36 – Remoção de todos os comentários de determinado usuário

Disco	Memória
00:05:27:520	00:00:10:527
00:05:28:947	00:00:10:343
00:05:29:057	00:00:10:590
00:05:29:600	00:00:10:363
00:05:29:210	00:00:10:652

Média	
00:05:28:867	00:00:10:495

Diferença
3034%



Fonte: do autor, 2017

4.1.7 Operação de alteração

Para a validação do tempo de resposta da alteração de dados, serão realizadas alterações dos dados da tabela *Posts*, em ambas as bases de dados, conforme as seguintes etapas:

1) Elaborar os comandos de *UPDATE* para alteração de dados;
2) Executar os comandos;
3) Registrar os tempos de resposta;

Com a execução da instrução disposta na Figura 37, foi verificada grande diferença de tempo para o processamento em disco e em memória. A instrução realizou alteração em 45 registros que pertencem ao usuário em estudo, resultando nos tempos da Figura 38.

Figura 37 – Instrução de alteração de dados

```
DBCC DROPCLEANBUFFERS
SET STATISTICS TIME ON

update posts set Body = 'Teste' where OwnerUserId = 150939
```

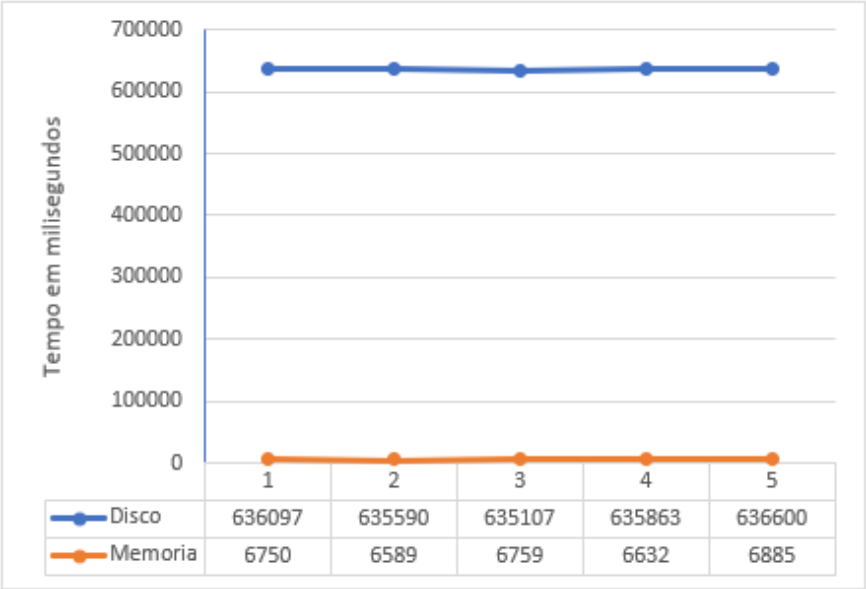
Fonte: do autor, 2017

Figura 38 – Comparação da alteração de dados

Disco	Memória
00:10:36:097	00:00:06:750
00:10:35:590	00:00:06:589
00:10:35:107	00:00:06:759
00:10:35:863	00:00:06:632
00:10:36:600	00:00:06:885

Média	
00:10:35:850	00:00:06:723

Diferença
9358%



Fonte: do autor, 2017

Com o resultado, pode-se observar que a diferença na média do tempo de processamento ficou 94 vezes maior no processamento em disco.

4.2 EXPERIMENTO II – OPERAÇÕES COM *JOIN*

O próximo experimento procura representar situações em que um *Select* faz acesso a 2 tabelas. Foram executados JOINS entre tabelas da mesma base de dados. Todas as operações foram executadas 5 vezes a fim de coletar os tempos para a realização e análise.

Um ponto interessante a ser destacado é que não é possível a realização de consultas entre tabelas armazenadas em memória e tabelas armazenadas em disco, ao tentar realizar este tipo de ligação é apresentado o erro da Figura 39.

Figura 39 – Erro ao unir tabelas de ambas as bases

```
Select * from dbMemoria..Posts P
inner join dbDisco..Users U
    on U.Id = P.OwnerUserId
where P.Id = 2334712

/***    ERRO    ***/
Msg 41317, Level 16, State 3, Line 1
A user transaction that accesses memory optimized tables or natively compiled modules cannot access
more than one user database or databases model and msdb, and it cannot write to master.
```

Fonte: do autor, 2017

4.2.1 Consulta com *JOIN*

Para a realização dos testes, foram utilizadas as principais tabelas do estudo, realizando vínculos tanto com *INNER JOIN* como com *OUTER JOIN*. Para a realização destes testes, foram seguidos os seguintes passos:

1) Elaborar o comando *SELECT* com vínculo *INNER JOIN* entre as tabelas *Posts* e *Users*;
2) Executar o comando, sem delimitadores, retornando todos os registros;
3) Registrar os tempos de resposta;
4) Elaborar o comando *SELECT* com vínculo *INNER JOIN* entre as tabelas *Posts* e *PostHistory*;
5) Executar o comando com o delimitador de ID de *post* especifico;
6) Registrar os tempos de resposta;
7) Executar o mesmo comando com o delimitador *OwnerUserId* retornando todos os registros um usuário específico;
8) Registrar os tempos de resposta;

Com o *script* apresentado na Figura 40 utilizando *join* entre as tabelas *Posts* e *Users*, foi obtido retorno de mais de 31 milhões de registros, fazendo com que o resultado levasse quase uma hora para ser alcançado. Foi percebido que a quantidade de registros não se encaixa perfeitamente com a relação de registros na tabela *Posts*, nem todos os *posts* possuem registro de identificador do proprietário, não sendo possível saber o motivo, mas sabe-se que estão de acordo com a base de dados original.

Figura 40 – Script *join* entre tabelas

```
DBCC DROPCLEANBUFFERS
SET STATISTICS TIME ON

Select * from Posts P
INNER JOIN Users U
    on U.id = P.OwnerUserId
```

Fonte: do autor, 2017

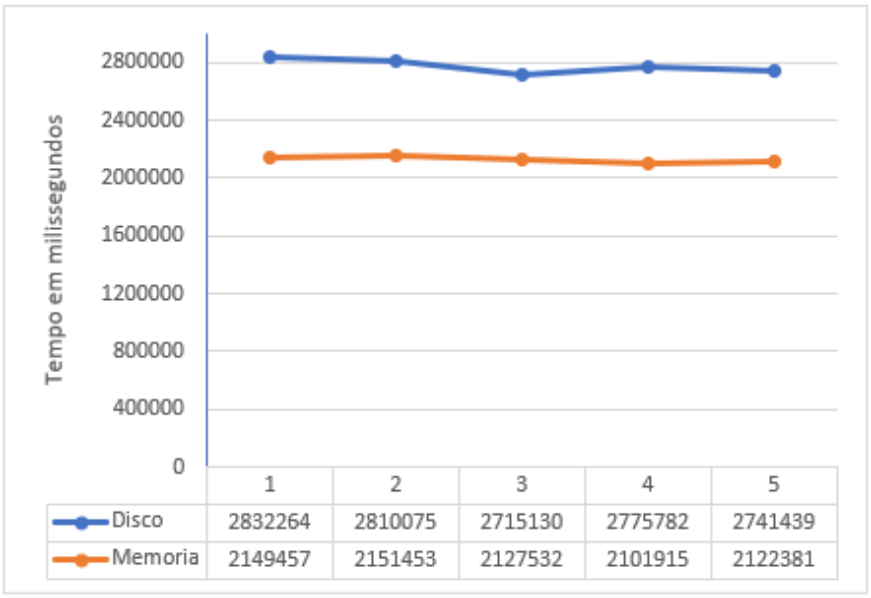
Na comparação entre as duas bases de dados, é possível observar que houve melhora no tempo de processamento em cerca de 30%, sendo a base em memória mais efetiva, conforme Figura 41.

Figura 41 – Comparação de *JOIN* entre *Posts* e *Users*

Disco	Memória
00:47:12:264	00:35:49:457
00:45:50:075	00:35:51:453
00:45:15:130	00:35:27:532
00:46:15:782	00:35:01:915
00:45:41:439	00:35:22:381

Média	
00:46:14:937	00:35:30:547

Diferença
30%



Fonte: do autor, 2017

Com os experimentos utilizando *INNER JOIN* entre as tabelas *Posts* e *PostHistory*, utilizando o *Script* da Figura 42, foi possível observar grande vantagem entre as operações em ambas as bases de dados, onde o custo na operação de busca por um único *post* foi cerca de 5 vezes maior em disco em relação a memória, retornando 15 registros, de acordo com a Figura 43.

