UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ELVIS MARQUES TEIXEIRA

METISIDX: INDEXAÇÃO DE DADOS PREDITIVA

FORTALEZA

2018

ELVIS MARQUES TEIXEIRA

METISIDX: INDEXAÇÃO DE DADOS PREDITIVA

Dissertação apresentada ao Curso de do Programa de pós-graduação em computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Computação. Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Javam de Castro Machado

FORTALEZA

2018

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

T265m Teixeira, Elvis M..
MetisIDX: Indexação de Dados Preditiva / Elvis M. Teixeira. – 2018.
80 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2018.
Orientação: Prof. Dr. Javam de Castro Machado.

1. Indexação de Dados. 2. Aprendizado automático. 3. Indexação Adaptativa. I. Título.

CDD 005

ELVIS MARQUES TEIXEIRA

METISIDX: INDEXAÇÃO DE DADOS PREDITIVA

Dissertação apresentada ao Curso de do Programa de pós-graduação em computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Computação. Área de Concentração: Ciência da Computação

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Javam de Castro Machado (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Caetano Traina Júnior
Universidade de São Paulo (USP)

Prof. Dr. Angelo Brayner
Universidade Federal do Ceará (UFC)

À professora Helley Abreu, que deu sua vida no esforço de salvar seus alunos das chamas durante o trágico episódio ocorrido em Janaúba, Minas Gerais. Que sua memória viva como um lembrete do que um ser humano é capaz de fazer em benefício de outros.

AGRADECIMENTOS

Aos professores dos departamentos de Física e Ciência da Computação da Universidade Federal do Ceará, que como todo mestre têm a grandeza de compartilhar com outros o conhecimento que acumulam como resultado de sua história. E com isso influenciaram a minha.

Ao Prof. Dr. Javam de Castro Machado, pela orientação durante o processo de pesquisa que resultou na elaboração desta dissertação.

A todos os colegas do Laboratório de Sistemas e Bancos de Dados (LSBD), pela convivência agradável dos últimos anos e pela colaboração valorosa durante a prática de pesquisa.

Aos amigos Denis Morais, Paulo Amora e Bruno Leal, por ajudarem um estudante de pós graduação latino americano sem dinheiro no banco, sem parentes importantes e vindo do interior em diferentes momentos de apuros.

À minha família, pelo apoio sem o qual teria sido impossível a realização deste trabalho, e também pelo aprendizado vital sobre reciprocidade, partilha e compreensão.

E a todas as mulheres e todos os homens que acreditam e produzem software livre e de código aberto, por não venderem livros dos quais só se pode ler a capa. Vocês também contribuíram e ainda contribuem com a minha educação.

"Physics is like sex: sure, it may give some practical results, but that's not why we do it."

(Richard Feynman)

RESUMO

A análise exploratória de dados caracterizada por cargas de trabalho OLAP é atualmente uma tarefa rotineira tanto na academia quanto nos ambientes corporativos. Nestes cenários, a velocidade com que os dados são produzidos e os padrões de acesso desconhecidos e dinâmicos fazem da escolha de métodos de acesso uma tarefa desafiadora. Técnicas de indexação adaptativa propõe o uso de índices parciais construídos de forma incremental, em resposta à carga de trabalho e como um efeito colateral do processamento de consultas, otimizando o acesso preferencialmente para os intervalos de chave já requisitados. Este trabalho apresenta um desenvolvimento adicional destes princípios utilizando a história recente de acessos para predizer os intervalos de chave e indexá-los antecipadamente. Deste modo, as consultas seguintes encontram os dados em sua posição final na estrutura de dados, equivalente a um índice completo, e possivelmente localizados em uma camada mais alta da hierarquia de memória. *Adaptive Merging*, um dos principais modelos de indexação adaptativa, é usado como ponto de partida para a arquitetura em termos de estruturas de dados e algoritmos de busca e construção de índices. Também é utilizado como referencial para comparação de performance. Uma máquina de aprendizado extremo é utilizada para a predição dos intervalos de chave e é continuamente treinada com as novas consultas processadas. Os experimentos mostram ganho de cerca de um terço em tempos de resposta depois de 1000 consultas. As ações de indexação são feitas em paralelo, portanto nenhuma complexidade extra é adicionada ao mecanismo de processamento de consultas.

Palavras-chave: Indexação de Dados. Indexação Preditiva. Adaptive Merging

ABSTRACT

Exploratory data analysis characterized by OLAP query workloads over large databases are now commonplace on both academia and industry. In these scenarios, data production velocity and unknown and drifting access patterns make the choice of access methods a challenging task. In this context, adaptive indexing techniques propose the use of partial indexes that are incrementally built in response to the actual query sequence and as a byproduct of query processing to optimize the access only to the key ranges of interest. This work presents a further development to this principle by leveraging the recent query history to predict the next key ranges and index them in advance, so the queries arriving in the near future find data in its final representation and placed higher in the storage hierarchy, since data must be loaded into main memory in order to be indexed. Adaptive Merging is used as base architecture for the data structures and merge operations are executed in parallel with query execution instead of being the same operation. An extreme learning machine is used to perform key range forecasting and undergo continuous training by the indexing thread. The experiments show approximately one third gain in query response times after 1000 queries. The result is lower overall response times and the fact that the select operator does not incur the costs of indexing.

Keywords: Data Indexing. Predictive Indexing. Adaptive Merging.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

DML	*Data Manipulation Language*
DRAM	*Dynamic Random Access Memory*
ELM	*Extreme Learning Machine*
GIN	*Generalized Inverted Index*
GiST	*Generalized Search Tree*
GPU	*Graphics Processing Unit*
HASH	*Hash Access Method*
HD	*Hard Drive*
LHC	*Large Hadron Colider*
LRU	*Least Recently Used*
LSST	*Large Synoptic Survey Telescope*
NoSQL	*Not Only SQL*
NVRAM	*Non Volatile Random Access Memory*
SGBD	Sistema de Gerenciamento de Bancos de Dados

LISTA DE SÍMBOLOS

Q	Sequência de consultas que caracteriza a carga de trabalho
l_j	Extremo inferior do intervalo de chave a ser predito
h_j	Extremo superior do intervalo de chave a ser predito
ϕ	Dependência funcional de l_j com j
ψ	Dependência funcional de h_j com j
x_i^k	Saída do i-ésimo neurônio da camada k
σ	Função sigmóide usada como ativação dos neurônios
u_{ji}^k	Peso da ligação entre o i-ésimo neurônio da camada k e o j-ésimo neurônio da camada $k+1$
$\mathbf{W_1}$	Matriz dos pesos das ligações entre os neurônios da camada de entrada e os neurônios da camada oculta
$\mathbf{W_2}$	Matriz dos pesos das ligações entre os neurônios da camada oculta e os neurônios da camada de saída
$\mathbf{X}$	Vetor dos atributos das amostras
$\mathbf{Y}$	Vetor dos saídas associadas com as amostras
H	Matriz de de saída da camada k
M_k	Matriz de covariância das saídas da camada k
C	Número de buscas a serem efetuadas em sequência
R_1	Acrécimo aleatório adicionado ao limite inferior do intervalo a ser requisitado
R_2	Acrécimo aleatório adicionado ao limite superior do intervalo a ser requisitado

SUMÁRIO

1 INTRODUÇÃO

As atividades humanas são caracterizadas pela dependência da capacidade de registrar informações sobre os acontecimentos, descobertas, negociações e relações entre as pessoas e entre elas e seu mundo. Isto é especialmente verdadeiro no período atual, onde todas as relações de trabalho e intercâmbio de conhecimento se dão através de transações eletrônicas. O armazenamento de dados e o acesso a eles se tornaram complexos o suficiente para motivar o surgimento de sistemas especializados responsáveis por gerenciar este recurso. Tais sistemas são conhecidos coletivamente como Sistema de Gerenciamento de Bancos de Dados (SGBD).

SGBDs tornam transparente a maneira como os dados são armazenados e manipulados, liberando as aplicações das tarefas relacionadas ao gerenciamento desses dados. O formato dos arquivos de dados, as características dos dispositivos de armazenamento e a presença de acesso concorrente por outros agentes são exemplos de problemas que as aplicações não precisam tratar quando fazem acesso a dados através da interface de um SGBD.

As aplicações que acessam os bancos de dados historicamente são marcadas por procedimentos bem definidos e padrões de acesso conhecidos. Os exemplos clássicos incluem sistemas de controle de pessoal, de estoque e de transações financeiras. Uma característica comum dessas aplicações é o fato de que há total conhecimento, por parte dos desenvolvedores da aplicação, sobre o formato dos dados manipulados por ela e substancial conhecimento sobre a maneira como estes dados serão acessados.

Importantes aplicações surgidas recentemente não se enquadram neste perfil. Aquelas que acessam bancos de dados gerados por equipamentos científicos são um exemplo, tais bancos são continuamente alimentados com dados novos compostos de atributos numéricos. Os pesquisadores interessados em extrair informações dos dados por sua vez não sabem quais consultas serão disparadas, isto é, qual será o padrão de acesso. Isto se dá por que é justamente o processo de exploração dos dados que irá ditar o que será de interesse a seguir. Com isto, o planejamento do esquema e das estruturas de dados a serem utilizadas no banco de dados não tem em quê basear-se, mesmo sendo esta uma parte vital da implantação do sistema.

Uma abordagem que vem se mostrando promissora é a de abandonar a construção de estruturas de acesso completas como um passo anterior à disponibilização do sistema para atender às consultas disparadas pelas aplicações. As estruturas de dados são então modificadas por meio de ações incrementais que ocorrem junto com o processamento de cada consulta, e esta abordagem possibilita que estas ações de transformação do nível físico do banco de

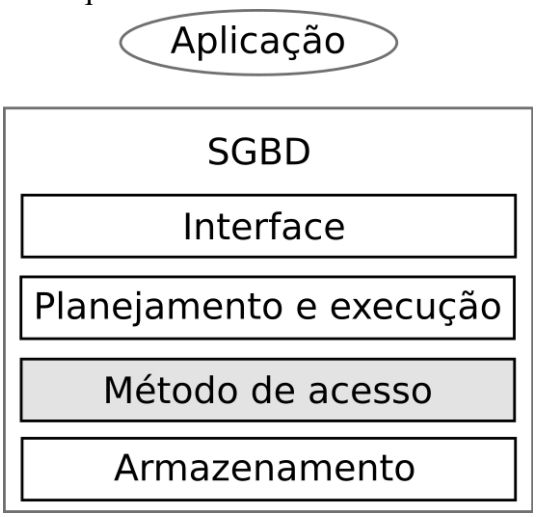
dados ocorram de forma orientada à carga de trabalho corrente. Com isto, esforço da ação de indexar é focado nas regiões dos dados que de fato são de interesse para as aplicações. Este princípio guia as estratégias de indexação adaptativa a serem descritas adiante e também inspira a contribuição apresentada neste trabalho, que aplica aprendizagem automática para aproveitar melhor a informação fornecida pela carga de trabalho já processada na construção do método de acesso.

1.1 Arquitetura

A arquitetura usual dos SGBDs é composta de diversos módulos, cada um responsável por uma etapa do processo de acesso aos dados armazenados. Considerando SGBDs que armazenam os dados em dispositivos não voláteis, como discos magnéticos e de estado sólido, os módulos mais internos são o gerenciador de arquivos e o gerenciador de memória ou *Buffer*. Estes módulos são também chamados de camadas mais baixas do SGBD e manuseiam diretamente os itens de dados, que daqui por diante serão chamados de registros. Nestes sistemas, os registros originalmente residem no armazenamento não volátil ou *Hard Drive* (HD), geralmente agrupados em páginas ou blocos para compor coleções identificáveis, chamadas comumente de arquivos. Quando registros são requisitados pelo processamento de alguma consulta, são copiados do HD para a memória principal, ou *Dynamic Random Access Memory* (DRAM), onde podem ser modificados. Posteriormente as versões modificadas dos registros são novamente escritas no HD, única localização considerada segura devido à não volatilidade. As transferências de dados entre HD e DRAM são objeto de cuidadoso planejamento por parte do SGBD pois são uma das operações mais custosas de seu fluxo de trabalho.

Módulos mais altos são responsáveis pela comunicação com os clientes. Esta comunicação inclui autenticação para garantir que cada usuário autorizado tenha acesso aos dados de seu interesse, e apenas a estes. Cada requisição ao SGBD é traduzida da linguagem de manipulação de dados ou *Data Manipulation Language* (DML), geralmente declarativa, para um plano de execução. O plano de execução, diferente da DML, não é declarativo, contendo as instruções procedurais a serem executadas pelos níveis inferiores. O plano é então otimizado com o intuito de incluir as ações menos custosas que se puder encontrar para produzir o resultado desejado pela aplicação. Controle de concorrência também é importante por causa da possível existência de múltiplos usuários acessando as estruturas de dados. Os módulos transacionais se encaixam nessa arquitetura de camadas como um nível intermediário, entre o acesso aos dados

Figura 1 – Arquitetura de alto nível de um SGBD



no armazenamento mais abaixo e a lógica de conexão com as aplicações acima.

A figura 1 ilustra esta arquitetura de níveis, onde o fluxo de controle parte da camada mais próxima da aplicação e volta até ela depois de ter possivelmente descido até o nível de acesso aos dispositivos de armazenamento. Isto pode não acontecer por alguns motivos, como por exemplo o otimizador detectar que a consulta contém condições impossíveis ou o usuário não ter permissão de acessar os registros requisitados, nestes casos uma resposta é retornada sem a necessidade de se atingir o nível de armazenamento. O nível de métodos de acesso, destacado na figura, compreende a interface que abstrai o armazenamento dos níveis acima e será discutido com mais detalhes na sessão 1.2 por ser de especial interesse para este trabalho.

Sistemas de bancos relacionais geralmente armazenam seus dados registro a registro, isto é, cada registro é mantido contiguamente, com cada atributo no endereço imediatamente após o anterior. A exceção são os bancos que utilizam armazenamento colunar ou decomposicional (STONEBRAKER *et al.*, 2005), onde cada atributo é armazenado contiguamente, e assim, para se recuperar um registro é necessário reconstituí-lo a partir dos valores buscados em cada uma das colunas. Os bancos colunares fazem parte do conjunto de sistemas conhecidos coletivamente como *Not Only SQL* (NoSQL)[1], mas frequentemente apresentam-se às aplicações através de uma interface SQL, o que faz com que se pareçam com os bancos relacionais do ponto de vista de interface de programação. Exemplos de SGBDs colunares são o Cassandra[2] e o MonetDB[3] (IDREOS *et al.*, 2012).

[1] Existe controvérsia sobre a origem e o significado, podendo ser *Not only SQL* ou uma forma mais forte: *No to SQL*

[2] http://cassandra.apache.org/

[3] https://www.monetdb.org/

No entanto, devido à diferença no armazenamento, bancos colunares e relacionais orientados a tuplas apresentam padrões de performance distintos para cargas de trabalho dominadas por buscas e atualizações pontuais e para cargas mais orientadas a consultas analíticas. O motivo desta assimetria é o fato de, nos bancos colunares, ser possível movimentar mais valores de um atributo entre os dispositivos de memória a cada transferência. Considere-se, por exemplo, uma consulta que solicita a soma dos valores de um dos atributos dos registros de uma relação. Em um armazenamento orientado a colunas todos estes valores são armazenados contiguamente e logo todas as páginas movidas estarão povoadas apenas com valores pertinentes.

Técnicas baseadas em armazenamento colunar têm recebido substancial interesse como alternativa de implementação para bancos de dados (PSAROUDAKIS *et al.*, 2016) (KASTRATI; MOERKOTTE, 2017). O foco destes sistemas é atender aplicações que buscam informações estatísticas e exploratórias nos dados, por serem mais eficientes para varreduras de atributo único. Em especial, algumas das primeiras estratégias de indexação adaptativa nasceram no contexto de bancos colunares em memória, em resposta às necessidades de aplicações que fazem análise de grandes volumes de dados em tempo real, como as apresentadas como exemplo motivador da sessão 1.3.

1.2 Métodos de acesso

Considerando ainda a arquitetura dividida em níveis, aqueles de maior interesse para este trabalho são os mais baixos, que se ocupam da organização dos dados no meio de armazenamento físico e de prover mecanismos de busca por registros específicos. Na implementação dos SGBDs, estes níveis são conhecidos pela interface que expõe às camadas acima, chamada de **Método de Acesso**. Com esta abstração, as camadas superiores muitas vezes não precisam implementar algoritmos específicos para as estruturas de dados onde os registros residem. Outro benefício da ideia de métodos de acesso é que o sistema se torna extensível podendo-se adicionar métodos de acesso novos com mudanças mínimas no código das outras camadas.

O mais simples método de acesso são os arquivos **heap**. Este método de acesso armazena os registros em páginas, geralmente de $4KB$ ou $8KB$, e as dispõe em lista ou criam uma estrutura de diretório para elas. Assim, arquivos heap são coleções não ordenadas e para acessar determinado registro é necessário testar cada um deles até encontrar o desejado. Portanto a complexidade das buscas é $O(n)$. A disposição de registros em grupos (páginas) é interessante

porque reflete a forma como as transferências entre a memória principal e os dispositivos de armazenamento secundário acontecem. Isto é, as páginas correspondem aos blocos, ou a um conjunto de blocos no HD, os quais as controladoras dos dispositivos de armazenamento podem garantir transferências atômicas.