Figura 42 – *Script join* entre tabelas por *post* especifico

```
DBCC DROPCLEANBUFFERS
SET STATISTICS TIME ON

Select * from Posts P
INNER JOIN PostHistory PH
    on PH.PostId = P.Id
Where P.Id = 2334712
```

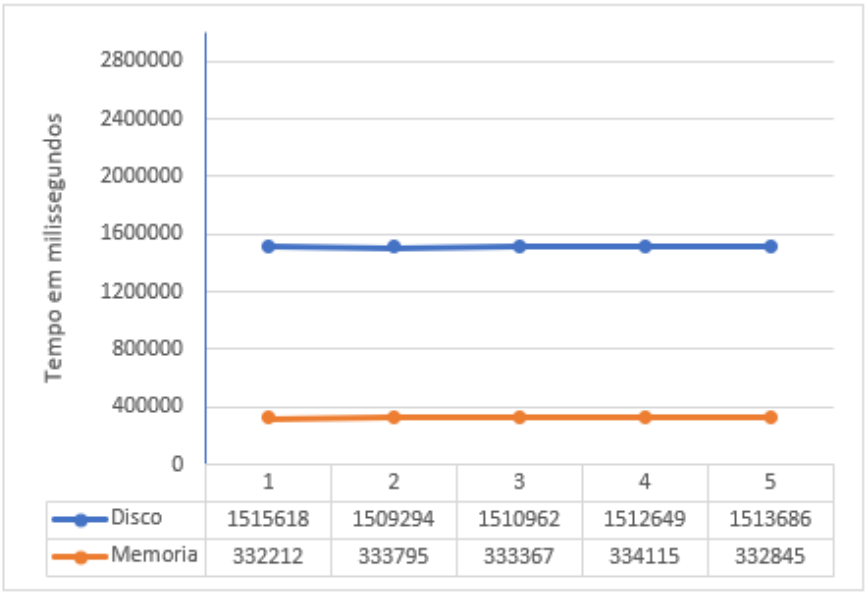
Fonte: do autor, 2017

Figura 43 – Comparação de *JOIN* entre *Posts* e *PostHistory* buscando por post

Disco	Memória
00:25:15:618	00:05:32:212
00:25:09:294	00:05:33:795
00:25:10:962	00:05:33:367
00:25:12:649	00:05:34:115
00:25:13:686	00:05:32:845

Média	
00:25:12:443	00:05:33:267

Diferença
354%



Fonte: do autor, 2017

Para a consulta de todos os registros de *posts* de um determinado usuário, foi utilizado o *script* apresentado na Figura 44, o mesmo obteve como retorno 160 registros, ficando em torno de 4 vezes mais rápido o processamento em memória, de acordo com a Figura 45.

Figura 44 – *Script join* entre tabelas por usuário

```
DBCC DROPCLEANBUFFERS
SET STATISTICS TIME ON

Select * from Posts P
INNER JOIN PostHistory PH
    on PH.PostId = P.Id
Where P.OwnerUserId = 150939
```

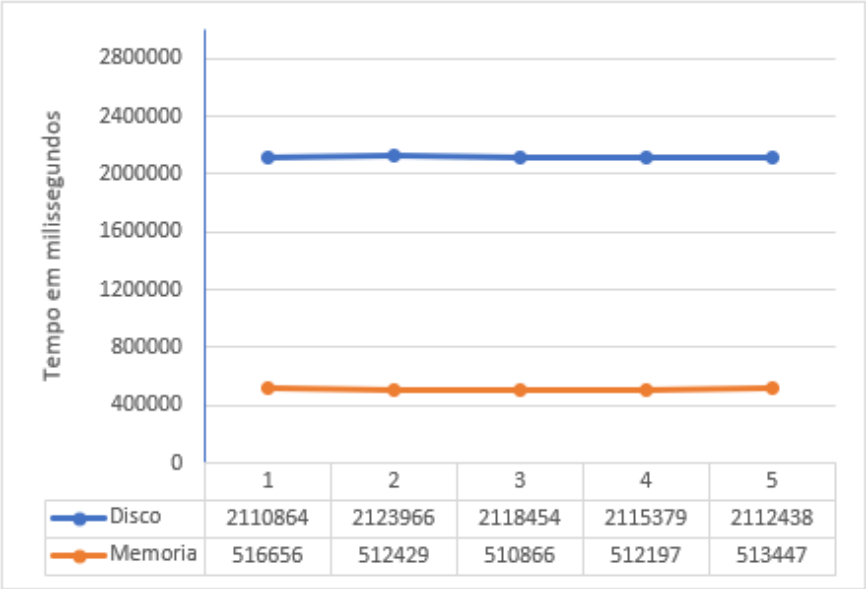
Fonte: do autor, 2017

Figura 45 – Comparação de *JOIN* entre *Posts* e *PostHistory* buscando por usuário

Disco	Memória
00:35:10:864	00:08:36:656
00:35:23:966	00:08:32:429
00:35:18:454	00:08:30:866
00:35:15:379	00:08:32:197
00:35:12:438	00:08:33:447

Média	
00:35:16:220	00:08:33:120

Diferença
312%



Fonte: do autor, 2017

4.3 EXPERIMENTO III – OPERAÇÕES COM *HASH INDEX*

Uma das possíveis formas de realizar melhorias no desempenho, que foi identificada durante a pesquisa bibliográfica, é a utilização de *Hash Index*. Que conforme o item 1.3, é capaz de encontrar registros únicos mais rápido do que qualquer outro tipo de índice do SQL Server, e tem seu melhor desempenho se combinado com a utilização de *Bucket* Count.

Conforme a Figura 46, que foi criada para exemplificar onde são definidas as posições de *Hash* e *Bucket Count*, ambas podem ser encontradas no final do *script* e pertencem a coluna Id, que é a *Primary Key* da tabela.

Figura 46 – Criação de tabela com Bucket e Hash Index

```
CREATE TABLE [dbo].[Posts](
    [Id] [int] NOT NULL PRIMARY KEY NONCLUSTERED,
    [PostTypeId] [int] NULL,
    [AcceptedAnswerId] [int] NULL,
    [CreationDate] [datetime] NULL,
    [Score] [int] NULL,
    [ViewCount] [int] NULL,
    [Body] [varchar](max) NULL,
    [OwnerUserId] [int] NULL,
    [LastEditorUserId] [int] NULL,
    [LastEditorDisplayName] [varchar](300) NULL,
    [LastEditDate] [datetime] NULL,
    [LastActivityDate] [datetime] NULL,
    [Title] [varchar](300) NULL,
    [Tags] [varchar](300) NULL,
    [AnswerCount] [int] NULL,
    [CommentCount] [int] NULL,
    [FavoriteCount] [int] NULL,
    [ParentId] [int] NULL,
    [ClosedDate] [datetime] NULL,
    [CommunityOwnedDate] [datetime] NULL,
    [OwnerDisplayName] [varchar](300) NULL
    index Id_Posts
    hash ([Id])
    WITH(BUCKET_COUNT = XXXXXXXXX)
) WITH( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA)
```

Fonte: do autor, 2017

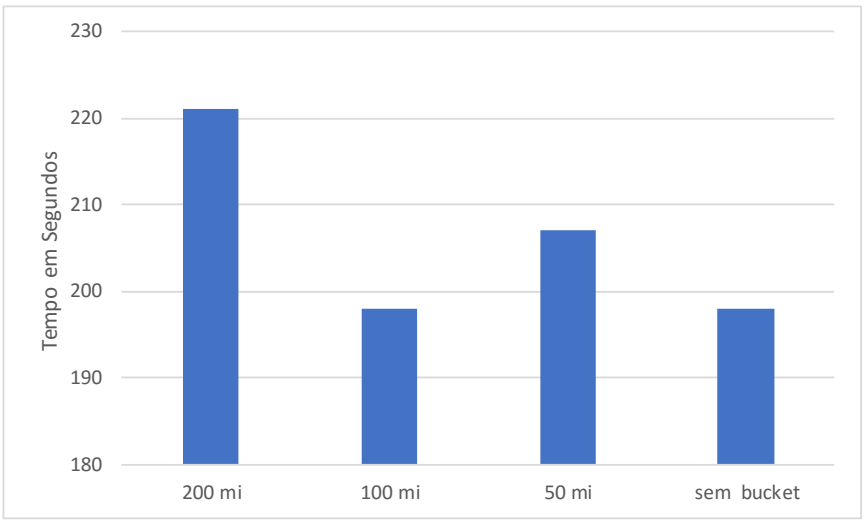
No mesmo *script* de criação da tabela, pode-se definir um número para o *Bucket Count*, no qual o número de *Bucket* deve ficar entre 1 e 10 vezes a quantia de registros distintos da coluna a ser aplicado, contudo, tem melhores resultados se a quantidade ficar entre 1.5 e 2 vezes.