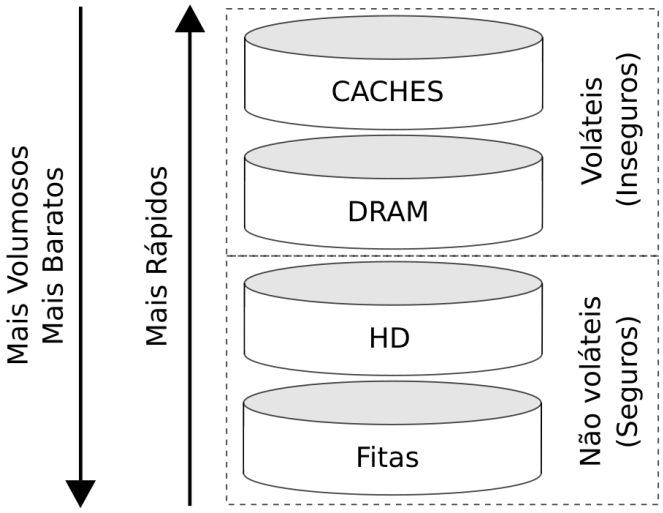
Tabelas hash também são ocasionalmente usadas como métodos de acesso por possibilitarem buscas com complexidade $O(1)$, sendo úteis para aplicações cujos acessos majoritariamente buscam por um registro específico usando como critério de busca, ou predicado da consulta, um teste por igualdade na chave usada pelo índice. Este método de acesso não é conveniente para buscas por intervalos da mesma forma que o são para buscas pontuais, isto é, uma tabela hash não auxilia uma consulta cujo predicado envolva desigualdades. Nos métodos de acesso baseados em hash o armazenamento paginado é mapeado para os *buckets* da tabela hash, sendo um *bucket* igual em tamanho a algum número fixo de páginas.

Outra classe de método de acesso particularmente popular é a dos armazenadores baseados em árvores B ou B+. A diferença entre as duas é que nas árvores B todos os nós são equivalentes, contendo registros de dados completos. Já as árvores B+ têm apenas as chaves de busca nos níveis não folha, os registros ficam todos no nível folha e os nós deste nível são ligados numa estrutura de lista encadeada, podendo então ser percorridos sem passar pelos níveis acima. A importância destas estruturas, em especial da árvore B+, é tal que muitos SGBDs relacionais a utilizam como método de acesso padrão, isto é, a utilizam caso o usuário não especifique outro. A eficiência no acesso é uma motivação para estas estruturas, outra é o fato de que operações sistemáticas com suporte a concorrência são conhecidas (LEHMAN; YAO, 1981).

Árvores B+ também organizam os registros em páginas, mas as páginas apontam umas para as outras em uma estrutura de árvore n-ária ordenada, tendo assim um comportamento logarítmico, $O(\log n)$, para buscas. A construção de uma árvore B+ requer a ordenação dos registros de dados, por isso é uma operação cuja complexidade é tipicamente $O(n \log n)$. A fim de construir tal estrutura sobre uma massa de dados maior que a memória principal, é necessário utilizar um algoritmo de ordenação externa. Em tais algoritmos, os dados são ordenados por partes, sendo cada parte transferida para a memória, ordenada e escrita de volta no armazenamento não volátil. Posteriormente os resultados destas ordenações parciais passam por um processo de *merge sort* para se tornar um todo globalmente ordenado, podendo ser lidos do HD e escritos de volta múltiplas vezes durante o processo.

1.2.1 Hierarquia de memória

Figura 2 – Hierarquia de dispositivos de armazenamento



Esta breve discussão das características das árvores B+ deixa evidente um aspecto adicional da elaboração de métodos de acesso: A necessidade de planejar as estruturas de dados e algoritmos de acordo com as características dos dispositivos de armazenamento presentes e quais transferências acontecerão em cada operação. Estas análises não são triviais pelo fato dos dispositivos apresentarem características bastante distintas entre si. A figura 2 mostra esquematicamente as principais características destes dispositivos, e representa a visão detalhada da camada de armazenamento presente na figura 1.

Partindo do topo, ou do ponto de vista do processador, os dados sobre os quais se pode operar de imediato são aqueles que residem nos caches. Os processadores modernos podem ter vários níveis de cache, conhecidos como L1, L2, etc. Estes são os dispositivos mais rápidos da hierarquia, pois se encontram junto do processador, porém a capacidade de armazenamento dos caches é severamente limitada. Os caches suportam poucos Megabytes, ou algumas centenas de registros típicos de um SGBD. Os processadores Intel® i7, por exemplo, têm caches L1 suportando 32KB até L3 suportando 2MB.

Devido ao pouco espaço, nenhum processo deve assumir que seus dados estejam nos caches, pois estes têm como propósito manter dados mais frequentemente ou mais recentemente acessados em uma localização de acesso mais rápido assumindo que eles podem ser necessários

novamente em breve. Quando novos dados são necessários ao processador e não se encontram nos caches, o próprio hardware do processador se encarrega de trocar alguns dos dados presentes no cache pelos novos dados de interesse com o próximo nível da hierarquia, a DRAM.

A DRAM, chamada por vezes memória principal, é o meio onde a maior parte dos algoritmos considera que seus dados residem, os modelos de programação são geralmente pensados para este nível e deixam as trocas com os caches a cargo da política interna do processador. A otimização mais frequente que considera os caches é fazer o tamanho dos nós das estruturas de dados ser igual a um múltiplo da unidade de transferência entre os dois níveis, para que o processador não perca tempo lendo estruturas incompletas e assim precisando fazer mais transferências parciais. A característica principal compartilhada entre os caches e as memórias é o fato de serem voláteis, isto é, não são capazes de manter os dados que armazenam caso o fornecimento de energia seja cortado. Por isto SGBDs precisam manter um registro (ou *log*) de todas as atualizações que realizam nos dados durante a sua operação, em um dispositivo não volátil, a fim de ser capaz de recuperar seu estado até um momento passado no caso de alguma falha de energia ou de dispositivos.

Uma das motivações do uso de outros dispositivos de armazenamento é o elevado preço por unidade de capacidade dos dispositivos voláteis citados acima. Embora este fato venha sendo rapidamente atenuado nos últimos anos pela queda exponencial no preço dos dispositivos DRAM, o fato de serem voláteis ainda faz imperativo o armazenamento dos dados nas camadas ainda mais baixas. Neste ponto se encontram os HDs, que compreendem as unidades de disco rígido e de estado sólido. Estes dispositivos trocam dados com a DRAM e têm como características serem ordens de magnitude mais lentos que ela, e também serem não voláteis. Com efeito, esta é a localização onde os SGBDs podem considerar os dados seguros, pois mesmo após uma falta de energia ou reinicialização da máquina onde o sistema executa, os dados ainda serão persistentes.

Os dispositivos não voláteis também diferem quanto à maneira como se acessam os dados que contém. Na DRAM é possível acessar com o mesmo custo qualquer byte e em qualquer ordem, já no caso de discos e SSDs por exemplo, todas as transferências de dados entre estes e a DRAM são paginadas, isto é, se dão em unidades de tamanho fixo com tamanho típico de 4KB, o que motiva o tamanho das páginas de dados utilizadas nos SGBDs. A ordem com a qual se acessam as páginas também importa em discos, sendo o acesso sequencial mais rápido que o acesso aleatório. Há ainda outros dispositivos em uso, como as fitas magnéticas,

cuja operação majoritariamente mecânica é ainda mais lenta que a dos HDs e a forma de acesso também é sequencial.

A possível presença de todos estes dispositivos de armazenamento faz as considerações tradicionais sobre o fluxo de dados em SGBDs, que consideram apenas a mecânica das trocas entre arquivos e um gerenciador de *buffer*, não acurada. Um modelo mais realista precisa estar ciente de que o armazenamento se compõe de um número variável de unidades com diferentes capacidades, diferentes velocidades de acesso e diferentes graus de volatilidade, e sendo alguns possivelmente remotos e ligados por rede. Com isto em mente, é possível se melhorar até algumas das mais básicas atividades de um SGBD, como ordenação de dados por exemplo (GRAEFE, 2011).

1.2.2 Outros modelos de armazenamento

Recentemente, se observa um crescente interesse em bancos de dados em memória, que são aqueles que consideram a DRAM como a principal residência dos registros. Este é um movimento natural dado que os preços atuais das memórias são tais que é perfeitamente viável hospedar a maioria dos bancos destinados a processamento de transações inteiramente em memória. Do ponto de vista acadêmico no entanto, isto levanta diversas questões relacionadas à volatilidade do armazenamento e com a escolha dos métodos de acesso mais adequados a este tipo de sistema (FAERBER *et al.*, 2017). A arquitetura dos bancos em memória apresenta basicamente os mesmos módulos mostrados na figura 1, e com as mesmas funções, mas a maneira de implementá-los segue paradigmas distintos.

A principal diferença é que SGBDs em memória não possuem um gerenciador de *buffer*, já que não é necessário trocar páginas de dados com um dispositivo mais lento onde os dados residem. Isto significa não só que o acesso é mais rápido, mas também que as estratégias de transferência devem ser distintas, uma vez que o endereçamento de memória se dá a nível de byte, e não de blocos. Também não há vantagens em se priorizar acessos sequenciais na DRAM. Além disto, gerenciar bloqueios e manter balanceada uma árvore B+ são operações que desperdiçam parte do ganho de desempenho obtido pelo armazenamento em memória, por isto outras classes de métodos de acesso tem sido exploradas nestes sistemas. Exemplos são as estruturas de dados probabilísticas, tais como *Skip Lists* (XIE *et al.*, 2017) (BENDER *et al.*, 2017), e *Bloom Filters* que são usadas onde possível para evitar I/O.

Contudo, a evolução das tecnologias usadas nos hardwares que compõe a hierarquia

de memória tem trazido mais uma assimetria nos tempos de acesso, particularmente no que se refere a transferências entre a memória e os caches do processador. Estas transferências compartilham similaridades com aquelas entre o disco e a memória: são paginadas e um meio é mais rápido que o outro. A analogia no entanto não é completa pois, embora a memória seja mais volumosa e mais lenta, é volátil e de acesso aleatório. Mesmo o cenário não sendo equivalente ao dos discos, as transferências paginadas renovam o interesse em estruturas baseadas em árvores B+, pois com elas transferências entre os níveis de memória e cache podem ser arranjadas de tal forma que um nó da árvore em memória tenha o mesmo tamanho que a unidade de transferência. Isto faz com que cada transferência mova um bloco de dados de interesse por vez, ao invés de possivelmente mover toda uma página por apenas um pequeno subconjunto de dados de interesse.

Outra classe de hardwares fazem a analogia entre as trocas disco-memória e memória-cache ainda mais próximas. Trata-se das máquinas com suporte a transferências com garantias transacionais entre os níveis de cache de CPU e DRAM e de dispositivos de armazenamento semelhantes à memoria mas persistentes, ou *Non Volatile Random Access Memory* (NVRAM), além de interfaces de programação para expor estas funcionalidades[4]. Estas novas tecnologias tendem a tornar as transferências entre HD e DRAM e aquelas entre DRAM e caches ainda mais análogas. Um interessante retorno no que diz respeito a projeto de métodos de acesso.

Um bom exemplo de implementação de métodos de acesso se encontra no código do PostgreSQL[5], que é aberto. Neste repositório, uma interface para métodos de acesso composta de funções C é declarada no arquivo `src/include/access/amapi.h`. A partir desta interface, classes de métodos de acesso são definidas, e incluem: *Generalized Search Tree* (GiST), baseados em árvores de busca incluindo B+, *Generalized Inverted Index* (GIN), que contemplam índices invertidos úteis a buscas textuais, e *Hash Access Method* (HASH), baseados em tabelas de dispersão. Todas estas interfaces são declaradas em cabeçalhos que se encontram no diretório `src/include/access`.

Três fatores são especialmente importantes no projeto de métodos de acesso:

1. a eficiência no acesso aos registros que ele proporciona;
2. o espaço extra de armazenamento de que necessita e
3. o custo de atualizar a estrutura (adicionar, remover ou alterar registros).

Se observa nos métodos de acesso existentes que há um compromisso entre estas três caracterís-

[4] http://pmem.io
[5] https://www.postgresql.org/ftp/source/

ticas, expresso na conjectura RUM (ATHANASSOULIS; IDREOS, 2016) (ATHANASSOULIS *et al.*, 2016), do inglês *Read, Update, Memory*. Esta conjectura, de maneira resumida, diz que não se consegue construir um método de acesso que otimize os três aspectos acima citados simultaneamente. O que acontece nos métodos de acesso conhecidos é que cada um deles é bom em até dois desses fatores mas apresenta custos adicionais em pelo menos um deles. A situação é semelhante à que se tem no teorema CAP do contexto de sistemas distribuídos (GILBERT; LYNCH, 2002). Portanto, é também uma escolha que precisa ser feita ao se planejar métodos de acesso: otimizar para leitura e escrita a custo de ocupar muito armazenamento, possivelmente com redundância, ou priorizar armazenamento compacto e leitura eficiente a custo de atualizações demoradas.

1.2.3 Métodos de acesso e índices

Por fim, resta considerar as dependências entre métodos de acesso e índices. Os últimos são elementos do esquema do banco de dados construídos por ordem do usuário, e têm como objetivo acelerar e também regular o acesso a um subconjunto dos atributos de uma relação. Assim, índices e métodos de acesso são conceitos distintos mas relacionados, pois a criação de índices impacta nas escolhas de métodos de acesso para os dados. Considere-se por exemplo uma relação armazenada segundo um método de acesso por tabela hash, neste caso o índice criado sobre a chave da tabela hash só pode auxiliar em consultas cujo predicado tem uma igualdade para a chave. Um índice com esta limitação só é criado se o usuário assim especificar e é chamado de *índice hash*. Naturalmente, a declaração de um índice hash resulta na construção de um método de acesso por tabela hash, mas nem sempre a correspondência de um tipo de índice para um método de acesso particular é única. Essas considerações mostram a relação entre índices e métodos de acesso: a escolha de um não determina mas limita as opções ou fornece indicações para o outro.

Nas sessões a seguir os termos índice e método de acesso serão ocasionalmente usados de forma indiferente, isto se dá pelo fato de alguns trabalhos relacionados utilizarem termos como índice adaptativo ou indexação adaptativa para se referir a estratégias de construção de estruturas de dados, que como tal, se aplicam no nível de métodos de acesso, e não necessariamente implicam a adição de índices no nível de esquema, se houver. Seguimos esta convenção ao usar a expressão “construção de índice” para a construção das estruturas de dados no nível de armazenamento.

1.3 Exploração de dados massivos

Exploração de dados é o conjunto de práticas realizadas com o intuito de obter uma visão geral da informação contida num conjunto de dados sem conhecimento prévio sobre seu conteúdo, frequentemente praticada com o intuito de guiar processos de decisão ou de planejamento de análises mais específicas (IDREOS, 2013) (IDREOS *et al.*, 2015). Em geral, ferramentas de visualização são utilizadas (KEIM, 2014) (VARTAK *et al.*, 2015) e valores estatísticos são coletados de amostras do banco como primeiro passo. As informações preliminares extraídas durante esta fase guiam os passos seguintes da análise dos dados, e ainda mais importante, no contexto de projeto de métodos de acesso, o padrão de acesso aos registros.

Novas formas de interagir com os dados têm aparecido, motivadas pelas características variadas dos conjuntos de dados disponíveis e também pelas novas tecnologias de interface de usuário, tais como terminais baseados em toque (NANDI, 2013) (LIAROU; IDREOS, 2014). Estes esforços refletem o fato de que, ao explorar dados novos, os pesquisadores e cientistas de dados estão interessados em se familiarizar com características gerais destes dados mas, justamente pela falta de conhecimento dos dados, não têm como definir as consultas antes de iniciarem o processo. Isto significa que cada consulta ocorrida durante o processo de exploração influencia na decisão do que consultar em seguida.

1.3.1 Volume e velocidade

O desafio atual vem do volume e da velocidade dos dados que se deseja explorar frequentemente no contexto científico e da análise necessária aos negócios baseados no comportamento virtual dos usuários. O volume implica que até mesmo informações estatísticas simples como as médias e desvios padrão dos atributos dos dados podem requerer longos tempos para serem recuperados com exatidão, por ser necessário o acesso a todos os registros do conjunto de dados. Um meio de contornar este obstáculo, nos casos em que respostas não exatas ou incompletas são aceitáveis, é extrair estas estatísticas de um subconjunto dos registros de dados, isto é, por amostragem.

A velocidade diz respeito à taxa temporal com que novos dados são produzidos, o que por sua vez torna desatualizados os dados que já se possuía. É necessária a capacidade de analisar os dados pelo menos à mesma velocidade em que eles são produzidos. Caso contrário, novos dados irão empilhar-se e perder valor ao se tornarem desatualizados, e os resultados produzidos

ao final da análise também serão de menor valor pois representam padrões e conclusões que se aplicam ao passado. Este passado pode significar apenas o dia anterior ou a semana anterior, períodos relevantes para algumas aplicações. É esta necessidade por velocidade de tempo real que torna o modelo de *Data Warehousing* impróprio para as aplicações de exploração, pois esta solução requer tempo para a implantação de um banco de dados analítico e para o processo de *Tuning*, onde um esquema com diversos índices é criado para auxiliar as consultas.