4.3.1 Consulta com *Like*

Para a obtenção de resultados e comparação com consultas sem *hash index*, foi utilizado o mesmo *script* do experimento 4.1.3, onde buscou-se todos os registros que continham a palavra SQL na coluna *text* da tabela *Comments*, no qual, o experimento não obteve melhoras em seu tempo de execução, de acordo com a Figura 47.

Figura 47 – Desempenho consulta *Hash* com *Like*

Qtd Bucket	Segundos
200 mi	221
100 mi	198
50 mi	207
sem bucket	198



Fonte: do autor, 2017

Como se trata de operação com cláusula *LIKE* em uma coluna sem índice, não foi obtida nenhuma melhora em relação a tabela original. A tabela possui pouco mais de 48 milhões de registros distintos para a coluna ID, foram realizados experimentos para os *buckets* de 50, 100 e 200 milhões. Pode-se observar que a melhor média de tempo foi com o dobro de *buckets* em relação ao número de registros total da tabela, e o custo de memória pode ser observado no Quadro 6.

Quadro 6 – Custo em memória com o uso de *Buckets* na tabela *Posts*

Buckets	Memória alocada	Memória utilizada	Aumento com bucket
0	36007296	35273057	X
50 mi	36250432	35524696	0,675%
100 mi	36259648	35524696	0,701%
200 mi	36260736	35524697	0,704%

Fonte: do autor, 2017

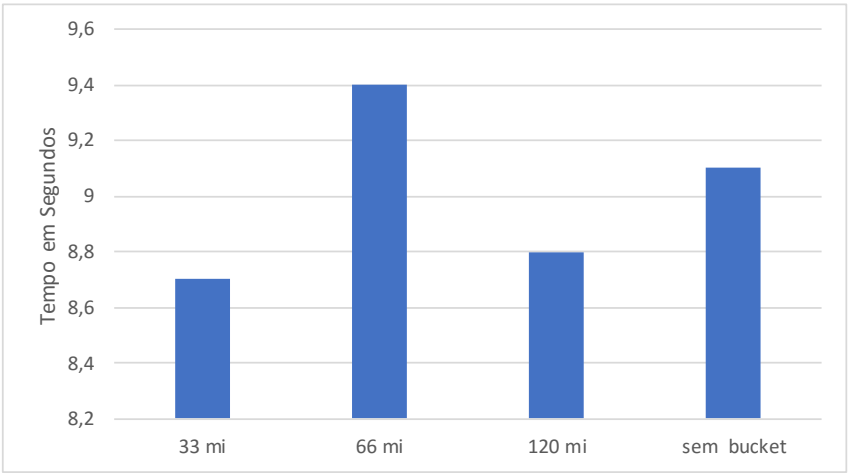
É possível verificar no Quadro 6 que a quantidade de memória aumenta com a adição de *buckets*, contudo, não chega a 1% de acréscimo em relação ao total de memória alocada para a tabela, concluindo-se que a adição no custo de memória é muito baixa em relação a todos os benefícios no tempo de processamento que serão obtidos.

4.3.2 Consulta com filtro pela *Foreign Key*

Realizando o filtro pela FK da tabela *Posts* que possui vínculo com a tabela *Users*, utilizando a mesma consulta do experimento 4.1.2, foi possível obter melhores resultados em comparação com a consulta sem *hash index*. Porém, conforme a Figura 48, o ganho foi de cerca de meio segundo utilizando-se a quantidade de 33 milhões de *buckets*, valor ligeiramente maior do que o total de registros, que é de mais de 29 milhões.

Figura 48 – Desempenho consulta Hash por usuário

Qtd Bucket	Segundos
33 mi	8,7
66 mi	9,4
120 mi	8,8
sem bucket	9,1



Fonte: do autor, 2017

A diferença de meio segundo apresentada anteriormente, equivale a pouco mais de 5% de melhora em tempo de processamento, o que pode ser considerado bom já que a coluna não possui nenhum tratamento de indexação no modelo do banco de dados original.

Foi criada uma segunda versão da mesma tabela e inseridos os mesmos dados da tabela original, contudo, a nova tabela foi criada com *Hash* na coluna *OwnerUserId* a fim de verificar a melhora que pode ser conquistada com uma simples alteração.

Para a criação da nova tabela, foi realizada a operação de consulta na tabela original a fim de obter a quantidade de valores distintos na coluna *OwnerUserId*, pois como a coluna não é o identificador principal da tabela sabe-se que muitos dos valores seriam repetidos, já que um único usuário pode ter inúmeros posts vinculados.

Com a consulta mencionada anteriormente, obteve-se a quantidade de 2.9 milhões de registros distintos, assim sendo, a nova tabela foi configurada com a quantidade de 3 milhões

de *bucket count* para o *hash index* da coluna *OwnerUserId*, o *script* de criação da tabela pode ser observado na Figura 49.

Figura 49 – *Script* criação tabela com *Bucket*

```
CREATE TABLE [dbo].[Posts3](
    [Id] [int] NOT NULL PRIMARY KEY NONCLUSTERED,
    [PostTypeId] [int] NULL,
    [AcceptedAnswerId] [int] NULL,
    [CreationDate] [datetime] NULL,
    [Score] [int] NULL,
    [ViewCount] [int] NULL,
    [Body] [varchar](max) NULL,
    [OwnerUserId] [int] NULL,
    [LastEditorUserId] [int] NULL,
    [LastEditorDisplayName] [varchar](300) NULL,
    [LastEditDate] [datetime] NULL,
    [LastActivityDate] [datetime] NULL,
    [Title] [varchar](300) NULL,
    [Tags] [varchar](300) NULL,
    [AnswerCount] [int] NULL,
    [CommentCount] [int] NULL,
    [FavoriteCount] [int] NULL,
    [ParentId] [int] NULL,
    [ClosedDate] [datetime] NULL,
    [CommunityOwnedDate] [datetime] NULL,
    [OwnerDisplayName] [varchar](300) NULL
    index Id_Posts
    hash ([OwnerUserId])
    WITH(BUCKET_COUNT = 3000000)
) WITH( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA)
```

Fonte: do autor, 2017

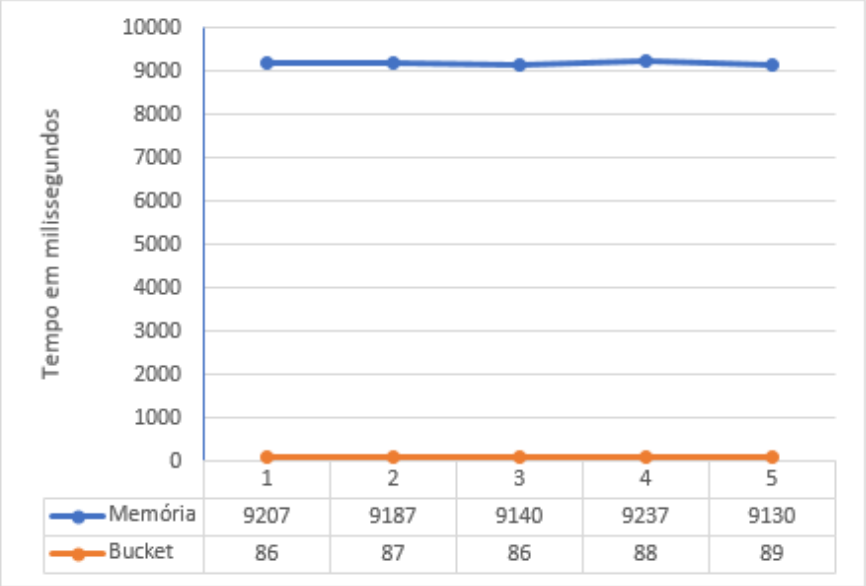
Com a execução da *query* do exemplo 4.1.2, onde a operação em memória obteve médias de cerca de 9 segundos, com a adição do novo *index* a média ficou em 87 milissegundos, ou seja, um desempenho 99% melhor com a adição de um único índice, fazendo com que a operação se tornasse visivelmente de execução instantânea, conforme observa-se na Figura 50.