Como exemplo concreto de aplicações com necessidades de gerenciamento de dados para análise tão estritas, considere o *Large Synoptic Survey Telescope* (LSST). Um telescópio ainda em construção cujo objetivo é fotografar todo o céu noturno repetidamente a fim de possibilitar o estudo de fenômenos transitórios ou que mudam no decorrer de poucas noites. O LSST contará com uma câmera digital de 3.2 Gigapixel, a maior já construída, e operará diariamente. De acordo com a página do projeto na internet [6]:

> Software is one of the most challenging aspects of LSST, as more than 30 terabytes of data must be processed and stored each night in producing the largest non-proprietary data set in the world.

Os dados gerados por este projeto, que serão abertos à toda a comunidade científica, apresentam os desafios de volume e velocidade supramencionados. Diversos pesquisadores interessados em extrair informações dessa massa de dados certamente o abordarão através de métodos exploratórios uma vez que eles são adequados à analise coletiva dos objetos observados, e não para estudo de objetos individuais. A busca por objetos desconhecidos, como asteroides e cometas, e por mudanças rápidas não preditas serão aplicações típicas, e para elas não é possível saber a priori quais partes do dado serão de maior interesse. Aplicações deste tipo já não são raras, outros equipamentos científicos, como o *Large Hadron Colider* (LHC)[7] geram dados igualmente massivos em poucos dias de funcionamento e o tamanho destes dados é sozinho um significativo desafio para os pesquisadores que os utilizam.

Tais projetos armazenam os dados produzidos pelos instrumentos em fitas magnéticas, por serem muito resilientes e baratas (preço por Gigabyte). Este detalhe é importante do ponto de vista de acesso pois fitas são os dispositivos de armazenamento mais lentos, uma vez que para acessar um registro em fita é necessário trazer a fita do depósito até a leitora, procedimento que pode ser feito manualmente ou por braço mecânico. Depois disso, a leitora precisa girar a

[6] https://www.lsst.org/about
[7] https://home.cern/topics/large-hadron-collider

bobina onde a fita está enrolada até a posição do registro e só então lê-lo, portanto um processo essencialmente mecânico e sequencial.

1.3.2 Construção de métodos de acesso para exploração

O elevado custo da falta de estruturas de acesso rápido aos registros, estejam eles em um arquivo heap de um único disco ou em um conjunto de fitas é uma manifestação do mesmo problema geral: A falta de conhecimento prévio sobre os padrões de acesso ou carga de trabalho. Construir estruturas de dados que englobem todos os registros de conjuntos de dados como estes não é uma alternativa interessante pois o processo pode ser mais demorado do que os ciclos de produção de dados novos, não acompanhando sua velocidade e magnificando o problema da desatualização ou decaimento de dados (KERSTEN; SIDIROURGOS, 2017). Além disto, a espera pela construção de métodos de acesso é um período onde a análise, o real processo de interesse, tem que simplesmente esperar sem fazer progresso. A dinamicidade dos dados também provoca a necessidade de ações de manutenção das estruturas ao se inserir registros novos.

Há ainda mais um complicador, pois para se construir o método de acesso como um passo prévio, é necessário escolher a chave de busca, ou conjunto de atributos sobre os quais as consultas serão otimizadas. Em alto nível este problema é chamado de seleção de índice, um problema bem tratado para os sistemas que atendiam às aplicações clássicas (GUPTA *et al.*, 1997) (SCHNAITTER; POLYZOTIS, 2009) (CHAUDHURI; NARASAYYA, 1997), e novamente é dependente de uma expectativa referente à carga de trabalho. Se a seleção não for bem sucedida, isto é, se o método de acesso utilizar uma chave diferente daquela sobre a qual as consultas baseiam seus predicados, então o tempo de construção foi perdido, pois o método de acesso não vai proporcionar nenhum ganho de performance. E pior ainda, pode haver perdas, pois se o banco de dados for dinâmico, havendo modificações no conjunto de registros, o sistema precisará também manter a estrutura consistente, um custo extra.

Aplicações com necessidades nestas escalas não se encontram apenas no contexto científico. Um bom exemplo tem sido o modelo de negócios particularmente bem sucedido de fornecer serviços de forma terceirizada através de aplicativos que conectam fornecedores e consumidores. Muitos ramos de serviços, tais como transporte individual e hotelaria, cujas companhias pioneiras foram respectivamente Uber[8] e Airbnb[9], têm sido revolucionados com

[8] www.uber.com
[9] www.airbnb.com

este modelo. O sucesso de tais negócios depende de serem capazes de oferecer experiência personalizada e rapidamente adaptável a novos padrões de uso para cada um dos usuários, que por sua vez são numerosos, diversos e dinâmicos.

O caráter individualizado destes serviços exige que eles coletem os dados de uso dos indivíduos e os analisem de forma mais próxima de tempo real quanto possível. Novamente, o modelo OLAP clássico deixa de ser o mais indicado pela rapidez e pela mutabilidade da aplicação. Estas aplicações têm características em comum com as aplicações científicas: grandes volumes de dados, dado semi estruturado, necessidades de coletar informação estatística rapidamente, e padrão de acesso imprevisível e mutável. Assim, para estes dois cenários, métodos de acesso com características semelhantes são necessários.

É necessário notar a diferença entre exploração de grandes volumes de dados e as tarefas de *processamento* de grandes volumes de dados. Estas últimas têm sido resolvidas principalmente com o uso de *frameworks* de processamento distribuído como o *Map Reduce*, que funcionam com os dados armazenados em sistemas de arquivos distribuídos, em geral segundo modelos pouco estruturados. A diferença, do ponto de vista de métodos de acesso, é que, no contexto de processamento, a carga de trabalho é fixa e conhecida. Isto por que já se sabe como processar os dados. Já em exploração, o padrão de acesso não é conhecido a priori, e é orientado a processamento de consultas. Logo os sistemas de arquivo distribuídos associados às técnicas de processamento distribuído não são possíveis soluções satisfatórias para o gerenciamento de dados para aplicações exploratórias.

O uso de armazenamento distribuído para exploração apresenta ainda um problema adicional que diz respeito à localidade dos dados. Pois ao processar uma consulta, os registros desejados podem não só estar fora da estrutura de dados principal ou do dispositivo mais rápido, mas podem estar em uma máquina remota arbitrária, tornando o acesso ordens de magnitude mais custoso, além de adicionar as considerações sobre latência de rede e outros custos de transmissão. Armazenamento distribuído e os problemas associados não são abordados neste trabalho.

1.4 Contribuições e estrutura do texto

A hipótese defendida por este trabalho é a de que a sequência das buscas efetuadas mais recentemente fornece conhecimento do padrão de acesso, útil para adiantar etapas da construção das estruturas de dados e transferências de páginas para localizações mais rapidamente

acessíveis. Um mecanismo de regressão é utilizado para aprender os intervalos de chave a serem indexados a partir das buscas e o resultado é avaliado através da localidade dos registros (*cache hit*) e dos tempos de resposta. Esta estratégia vai além dos índices adaptativos, que funcionam de forma reativa e baseada na consulta atual isoladamente. Comportamento preditivo é então adicionado com base em um modelo do *padrão* atual da carga de trabalho, além de atualizar continuamente este modelo com cada nova consulta processada.

Como consequência da elaboração e da avaliação desta hipótese os artefatos listados na tabela 1 foram publicados e expostos à apreciação da comunidade.

Tabela 1 – Publicações

Título	Veículo
MetisIDX: From Adaptive to Predictive Data Indexing	EDBT 2018
Adaptive Database Kernels	SBBD 2017
Thesis and Dissertations Workshop	SBBD 2017

A estrutura do texto a seguir constitui-se da seguinte forma: O capítulo 2 descreve com certo nível de detalhe as principais técnicas de indexação adaptativas que precedem e inspiram as contribuições apresentadas. O capítulo 3 descreve a abordagem de proposta de construir a estrutura de acesso em paralelo com as buscas e guiando-se pelo modelo da carga. O capítulo 4 mostra os resultados de uma avaliação empírica da nossa implementação da técnica, bem como uma comparação de desempenho com a técnica de indexação adaptativa estado da arte com características mais próximas. O capítulo 5 sumariza as contribuições apresentadas, conclui e contextualiza a discussão dos resultados dos experimentos e aponta direções para novos esforços de pesquisa de interesse para a área.

2 ÍNDICES ADAPTATIVOS

Os sistemas clássicos contavam com a premissa de que há conhecimento sobre os dados e, consequentemente, sobre a maneira como eles serão acessados. Assim, assumiam que as estruturas de acesso mais adequadas podiam ser escolhidas previamente. Estas premissas já não se verificam para um número crescente de aplicações, pelo fato de que agora os dados são imprevisíveis e coletados em grande quantidade e de forma automática. Adicionalmente, as aplicações analíticas atuais são cada vez mais voltadas à atividade de buscar por "padrões interessantes", mesmo sem saber que padrões são esses e, menos ainda onde se encontram. Por isso, é necessário se repensar os métodos de acesso e seus processos de construção, para poderem operar também de forma orientada a aprender com as demandas correntes.

2.0.1 Índices completos, parciais e online

Os processos utilizados para se construir estruturas de dados nestes sistemas tradicionais são executados em um único passo, isto é, todos os registros disponíveis são arranjados na estrutura e então ela é considerada concluída. Após esta fase, operações de modificação dessas estruturas ocorrem apenas quando o conjunto dos registros muda, ou seja, a estrutura sofre manutenção no caso de inserções, remoções ou atualizações de registros. Tais processos de criação e destruição de métodos de acesso ocorrem, por exemplo, quando um usuário dispara comandos `CREATE INDEX` via SQL. Índices com estas características: criados em um único passo, por comando explícito do administrador e cobrindo todos os registros, são chamados de *índices offline*.

Reconhecendo que há subconjuntos dos dados que são consultados com maior frequência e outros que são raramente acessados, ou mesmo nunca acessados, *índices parciais* (SESHADRI; SWAMI, 1995) foram propostos. Estes índices incluem apenas parte dos registros de dados da relação e têm as vantagens de ter custo de manutenção mais baixo e ocupar menos espaço. As implementações de índices parciais dos sistemas atuais, no entanto, filtram os registros a serem indexados de acordo com uma condição fornecida pelo usuário e com isso mantêm o requisito de conhecimento prévio do padrão de acesso e sua criação também se dá em uma única operação ordenada pelo administrador ou por uma ferramenta de monitoramento (BRUNO; CHAUDHURI, 2007). Estas ferramentas materializam a estratégia conhecida como auto-tuning, onde a performance do sistema (em geral medida pelo throughput), a configuração

da plataforma (sistema operacional e hardware) e o perfil da carga de trabalho são monitorados por um processo externo ao SGBD que periodicamente dispara comandos ao sistema, tais como criar ou destruir índices. Índices gerenciados desta forma ganham a denominação de *Índices online*.

Os índices adaptativos são estratégias criadas especificamente para o cenário de exploração de dados, e têm características em comum com índices parciais e índices online. Assim como os índices online, os índices adaptativos são criados em decorrência do padrão de acesso e independentemente de comandos do administrador, eles também são índices parciais, pois cobrem apenas um subconjunto dos registros. A diferença em relação aos índices online ou auto-tuning, onde a criação e o descarte de estruturas são atômicos e disparados por monitoramento periódico, é o fato de que índices adaptativos são modificados continuamente enquanto o sistema processa as consultas. Isto é, a sua construção se dá de maneira incremental e sempre guiada pela carga de trabalho atual. Além disto, as ações de construção ou modificação das estruturas de dados e as ações de busca por um conjunto de registros são a mesma, o mesmo algoritmo faz as duas tarefas. O índice se torna um efeito colateral do processamento de consultas.

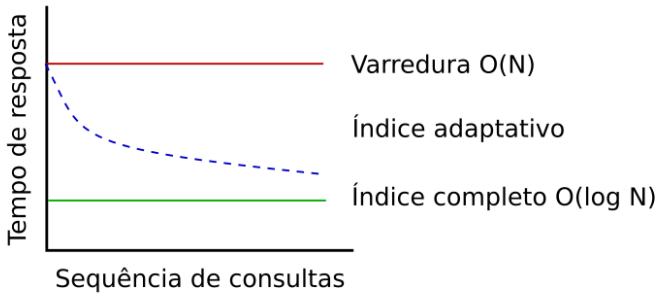
Índices completos podem levar muito tempo para serem construídos, dadas as exigências de velocidade da exploração de dados uma espera da ordem de várias horas para se poder disparar a primeira consulta não é tolerável, e pode ser observada mesmo com um volume de dados não muito grande (Ex: menor que 1TB). Um sistema que utiliza índices adaptativos se põe disponível para responder a consultas imediatamente, mesmo sem o suporte de índices. A cada consulta que responde, melhora a estrutura de forma a oferecer melhor performance para as consultas seguintes. Isto é, o sistema ganha conhecimento dos dados armazenados ao acessá-los, usando cada nova consulta como uma sugestão sobre como organizar os registros.

2.0.2 Construção incremental

A figura 3 mostra o comportamento esperado para a performance de um índice adaptativo. O valor constante mais alto corresponde a uma varredura ou busca sobre os dados sem auxílio de índice, que por necessitar verificar cada um dos registros tem complexidade linear no número de comparações. O valor constante mais abaixo corresponde a uma busca sobre o dado contido em um índice de árvore completo, que tem complexidade logarítmica. A primeira consulta sobre o índice adaptativo não tem nenhum auxílio da estrutura de dados e por isso tem performance comparável ao da varredura. Ao longo da sequência de consultas

a estrutura adaptativa se torna mais abrangente e capaz de podar o espaço de busca mais eficientemente. Como pode ser visto nas seções seguintes, sobre técnicas específicas, após um número razoavelmente pequeno de buscas o índice adaptativo alcança performance semelhante ao de índices completos. É mostrado também que, considerando o tempo de construção do índice completo, o índice adaptativo mostra-se mais vantajoso.

Figura 3 – Perfil de desempenho esperado



As vantagens apresentadas por índices adaptativos são variadas, a primeira é não haver a necessidade de um período dedicado de indexação, mas os atrativos dessas estratégias como métodos de acesso para tarefas de exploração de dados vão além. O segundo diferencial é que a necessidade de seleção de índice é aliviada, pois o atributo a ser indexado pode ser escolhido apenas no ato de disparar a primeira consulta. Há também a interessante propriedade dos índices adaptativos de imprimir na própria disposição física dos dados no armazenamento a história das consultas já processadas, isto pode ser usado para extrair informação tanto da distribuição dos dados quanto da carga de trabalho processada até então, servindo como um histograma.

2.1 Database Cracking

A primeira técnica de indexação adaptativa a ser implementada e testada em um SGBD real foi o *Database Cracking* (IDREOS *et al.*, 2007) (PIRK *et al.*, 2014). Esta estratégia consiste em ordenar incrementalmente o atributo usado como chave em um banco de dados com armazenamento colunar (MonetDB). Assim, ao aproximar-se de uma coluna ordenada o custo de acesso se torna semelhante àquele de uma busca binária. Cada nova consulta particiona

o atributo em um dado valor, ou pivô, posicionando os registros com valores deste atributo menores que o ponto de particionamento antes daqueles com valores maiores ou iguais a este pivô. Este esquema de particionamento, ou *cracking* como é chamado, é semelhante a um passo do algoritmo *Quick Sort*, e por isso o *Database Cracking* é algumas vezes descrito como um *quick sort* incremental.

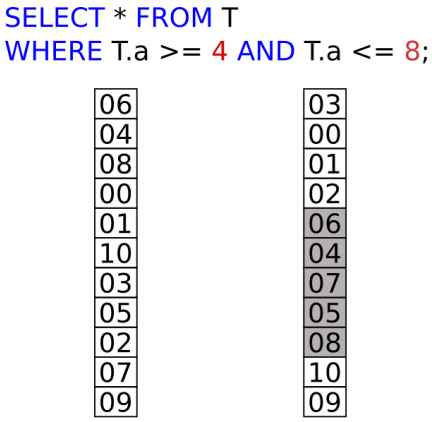
Os pontos de particionamento escolhidos a cada consulta são os extremos do intervalo de valores de chave que satisfazem o predicado da consulta. No processo de reorganização da coluna os valores são efetivamente trocados de posição, e se pode de imediato notar uma das características de um índice adaptativo. Isto é, a reorganização ocorre fisicamente no armazenamento, com movimento dos registros que frequentemente dispõe aqueles que fazem parte da resposta da consulta atual agrupados ao fim do processo. Consultas posteriores por intervalos do mesmo atributo que estejam contidos no intervalo já consultado são processadas muito rapidamente, pois estes registros se encontram agrupados e seus limites são conhecidos. Esta é outra característica dos índices adaptativos.