Figura 50 – Desempenho de consulta com *Bucket*

Memória	Bucket
00:00:09.207	00:00:00.086
00:00:09.187	00:00:00.087
00:00:09.140	00:00:00.086
00:00:09.237	00:00:00.088
00:00:09.130	00:00:00.089

Média	
00:00:09.180	00:00:00.087

Diferença
10428%



Fonte: do autor, 2017

4.4 EXPERIMENTO IV – OPERAÇÕES COM *COMPILED STORED PROCEDURE*

O quarto conjunto de experimentos teve como foco a utilização do novo formato de *stored procedure.* Foram realizados experimentos com 1 milhão de registros em tabelas criadas exclusivamente para este teste, conforme Figura 51, onde foram criadas 3 versões da tabela *Comments*, uma para a inserção de dados em tabela armazenada em disco, outra para a inserção tradicional em memória e por fim, a terceira tabela em memória para a inserção através de *procedure.*

O motivo pelo qual foram criadas as 3 tabelas é para que os experimentos ficassem separados, não tendo conflito durante a inserção de seus registros. As duas versões em memória foram criadas com *hash index* de 1 milhão, pois seria a quantidade pré-determinada de registros para o experimento.

Figura 51 – Script criação tabelas de teste para procedure

```
--Tabela com armazenamento em Disco
CREATE TABLE [dbo].[CommentsDisco](
    [Id] [int] NOT NULL PRIMARY KEY,
    [PostId] [int] NULL,
    [Score] [int] NULL,
    [Text] [varchar](max) NULL,
    [CreationDate] [datetime] NULL,
    [UserId] [int] NULL,
    [UserDisplayName] [varchar](300) NULL
)

--Tabela com armazenamento em Memória
CREATE TABLE [dbo].[CommentsMemoria](
    [Id] [int] NOT NULL PRIMARY KEY NONCLUSTERED HASH WITH (BUCKET_COUNT=1000000),
    [PostId] [int] NULL,
    [Score] [int] NULL,
    [Text] [varchar](max) NULL,
    [CreationDate] [datetime] NULL,
    [UserId] [int] NULL,
    [UserDisplayName] [varchar](300) NULL
) WITH( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA)

--Tabela com armazenamento em Memória através de procedure
CREATE TABLE [dbo].[CommentsMemoriaProc](
    [Id] [int] NOT NULL PRIMARY KEY NONCLUSTERED HASH WITH (BUCKET_COUNT=1000000),
    [PostId] [int] NULL,
    [Score] [int] NULL,
    [Text] [varchar](max) NULL,
    [CreationDate] [datetime] NULL,
    [UserId] [int] NULL,
    [UserDisplayName] [varchar](300) NULL
) WITH( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA)
```

Fonte: do autor, 2017

4.4.1 Operações de inserção

A realização do experimento, visa verificar o ganho real entre as operações tradicionais de inserção com o desempenho da utilização de *procedure* para o mesmo fim. Foi criada uma *procedure* com o exclusivo objetivo de inserir registros na tabela CommentsMemoriaProc, a qual pode ser observada na Figura 52, onde a informação de *native_compilation* faz com que o SGBD saiba que deve gerar DLL para a *procedure* e *SchemaBinding* define que será associada com a tabela, tendo comportamento semelhante a utilização de *Foreign Key*, não sendo possível a exclusão da tabela sem antes a exclusão da *procedure*.

Figura 52 – Script *procedure* de inserção

```
create procedure spInserirDados
    @Id int,
    @PostId int,
    @Score int,
    @Text varchar(max),
    @CreationDate datetime,
    @UserId int,
    @UserDisplayName varchar(300)
with native_compilation, schemabinding
as
begin atomic
with (transaction isolation level = snapshot, language = N'us_english')
    declare @i int = 1
    while @i <= @Id
    begin
        insert into dbo.CommentsMemoriaProc (Id,PostId,Score,Text,CreationDate,UserId,UserDisplayName
        values (@i,@PostId,@Score,@Text,@CreationDate,@UserId,@UserDisplayName)

        set @i += 1
    end
end
```

Fonte: do autor, 2017

A inserção foi realizada através de *loop* simples limitado a 1 milhão de registros para as duas instruções de inserção tradicionais, para a utilização da *procedure* foi realizado apenas a execução com o valor máximo de registros a serem inseridos.

Os resultados obtidos foram bastante similares para as duas instruções de inserção tradicionais, resultando em um tempo médio de 2 minutos e 5 segundos para a inserção em memória e 2 minutos e 30 segundos para a mesma operação em disco, contudo, a inserção através da utilização da *stored procedure* foi concluída em apenas 9 segundos.

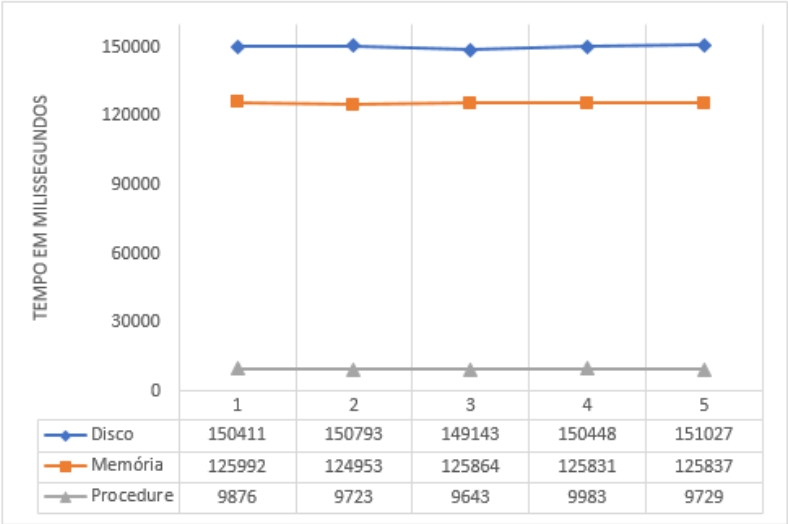
Este resultado pode ser observado na Figura 53, sendo a *procedure* uma forma muito eficiente para a inserção de grandes quantias de dados, isso é possível pois quando as *procedures* são executadas o SQL Server verifica o melhor plano de execução para a operação, após ter seu plano verificado é armazenado em arquivos do próprio sistema para que sempre que a mesma for executada, busque dessa “biblioteca” de planos de execução.

Figura 53 – Resultados obtidos com a inserção por *procedure*

Disco	Memória	Procedure
00:02:30.411	00:02:05.992	00:00:09.076
00:02:30.793	00:02:04.953	00:00:09.723
00:02:29.143	00:02:05.864	00:00:09.643
00:02:30.448	00:02:05.831	00:00:09.983
00:02:31.027	00:02:05.837	00:00:09.729

Média		
00:02:30.364	00:02:05.695	00:00:09.791

Diferença	
Disco x Memória	20%
Memória x Procedure	1184%
Disco x Procedure	1436%



Fonte: do autor, 2017

Os parâmetros utilizados para a inserção e os *scripts* dos experimentos citados acima, podem ser verificados na Figura 54, onde principalmente foram definidas a quantia de 1 milhão de registros para a ação do *while*, o texto que seria inserido e a obtenção do horário em que teve início a execução do bloco, sendo que o texto foi gerado com informações aleatórias, somente com o foco de obter custo para a conclusão das operações.

Ao finalizar a inserção de cada bloco, foi adicionado um *Select* com o único objetivo de retornar o tempo decorrido entre o início e o fim do trecho de inserção, sendo a forma mais precisa para obter este dado.

Após cada execução do conjunto de inserções, foram coletados os tempos e realizada a exclusão de todos os registros das 3 tabelas envolvidas no experimento, fazendo com que todas as operações tivessem as tabelas limpas para a próxima inserção, não havendo qualquer interferência.

Figura 54 – Scripts de inserção de dados

```
declare @i int = 1
declare @id int = 1000000
declare @PostId int = 2334712
declare @UserId int = 150939
declare @Text varchar(100) = 'Teste123 Teste123 Teste123 Teste123 Teste123 Teste123 Teste123'
declare @ms int
declare @inicio datetime2 = sysdatetime();

begin tran
    while @i <= @id
    begin

        insert into CommentsDisco (Id,PostId,Score,Text,CreationDate,UserId,UserDisplayName)
        values (@i,@PostId,123,@Text,getdate(),150939,'Teste Vargas')

        set @i += 1
    end
commit

set @ms = DATEDIFF(ms,@inicio,sysdatetime());
select cast(@ms as varchar(10)) + ' ms - Tabela em Disco'

-------------------------------------

set @i = 1
set @inicio = sysdatetime()

begin tran
    while @i <= @id
    begin

        insert into CommentsMemoria (Id,PostId,Score,Text,CreationDate,UserId,UserDisplayName)
        values (@i,@PostId,123,@Text,getdate(),150939,'Teste Vargas')

        set @i += 1
    end
commit

set @ms = DATEDIFF(ms,@inicio,sysdatetime());
select cast(@ms as varchar(10)) + ' ms - Tabela em memória'

---------------------

set @inicio = sysdatetime()

exec spInserirDados @id,@PostId,123,@Text,@starttime,150939,'Teste Vargas'

set @ms = DATEDIFF(ms,@inicio,sysdatetime());
select cast(@ms as varchar(10)) + ' ms Tabela em memória com uso de Proc'
```

Fonte: do autor, 2017

4.4.2 Operações de exclusão

Com os dados gerados pela operação de inserção realizada anteriormente, foram criadas instruções com o caminho inverso, a fim de realizar a exclusão de todos os registros dentro de

um *loop*, removendo assim um por vez. A procedure criada para o experimento está disposta na Figura 55.

Figura 55 – Script procedure de remoção

```
create procedure spExcluirDados
    @Id int
with native_compilation, schemabinding
as
begin atomic
with (transaction isolation level = snapshot, language = N'us_english')
    declare @i int = 1
    while @i <= @Id
    begin
        delete dbo.CommentsMemoriaProc
        Where Id = @i

        set @i += 1
    end
end
```

Fonte: do autor, 2017

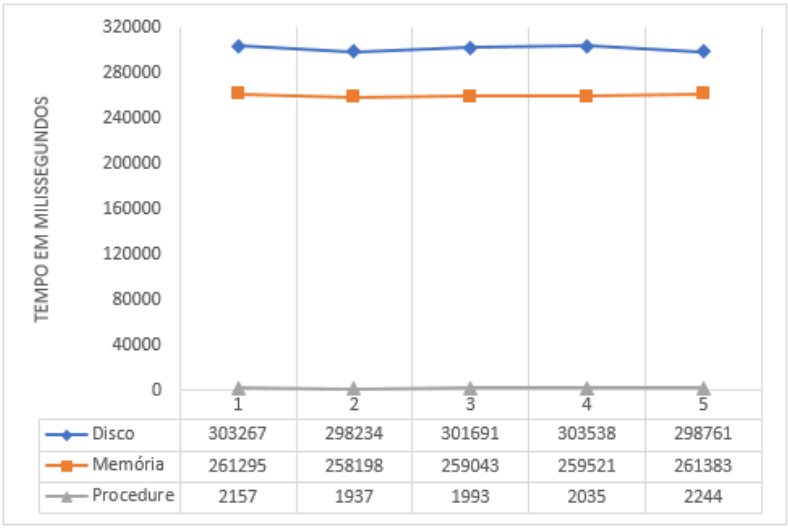
Os resultados apresentados na Figura 56 obtidos através da utilização do *script* da Figura 57 demonstram que o custo foi ainda melhor do que a operação de inserção, concluindo-se que a utilização da tecnologia *in-memory* em conjunto com os procedimentos compilados é o melhor resultado obtido nos estudos. Ressaltando a importância da utilização dos conjuntos corretos na indexação de tabelas.

Figura 56 – Resultados obtidos com a exclusão por *procedure*

Disco	Memória	Procedure
00:05:03.267	00:04:21.295	00:00:02.156
00:04:58.234	00:04:18.198	00:00:01.937
00:05:01.691	00:04:19.043	00:00:01.993
00:05:03.538	00:04:19.521	00:00:02.035
00:04:58.761	00:04:21.383	00:00:02.244

Média		
00:05:01.098	00:04:19.888	00:00:02.073

Diferença	
Disco x Memória	16%
Memória x Procedure	12436%
Disco x Procedure	14423%



Fonte: do autor, 2017

A diferença da relação memória por *procedure* é muito grande, sendo que ambos os métodos de exclusão podem ser criados em um mesmo ambiente (não sendo necessária nenhuma alteração de tabelas), mas com a falta de informações devido a ser uma tecnologia muito recente, tende a levar algum tempo até que se obtenha tal conhecimento.

Figura 57 – Scripts de exclusão de dados

```
declare @i int = 1
declare @id int = 1000000
declare @timems int
declare @starttime datetime2 = sysdatetime();

begin tran
    while @i <= @id
    begin

        delete dbo.CommentsDisco
        where Id = @i

        set @i += 1
    end
commit

set @timems = DATEDIFF(ms,@starttime,sysdatetime());
select cast(@timems as varchar(10)) + ' ms - Tabela em disco'

--------------------------------------

set @i = 1
set @starttime  = sysdatetime();

while @i <= @id
begin

    delete dbo.CommentsMemoria
    where Id = @i

    set @i += 1
end

set @timems = DATEDIFF(ms,@starttime,sysdatetime());
select cast(@timems as varchar(10)) + ' ms - Tabela em memória'

---------------------

set @starttime = sysdatetime()

exec spExcluirDados @id

set @timems = DATEDIFF(ms,@starttime,sysdatetime());
select cast(@timems as varchar(10)) + ' ms  - Tabela em memória com proc'
```

Fonte: do autor, 2017

4.5 EXPERIMENTO IV – COMPRESSÃO

Durante o estudo teórico sobre o SGBD de estudo, foi visto as formas de compressão suportadas pelo SQL Server, com isso, foram aplicadas as formas de compressão de dados por página e por linha em todas as tabelas do estudo, a fim de obter o ganho real em economia de espaço de armazenamento/memória em uso.

Como material para o estudo, serão usadas as tabelas já preenchidas com todos os dados dos experimentos anteriores, sendo aplicadas as compressões separadamente em cada uma das tabelas e comparando os resultados obtidos.

4.5.1 Experimentos em disco

Para a realização dos testes, foi utilizada a instrução de alteração de tabela do *SQL Server*, que juntamente com a instrução de qual o método de compressão será utilizado, aplica e realiza o processamento de seus dados. O método de compressão deve ser vinculado a tabela, conforme a sintaxe apresentada na Figura 58.

Figura 58 – *Script* de aplicação de compressão

```
--Compressão por linha
ALTER TABLE BADGES REBUILD PARTITION = ALL
WITH (DATA_COMPRESSION = ROW)

--Compressão por página
ALTER TABLE BADGES REBUILD PARTITION = ALL
WITH (DATA_COMPRESSION = PAGE)
```

Fonte: do autor, 2017

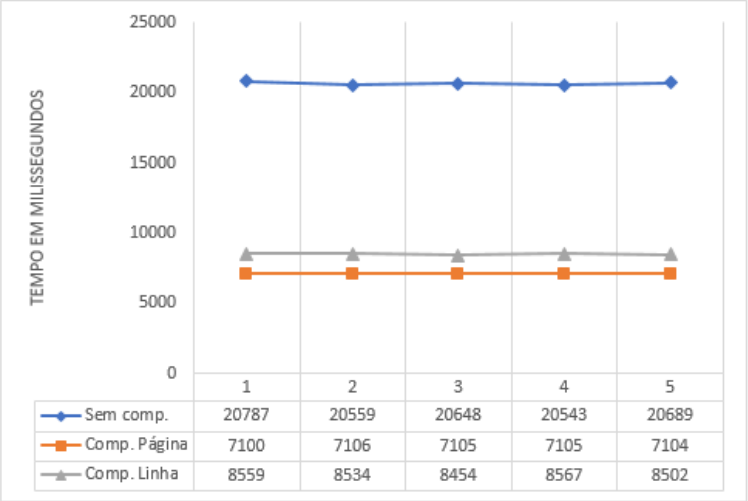
Após a compressão das tabelas *Badges* e *Posts*, foram realizados testes de desempenho na consulta de dados presentes em ambas. Para a consulta simples na tabela *Badges* de todos os registros de um usuário específico, que resultava em 71 registros, o tempo médio na tabela sem compressão ficou em 20 segundos. Enquanto isso, após a aplicação da compressão por linha, a consulta finalizou em 8.5 segundos, sendo 1 segundo e meio a mais do que o resultado da compressão por página, conforme Figura 59.