2.1.1 Estruturas de dados

Considerando a figura 4, os pontos de particionamento da coluna são escolhidos como os extremos de chave de busca admitidos pela seleção. Assim ao fim do processamento da consulta, a coluna terá sido dividida em três regiões: Uma contendo os valores de chave menores do que 4, uma ao centro contendo os valores maiores ou iguais a 4 e menores que 8, e uma ao final contendo os outros valores. Quaisquer consultas futuras não precisarão escanear a coluna inteira, pois poderão excluir ou incluir os registros de pelo menos uma das partições de imediato (sem verificar a chave) de acordo com o intervalo correspondente a cada partição. É necessário manter registro de onde se encontram os limites de cada partição na coluna e para este fim a técnica utiliza uma árvore AVL. Assim, a cada consulta, a coluna é particionada e o índice do primeiro registro pertencente à nova partição é inserido na AVL, junto com o valor de chave correspondente. Na mesma figura, é possível ver o estado final da coluna que contém os números do intervalo $[0, 10]$, inicialmente desordenados, após a partição ocasionada pela consulta.

A figura 5 mostra a evolução da coluna da figura 4 após mais uma consulta, que desta vez ocasiona uma partição que separa os valores maiores ou iguais a 7 e menores que 9. Os nós da árvore AVL correspondente a esta coluna também são mostrados na figura 5, neles, o número à esquerda é um valor de chave de busca onde existe uma partição e que também é

Figura 4 – Particionamento de uma coluna por uma consulta

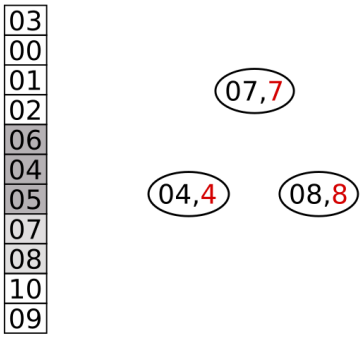


usado como chave de busca da árvore, enquanto o número à direita é a posição desta partição na coluna. Por exemplo, a raiz da árvore nos diz que o número 7 é um ponto de partição e que esta partição se encontra na 7ª posição da coluna, logo todos os registros com chave menor que 7 ocorrem antes dessa posição.

Para responder a uma consulta, inicia-se por buscar na árvore AVL pelos nós cujas chaves sejam as mais próximos dos valores extremos da chave de busca que satisfazem ao predicado. Com isto encontram-se as partições onde o intervalo de registros desejado começa e termina respectivamente. Apenas os registros nestas duas partições precisam ser testados pelo predicado, e todos os registros em partições que se encontrem entre estas extremidades pertencem à resposta. A coincidência dos números nos nós, na figura 5, se dá pelo fato de a coluna conter os números naturais entre 0 e 10, assim ao ordená-los, cada um que marca a fronteira entre duas partições ocupará a posição de índice igual a seu valor na coluna. Os autores mostram também que não é conveniente criar partições menores que o tamanho de uma linha de cache do processador, e com isto há a necessidade de ordenar completamente a coluna. A razão para isto é que a linha é a menor unidade de transferência entre a memoria e os caches, e por isso nenhuma transferência é economizada criando partições menores do que uma linha.

Dois algoritmos existem para efetuar as buscas e reorganizações da coluna, os quais os autores chamaram de *CrackInTwo* and *CrackInThree* (IDREOS *et al.*, 2007) (SCHUHKNECHT *et al.*, 2013). Como o nome sugere, o primeiro particiona um intervalo em duas novas partições e é detalhado no algoritmo 1. Nele, os valores `posL` e `posH` são as posições na coluna do primeiro e do último valores pertencentes ao intervalo a ser particionado. O valor `med` é o

Figura 5 – Evolução da AVL auxiliar



ponto de particionamento, isto é, ao término da execução todos os valores menores que `med` estarão aglomerados no início da coluna, enquanto os que forem maiores ou iguais a `med` estarão aglomerados no final no intervalo. A escolha natural de `med` para um índice adaptativo é algum valor representativo do intervalo consultado, por exemplo, um dos valores extremos de chave de busca requisitados ou a média deles.

Algoritmo 1: CrackInTwo(c, posL, posH, med, inc)

```
x1 := ponto na posição posL;
x2 := ponto na posição posH;
while Posição(x1) < Posição(x2) do
    if Valor(x1) < med then
        x1 := ponto na próxima posição;
    else
        while Valor(x2) < med && Posição(x2) > Posição(x1) do
            x2 := ponto na posição anterior;
        end
        Troca(x1, x2);
        x1 := ponto na próxima posição;
        x2 := ponto na posição anterior;
    end
end
```

O algoritmo *CrackInThree* é uma forma modificada do *CrackInTwo* no qual o intervalo é dividido em três partes. Os pontos de divisão são então os extremos de chave admitidos pela consulta, assim todos os registros que fazem parte da resposta ficam juntos na partição central gerada no intervalo original. Embora o *CrackInThree* tenha a possibilidade de

gerar partições menores e mais numerosas aproximando o estado da coluna mais rapidamente do estado ordenado, os autores notam através de experimento que o efeito na performance é semelhando ao do *CrackInTwo* e que este adiciona custo menor ao processamento das consultas e por isso recomendam seu uso.

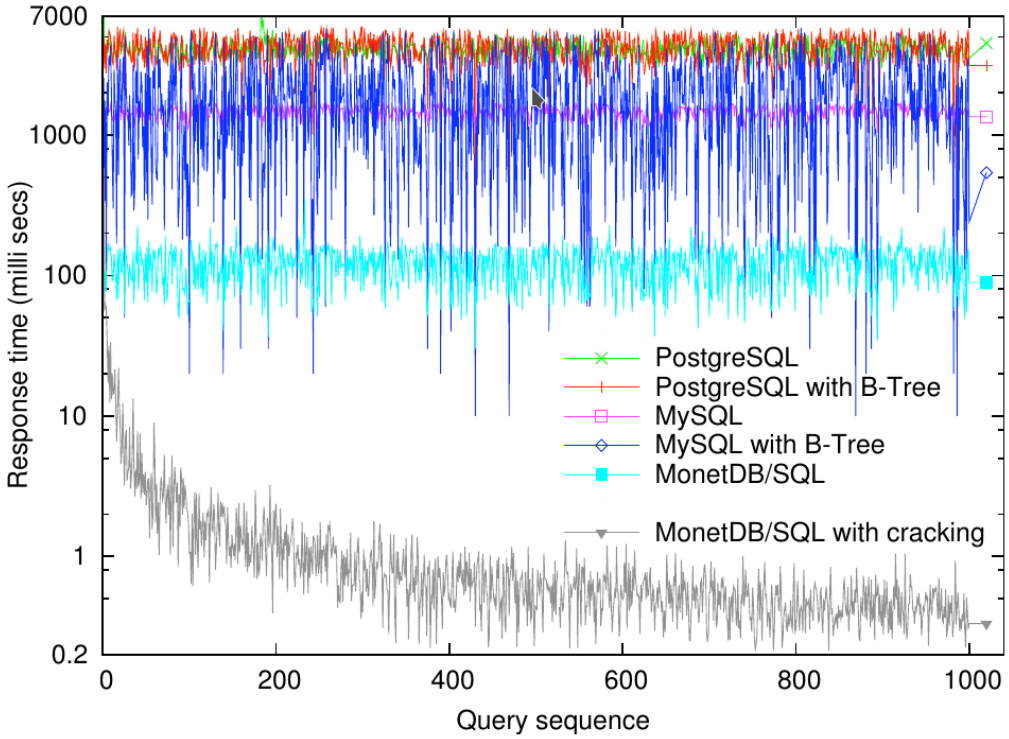
Quanto às estruturas utilizadas para manter um índice to tipo *Database Cracking*, a implementação original efetua a reorganização de uma cópia da coluna a ser indexada em memória principal, que é chamada de *Cracker Column* e a árvore AVL que mantém registro das partições é chama de *Cracker Index*. Assim, esta estratégia exige que a coluna, ou atributo da relação usando o jargão relacional, seja menor que a memória disponível. Esta não é uma limitação muito severa visto que o interesse são atributos numéricos e as demais colunas, se houverem, podem ser residentes and HDDs. A contínua reorganização de uma coluna isoladamente cria um problema para aplicações OLTP, pois se mais de um atributo for requisitado o sistema tera mais trabalho para reconstruir os registros, visto que seus atributos estarão desalinhados. Esta porém, também não é uma limitação substancial para o método devido a seu foco em aplicações analíticas, o que fica evidente desde o início pela própria escolha de um SGDB com armazenamento orientado a colunas.

2.1.2 Resultados

A avaliação experimental do *Database Cracking* foi feita sobre o MonetDB carregado com um dado composto de uma coluna com 10 milhões de valores inteiros a serem indexados. A carga de trabalho utilizada se compõe de 1000 consultas por intervalo, isto é, consultas que SQL têm forma geral `SELECT...WHERE x >= a AND x <= b;` onde `x` é o atributo a ser consultado e indexado e `a` e `b` são os extremos do intervalo que satisfazem à consulta.