Figura 59 – Desempenho da compressão na tabela *Badges*

Sem comp.	Comp. Página	Comp. Linha
00:00:20.788	00:00:07.100	00:00:08.559
00:00:20.559	00:00:07.106	00:00:08.534
00:00:20.648	00:00:07.105	00:00:08.454
00:00:20.543	00:00:07.105	00:00:08.567
00:00:20.689	00:00:07.104	00:00:08.502

Média		
00:00:20.645	00:00:07.104	00:00:08.523

Diferença	
Sem comp. X Comp. página	191%
Comp. página X Comp. linha	-17%
Sem comp. X Comp. linha	142%



Fonte: do autor, 2017

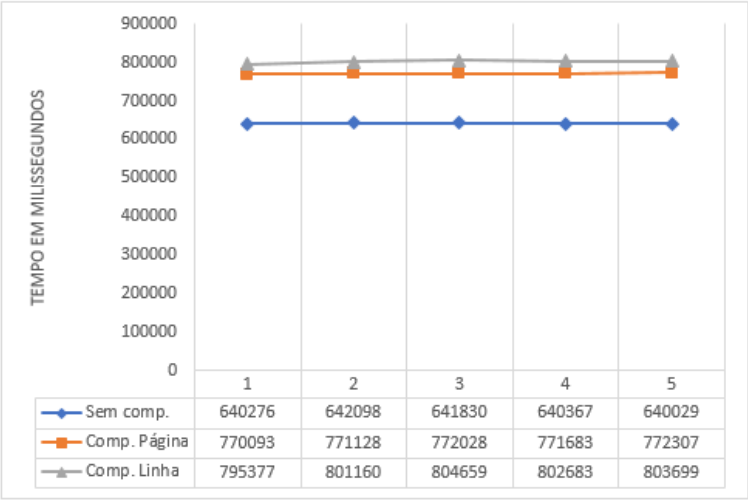
Para a consulta na tabela *Posts* que obtinha 45 registros para o mesmo usuário, o tempo obtido sem compressão foi de 10 minutos e 40 segundos, resultando em melhor desempenho do que os tempos pós aplicação de compressão. Para a compressão em linha, o tempo médio foi de 13 minutos e meio, enquanto na compressão por página ficou com 12 minutos e 40 segundos, ocasionando no aumento do custo de processamento após ambas as compressões, conforme Figura 60.

Figura 60 – Desempenho da compressão na tabela *Posts*

Sem comp.	Comp. Página	Comp. Linha
00:10:40.276	00:12:50.093	00:13:15.377
00:10:42.098	00:12:51.128	00:13:21.160
00:10:41.830	00:12:52.028	00:13:24.659
00:10:40.367	00:12:51.683	00:13:22.683
00:10:40.029	00:12:52.307	00:13:23.699

Média		
00:10:40.920	00:12:51.447	00:13:21.517

Diferença	
Sem comp. X Comp. página	-17%
Comp. página X Comp. linha	-4%
Sem comp. X Comp. linha	-20%



Fonte: do autor, 2017

Sendo assim, deve-se levar em consideração o ganho de processamento nas operações com a quantidade de espaço em disco que será economizada com quaisquer dos métodos de compressão. Para se obter a informação de quanto será a economia de espaço com a aplicação dos métodos de compressão, o *SQL Server* possui uma *procedure* nativa do sistema, que pode ser encontrada executada pelo comando *EXEC sp_estimate_data_compression_savings*.

Com a execução da procedure citada, juntamente com os parâmetros de nome da tabela e método de compressão, pode-se obter os dados apresentados na Quadro 7, onde se tem o espaço original ocupado e qual seria o espaço após os métodos de compressão.

Quadro 7 – Tamanho ocupado por cada tabela em disco

	Tamanho em KB			Porcentagem de Redução	
Tabela	**Original**	**Row**	**Page**	**Row**	**Page**
Posts	34235696	33058296	33006504	-3,44%	-3,59%
PostHistory	52628672	52564000	52303800	-0,12%	-0,62%
Badges	864352	633960	307416	-26,65%	-64,43%
PostLinks	109224	74320	60424	-31,96%	-44,68%
Users	826272	676912	532432	-18,08%	-35,56%
Votes	3870688	2356648	1482656	-39,12%	-61,70%
Comments	15656096	15158192	15168240	-3,18%	-3,12%
Tags	1912	1496	1496	-21,76%	-21,76%

Fonte: do Autor, 2017

4.5.2 Experimentos em memória

Conforme documentação da Microsoft, ainda não existe a possibilidade de compressão para tabelas em memória no SQL Server, não sendo possíveis experimentos com a utilização da base de dados em memória, o que deixa a ferramenta em desvantagem se comparada ao seu concorrente desenvolvido pela Oracle, o TimesTen. Contudo, é possível obter os dados de consumo de memória através de *scripts* nativos do SQL Server.

O Quadro 8 representa o consumo de memória por cada tabela, que é apresentada com duas colunas principais, onde consta a memória alocada para a utilização em cada tabela e a memória real que está sendo usada por cada.

Quadro 8 – Memória alocada por tabela

Tabela	Tamanho em KB	
	Memória alocada	Memória em uso
Posts	36007296	35273057
PostHistory	26554560	26081939
Badges	3259392	3243338
PostLinks	209664	208477
Users	1172224	1145432
Votes	6613760	6581359
Comments	12448384	12368821
Tags	3520	3119

Fonte: do autor, 2017

4.6 CONSIDERAÇÕES DOS EXPERIMENTOS

Durante a realização dos testes, alguns erros que impediram o prosseguimento dos experimentos tiveram de ser ajustados. Um dos principais problemas foi o grande consumo de memória por parte do servidor, chegando muitas vezes a 100% do uso para o SQL Server, fazendo com que nenhuma execução fosse finalizada com sucesso. Para a correção do problema citado, tiveram de ser realizadas reinicializações do serviço *SQL Server Windows NT*, que era responsável pelo gerenciamento do banco de dados em memória.

Com a reinicialização do serviço, foi diagnosticado outro problema, que se trata da disponibilidade da base de dados para o uso. O banco de dados em disco pode ser utilizado segundos após a inicialização do *SQL Server Management Studio*, não prejudicando demandas da utilização do SGBD.

Para a utilização do banco de dados em memória, foi necessário aguardar cerca de 45 minutos para que o mesmo estivesse disponível para a utilização, esse tempo excessivo se dá devido ao grande volume de dados que tem necessidade de ser realocado para a memória do servidor.

5. COMPARAÇÃO COM TRABALHO RELACIONADO

O Trabalho de Horn (2016) estudou a ferramenta TimesTen, da Oracle, para obter informações de processamento e armazenamento de dados em memória e o Oracle para a processamento em disco. Com a possibilidade de comparação com o estudo prévio, foi utilizada a mesma base de dados na ferramenta SQL Server 2016. Contudo, a comparação entre as ferramentas não poderá ser realizada considerando simplesmente o tempo de execução em segundos.

Não foi possível a utilização do mesmo servidor utilizado por Horn (2016), pois o mesmo não estava disponível para a aplicação do novo estudo, sendo a configuração do novo servidor diferente e tecnicamente inferior, conforme Quadro 9.

Quadro 9 – Configurações dos servidores

Característica	Servidor TimesTen	Servidor SQL Server
Sistema operacional	Windows Server 2012 R2	Windows Server 2016 Standard
Processador	Intel (R) Xeon (R) CPU E5-2690 v3 2,6 GHz	Intel (R) Xeon (R) CPU E5506 2.13Ghz
Memória RAM	256 GB	144 GB
Núcleos físicos	24	8

Fonte: do Autor, 2017

Durante o processo de experimento da inserção dos dados para estudo, verificou-se que a metodologia utilizada anteriormente não contemplava plenamente todos os registros disponibilizados no XML principal, nem a quantidade de dados em cada coluna, limitando-os em 400 caracteres.

Apesar das diferenças apresentadas anteriormente, realizou-se alguns experimentos a fim de viabilizar a comparação com os resultados obtidos na ferramenta TimesTen. Não sendo possível a comparação em tempos de execução, foram realizadas comparações em nível de escala percentual.

5.1 OPERAÇÕES

Para a realização de testes de algumas operações básicas, as instruções utilizadas foram cópias idênticas extraídas do trabalho de Horn (2016), sendo realizados três execuções para a

obtenção de tempos médios. O primeiro teste consiste na consulta de dados entre duas tabelas através do *JOIN* entre as tabelas *Posts* e *Users*, conforme Figura 61, tendo como retorno todos os *posts* de um determinado usuário. Este experimento gerou a primeira divergência de registros, onde o estudo atual encontra 187 registros, sendo 1 a mais que o estudo anterior.

Figura 61 – Script de consulta trabalho anterior

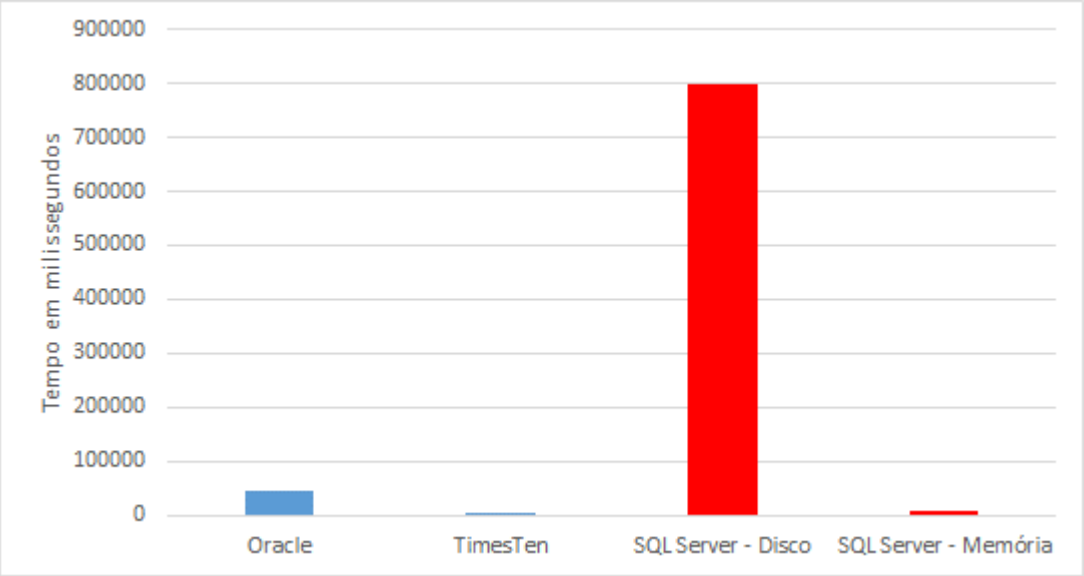
```
select
  P.Id as PostId
  , p.body as Corpo
  , p.Title as Titulo
  , p.Tags as Tags
  , u.DisplayName as NomeUsuario
  , p.ViewCount as QtVisualizacoes
  , p.Score as Votos
  , p.CreationDate as DataCriacaoPost
  , p.LastEditDate as DataUltimaAtualizacao
from
  Posts P
Inner join Users u
  on u.Id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

Fonte: Horn, 2016

Após a execução da instrução acima citada em ambos os ambientes, realizou-se a coleta do tempo em milissegundos e aplicou-se a função de média, obtendo-se o tempo médio das execuções. Enquanto o Oracle obteve médias de 44,4 segundos para a realização da operação em disco, o SQL Server obteve 13 minutos e 20 segundos, enquanto em memória foi de 1,7 segundos no TimesTen e 9 segundos no SQL Server.

Percebe-se que há uma grande diferença nos resultados, principalmente quando a execução é em disco. Porém, ao analisar os dados constata-se que em porcentagem de desempenho entre as soluções de mesma ferramenta, o ganho da memória no SQL Server chegou a 98%, sendo 2% a mais do que o TimesTen. A Figura 62 apresenta graficamente a diferença entre as operações em disco e memória dos dois SGBDs.

Figura 62 – Desempenho de ambas as ferramentas



Fonte: do autor, 2017

O experimento aplicado para a alteração de dados, teve como propósito a alteração da tabela *Comments* através do filtro *LIKE* da coluna *text*, a qual não possui nenhum tratamento de indexação, o *script* utilizado encontra-se na Figura 63.

Figura 63 – Script de alteração de dados

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

Fonte: Horn, 2017

Resultando na alteração de 2.857.537 registros, 43.532 a mais que no experimento do estudo anterior, os tempos anteriormente obtidos no Oracle de 9 minutos e 45 segundos para a operação em disco e 2 minutos e 40 segundos para o processo em memória no Timesten, foram melhores que os atuais 24 minutos e 15 segundos para a memória e 38 minutos e 30 segundos para o disco com a execução do SQL Server.

Realizando a comparação em segundos, pode-se observar que as operações no TimesTen tiveram ganho de até 72% no seu desempenho, enquanto o SQL Server obteve apenas a metade disso, 36% de melhora. Nesse caso em especifico, existe grande possibilidade de que a restrição de 400 caracteres do estudo anterior impacte de forma significativa nos resultados obtidos.

Para a realização do comparativo de inserção, houve a necessidade da criação de linhas de instrução no mesmo formato do estudo anterior, no qual foi utilizada uma linha de *INSERT* para cada linha de registro, fazendo com que as operações ficassem com maior lentidão, resultado na *Query* exibida na Figura 64.

Figura 64 – Inserção de dados em linha

```
insert into Tags values (150001,'.net',233796,3624959,3607476)
insert into Tags values (150002,'html',545013,3673183,3673182)
insert into Tags values (150003,'javascript',1143897,3624960,3607052)
insert into Tags values (150004,'css',395577,3644670,3644669)
insert into Tags values (150005,'php',933625,3624936,3607050)
insert into Tags values (150008,'c',220080,3624961,3607013)
insert into Tags values (150009,'c#',962767,3624962,3607007)
insert into Tags values (150010,'c++',452918,3624963,3606997)
insert into Tags values (150012,'ruby',161786,3624964,3607043)
insert into Tags values (150014,'lisp',4532,3656743,3656742)
insert into Tags values (150016,'python',587704,3624965,3607014)
insert into Tags values (150017,'java',1086539,3624966,3607018)
insert into Tags values (150018,'regex',146736,3624967,3607017)
insert into Tags values (150019,'xml',138949,3624968,3607588)
insert into Tags values (150021,'mysql',402070,3624969,3607033)
insert into Tags values (150022,'sql',335107,3625226,3607304)
```

Fonte: do autor, 2017

Após a inserção de 10 mil registros, realizou-se a coleta e análise dos tempos de execução e sua média, chegando assim ao resultado muito semelhante entre as operações do SQL Server, no qual o disco teve apenas 100 milissegundos a mais que a memória, com tempos de 3.3 e 3.4 segundos, respectivamente.

Com a realização das comparações com os resultados obtidos por Horn (2016), percebe-se que para uma validação precisa há a necessidade de que seja utilizado o mesmo *hardware* para ambos os SGBDs, sendo que, os dados devem ser gerados partindo da mesma aplicação, assim não perderá linhas de registro e nem será restringido o número de caracteres armazenados em um único registro.

CONCLUSÃO

Com a constante queda do custo de memória, a utilização da tecnologia em memória para banco de dados vem se tornando cada dia mais viável. O maior investimento pode ser recuperado em pouco tempo à medida que as empresas aproveitarem o menor tempo de resposta para as suas operações. Além disso, o tempo de resposta das operações pode resultar em maior rendimento dos funcionários, já que há a necessidade da informação para a realização de diversas atividades diárias.

Acredita-se que os SGBDs tradicionais, com armazenamento realizado em disco ainda serão utilizados por muitos anos. No entanto, os movimentos tanto do mercado de fornecedores de SGBDs quanto das empresas, permite afirmar que os *In-Memory Databases* irão ocupar uma fatia crescente nos próximos anos. Os experimentos realizados neste trabalho reforçam esta conclusão. A utilização da versão 2016 do SQL Server demonstrou ótima performance para o processamento de dados e grande simplicidade para tratamento dos mesmos, com fácil migração dos dados de disco para a memória.

A Microsoft incluiu os primeiros recursos de Banco de Dados em memória na versão 2014 do SQL Server. Contudo, a versão continha diversos problemas ocasionando em uma baixa aceitação por parte das empresas. Na versão 2016 foram realizadas melhorias e correções necessárias, e o mesmo se tornou uma das grandes potencias para dados em memória, superando a Oracle em algumas pesquisas de mercado.

Nos experimentos realizados, pode-se perceber o quão benéfico pode ser o uso da memória para as diversas operações no SGBD, proporcionando ganhos extremamente expressivos quando comparados com o tradicional armazenamento em disco. Contudo, foram observadas algumas limitações com a utilização da memória, principalmente com a necessidade de tempo do SGBD após sua reinicialização, fazendo com que todos os dados sejam recarregados para a memória principal, o que para a base de dados do estudo demorou quase uma hora para finalizar. Considerando que em empresas a reinicialização do Banco de Dados acontece poucas vezes durante o ano, com planejamento prévio, o mesmo não deve ser considerado um problema.

A possibilidade de manter os dados armazenados em ambos os ambientes em um único SGBD, fez com que houvesse fácil manuseio dos dados para a migração entre disco e memória, não havendo necessidade da utilização de nenhuma aplicação da Microsoft para importação de

dados, realizando a carga através da leitura do mesmo XML utilizado para inserção dos dados em disco.

Os ajustes realizados nas tabelas em memória, tanto na criação como na alteração de indexação, foram encontrados principalmente na documentação do produto e nos fóruns de desenvolvedores, os quais não medem esforços para ajudar a quem precisar.

O trabalho atingiu plenamente os objetivos propostos realizando a análise técnica e prática da utilização do SQL Server 2016, apresentando as características e técnicas utilizadas por este SGDB no armazenamento em disco e em memória, relacionando vantagens e desvantagens, em termos de desempenho, armazenamento e custos.

Com o objetivo de realizar a análise técnica e prática do SGBD SQL Server com armazenamento em disco e memória, apresentando suas principais características, benefícios e deficiências em suas diversas operações, melhorias possíveis para melhor aproveitamento de tempo de processamento, apresentação de gráficos comparativos entre operações em ambos ambientes com dados reais de um fórum de programação confiável, foram realizados os experimentos propostos e obteve-se o que se era esperado no início, a melhor opção em desempenho é a memória, contudo, ainda não é a melhor no quesito custo.

Foi comprovado que o SGBD da Microsoft pode ser utilizado de forma eficiente para os dois ambientes, contudo, em alguns casos, na comparação com os resultados obtidos por Horn (2016) utilizando o SGBD TimesTen da Oracle, deixou a desejar. A grande diferença em tempo de processamento pode ser devido a configuração do servidor, inferior ao utilizado anteriormente, pois com o cálculo em porcentagem de ganho as duas soluções foram muito semelhantes.

Como sugestão para um trabalho futuro, seria interessante a comparação entre o SQL Server e o TimesTen em um único servidor com ambos os SGBDs, para que assim possa ser concluído qual opção tem a melhor relação de benefícios. Outro trabalho interessante seria a comparação com SGBDs que possuem armazenamento em memória e são de uso gratuito, como o Redis e o VoltDB, os quais aparecem através de citações em artigos científicos lidos para a realização do trabalho atual.

REFERÊNCIAS BIBLIOGRÁFICAS

ABRAMOVA, V., Bernardino, J., Furtado, P., 2014. **Which NoSQL database? A performance overviews**. Open J. Databases 1, 17–24.

APPLEBY, John. **SAP HANA – Scale-up or scale-out Hardware**. Dec 2014. Acesso em: 10/05/2017. Disponível em: <https://blogs.saphana.com/2014/12/10/sap-hana-scale-scale-hardware>

ARTEMIOU, Artemakis. **Introducing SQL Server In-Memory OLTP**. Sep 2015. Acesso em: 17/05/2017. Disponível em: <https://www.simple-talk.com/sql/learn-sql-server/introducing-sql-server-in-memory-oltp/>

ARUMILLI, Uday. **SQL THE ONE**. Dec 2016. Acesso em: 17/05/2017. Disponível em: < https://books.google.com.br/books?id=qP27DQAAQBAJ>

CEBOLLERO, Miguel; COLES, Michael; NATARAJAN, Jay. **Pro T-SQL Programmer's Guide**. Apress, 2015.

DIACONU, Cristian et al. **Hekaton: SQL server's memory-optimized OLTP engine**. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013. p. 1243-1254.

EVANS, Chris. (2014). **In-memory databases – what they do and the storage they need.** Acesso em: 27/05/2017. Disponível em: < http://www.computerweekly.com/feature/In-memory-databases-What-they-do-and-the-storage-they-need>

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.

GUPTA,Mohit Kumar; VERMA,Vishal; VERMA, Megha Singh. **In-Memory Database Systems - A Paradigm Shift**. International Journal of Engineering Trends and Technology (IJETT) – Volume 6 Number 6- Dec 2013. Acesso em: 14/03/2017. Disponível em: <http://arxiv.org/ftp/arxiv/papers/1402/1402.1258.pdf>

HORN, Cristiano. **ANÁLISE TÉCNICA E APLICAÇÃO DE BANCO DE DADOS EM MEMÓRIA.** Acesso em: 11/03/2017. Disponível em: < http://tconline.feevale.br/tc/files/0002_4241.pdf >

KEMPER, Alfons et al. **Transaction Processing in the Hybrid OLTP&OLAP.** Main-Memory Database System HyPer. 2013.

KLEIN, Andy (2017). **Hard Drive cost per gigabyte.** Acesso em: 15/12/2017. Disponível em: < https://www.backblaze.com/blog/hard-drive-cost-per-gigabyte>

KOROTKEVITCH, Dmitri. **Pro SQL Server Internals.** 2nd Edition. 2016.

KOSEV, Kaloyan (2016). **Deep dive into Nonclustered indexes for In-Memory OLTP Tables.** Acesso em: 01/11/2017. Disponível em: < https://www.sqlshack.com/deep-dive-nonclustered-indexes-memory-oltp-tables>

LOSHIN, David; **Top-of-Mind Time for In-Memory Data** (2014).

MIAN, Rizwan. **Cost-Effective Resource Configurations for Executing Data-Intensive Workloads in Public Clouds**. 2013. Tese de Doutorado. Queen's University.

MICROSOFT (2012). **Scaling Out SQL Server.** Acesso em: 01/05/2017. Disponível em: < https://msdn.microsoft.com/en-us/library/aa479364.aspx>

MICROSOFT (2015). **Guidelines for Using Indexes on Memory-Optimized Tables.** Acesso em: 27/04/2017. Disponível em: < https://msdn.microsoft.com/en-us/library/dn133166.aspx>

MICROSOFT (2016a). **Microsoft SQL SERVER: Mission-Critical Performance**. Pp. 7-8, Mar.

MICROSOFT (2016b). **Columnstore indexes - Overview.** Acesso em: 03/05/2017. Disponível em: <https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview>

MICROSOFT (2016c). **ACID Properties.** Acesso em: 25/04/2017. Disponível em: < https://msdn.microsoft.com/en-us/library/aa480356.aspx >

MICROSOFT (2016d). **Control transaction durability.** Acesso em: 27/04/2017. Disponível em: < https://docs.microsoft.com/en-us/sql/relational-databases/logs/control-transaction-durability >

MICROSOFT (2016e). **Memory management architecture guide.** Acesso em: 05/06/2017. Disponível em: < https://docs.microsoft.com/en-us/sql/relational-databases/memory-management-architecture-guide >

MICROSOFT (2016f). **Estimate memory requirements for memory-optimized tables.** Acesso em: 15/12/2017. Disponível em: < https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/estimate-memory-requirements-for-memory-optimized-tables>

MICROSOFT (2017a). **Defining durability for memory-optimized objetcs.** Acesso em: 20/03/2017. Disponível em: <https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/defining-durability-for-memory-optimized-objects>

MICROSOFT (2017b). **Download SQL Server Management Studio (SSMS).** Acesso em: 20/10/2017. Disponível em: < https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms>

NAJAM (2016). **Rise of the in-Memory Database.** Acesso em: 07/05/2017. Disponível em: < https://designcoral.com/article/rise-of-the-in-memory-database>

NVD (2017). **National Vulnerability Database.** Acesso em: 20/10/2017. Disponível em: < https://nvd.nist.gov>

NEVAREZ, Benjamin, (2016). **High Performance SQL Server.**

NUODB (2017). **License Cost Calculator.** Acesso em: 20/10/2017. Disponível em: < https://www.nuodb.com/product/license-cost-calculator>

OTEY, Michael (2014). **Application Performance with the In-Memory OLTP Engine.** Acesso em: 02/10/2017. Disponível em: < http://beta.itprotoday.com/microsoft-sql-server/rev-application-performance-memory-oltp-engine >

PLATTNER, Hasso; LEUKERT, Bernd. **The in-memory revolution: how SAP HANA enables business of the future**. Springer, 2015.

PLATTNER, Hasso; ZEIER, Alexander. **In-memory data management: technology and applications**. Springer Science & Business Media, 2012.

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.

SOLID IT (2017a). **DB Engines Ranking.** Acesso em: 14/05/2017. Disponível em: < https://db-engines.com/en/ranking>

SOLID IT (2017b). **System Properties Comparison Microsoft SQL Server vs. MySQL vs. Oracle vs. SAP HANA.** Acesso em: 14/05/2017. Disponível em: < https://db-engines.com/en/system/Microsoft+SQL+Server%3BMySQL%3BOracle%3BSAP+HANA>

STACKOVERFLOW (2014). **Database schema documentation for the public data dump and SEDE.** Acesso em: 16/08/2017. Disponível em: < https://meta.stackexchange.com/questions/2677/database-schema-documentation-for-the-public-data-dump-and-sede>

STACKOVERFLOW (2017). **Index of Stackexchange.** Acesso em: 27/08/2017. Disponível em: < https://ia800500.us.archive.org/22/items/stackexchange/>

VARGA, Stacia; CHERRY, Denny; D'ANTONI, Joseph (2016e). **Introducing Microsoft SQL Server 2016: Mission-Critical Applications, Deeper Insights, Hyperscale Cloud.**

YAMAN, S. G; **Introduction to Column-Oriented Database Systems** (Nov 2012).