Na figura 6 vemos os resultados de tempo de resposta obtidos pelos autores. O MonetDB apresenta os melhores resultados tanto sem índices quanto com o *Database Cracking* em ação, o melhor cenário. Os SGBDs com armazenamento orientado a tuplas, que neste caso são o PostgreSQL e o MySQL apresentam tempos maiores, como esperado para este tipo de carga de trabalho. Um resultado que chama a atenção é o fato de o PostgreSQL ter apresentado basicamente os mesmos valores com ou sem um índice baseado em árvore.

Figura 6 – Database Cracking - Tempos de resposta



Fonte: (IDREOS *et al.*, 2007)

2.2 Adaptive Merging

Na intenção de elaborar uma estratégia de indexação adaptativa com características semelhantes às do *Database Cracking*, porém adequada a dados armazenados como tuplas e residentes em HDs, buscou-se por estruturas baseadas em árvores B que pudessem ser construídas incrementalmente e que fossem úteis às consultas mesmo antes de estarem completas.

2.2.1 Estruturas de dados

A estrutura escolhida foi a árvore B+ particionada (GRAEFE, 2003). Estas estruturas, são idênticas às árvores B+ comuns com a exceção de que um atributo extra, comumente chamado de chave artificial, é adicionado a cada um dos registros para garantir a ordem global sem a necessidade de acessar cada registro mais de uma vez ($O(n)$ em termos de número de leituras + escritas). O processo de construção desta estrutura de árvore a partir de um conjunto de registros desordenados, como um arquivo *heap*, se dá de acordo com o algoritmo 2.

A variável `J` denota a partição corrente, e é sequencial, então os registros de cada partição criada terão chave artificial `0, 1, 2, etc...` O número de registros lidos a cada vez

Algoritmo 2: ConstroiArvoreParticionada()

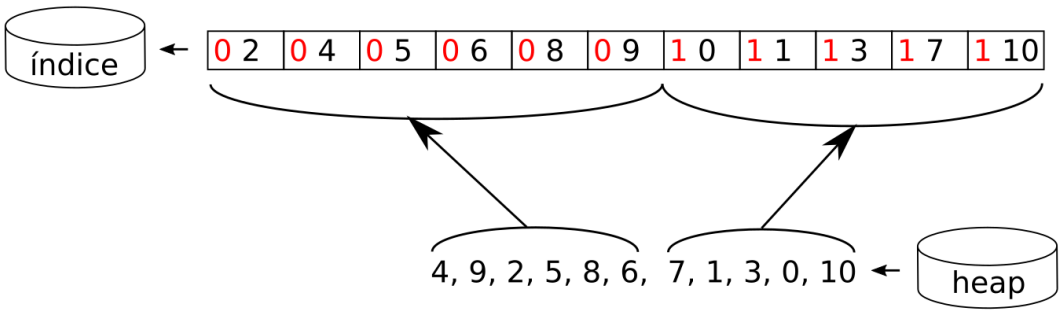
```
J = 0;
while Ainda existem registros no arquivo heap do
    Leia N registros do arquivo heap;
    Ordena os registros em memória;
    Adiciona J como a chave artificial de cada registro;
    Constroi niveis não folha (Bulk Load);
    Escreve os registros ordenados no arquivo índice;
    Incremente J;
end
```

é limitado pela quantidade de memória disponível, e é interessante que seja o maior possível pois partições pequenas ocasionam um número maior de partições e, assim, contribuem pouco para acelerar as buscas. A ordenação dos registros da partição corrente se dá em memória e pode ser executado seguindo-se qualquer algoritmo adequado para este contexto.

2.2.2 Construção

A construção dos níveis não folha se dá através de um algoritmo de *Bulk Loading*, operação implementada em todos os SGBDs com índices B+ e não sofre nenhuma modificação para o contexto de árvores particionadas. Note-se que a adição de um atributo novo na chave de busca de um índice B+ não constitui novidade, trata-se apenas de um índice com chave composta, onde a operação de comparação de chaves se dá de forma lexicográfica, isto é, compara-se o primeiro atributo da chave se no caso de igualdade usa-se o próximo para desempate.

Figura 7 – Construção do nível folha



A figura 7 ilustra o processo de construção do nível folha, onde os dados se encontram. Os registros são lidos do *heap* e cada um contém a chave de busca definida pelo usuário (seleção

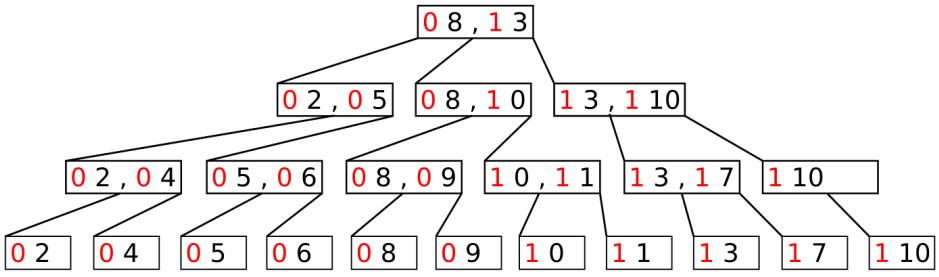
de índice) em ordem aleatória, cada conjunto que constituirá uma nova partição da árvore é então ordenado e ganha a chave artificial como prefixo (número em vermelho) e então, este dado organizado é escrito no arquivo de dados que conterá a a relação indexada, liberando espaço em memória para a próxima partição.

A construção neste nível requer uma leitura sequencial dos dados, a mesma carga de leitura necessária para uma consulta sobre os dados não ordenados. Assim, o *Adaptive Merging* (GRAEFE; KUNO, 2010) executa esta leitura para responder à primeira consulta da carga de trabalho e aproveita a passagem dos registros pela memória para ordená-los. O custo adicional nesta primeira consulta é o processamento necessário para ordenar as partições. A escrita das partições ordenadas pode ser feita em novo arquivo, possivelmente em um outro dispositivo por isso não representa perda de performance significativa para a carga de trabalho corrente. A estrutura mostrada nas figuras 7 e 8 é ilustrativa e deliberadamente compacta, aquelas construídas na prática com o fim de servir como índices comportam múltiplas chaves por nó, frequentemente da ordem de 100 chaves, e mantém cerca de um quarto da capacidade livre para receber inserções de novos registros.

A figura 8 mostra a forma final da árvore particionada, após o processo de *Bulk Loading* ser finalizado incluindo os níveis não folha. A partir desta figura pode-se notar as principais características desta estrutura:

- Todos os registros estão em ordem, considerando-se a adição do atributo que identifica a partição (em vermelho).
- A chave artificial se torna apenas mais um atributo dos registros aparecendo tanto nos dados contidos nas folhas quanto nas chaves de busca dor níveis acima
- A árvore particionada ocupa mais espaço que uma não particionada contendo os mesmos registros, pelo fato de cada registro e entrada de chave conter uma atributo extra.

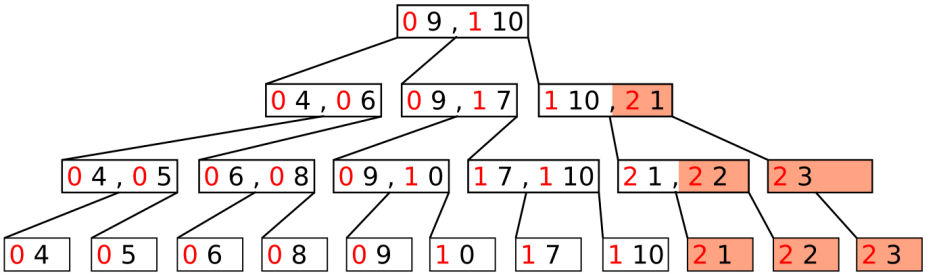
Figura 8 – Árvore B+ particionada



Em um índice de chave composta, as buscas tem a melhor performance se o predicado selecionar pelo valor do primeiro atributo. Este nunca é o caso em uma árvore B+ particionada, pois o primeiro atributo da chave de busca é o identificador da partição, que não tem nenhum significado para as aplicações interessadas nos dados. Assim, o processo de busca na árvore particionada, por um valor ou por um intervalo de valores, se guiará em geral pelo segundo atributo da chave (o número em preto na figura 8). Para se responder a estas buscas é necessário então percorrer a árvore da raiz até as folhas uma vez para cada partição. A cada descida, busca-se o registro que tenha atributo artificial correspondente ao número da partição corrente e o segundo atributo da chave igual ao menor valor aceito pela consulta, daí segue-se o encadeamento das folhas até recolher todos os registros que fazem parte da resposta existentes na partição corrente. Repete-se este procedimento para cada uma das partições da árvore. Tendo em vista este procedimento é possível notar-se o porquê de, para um número fixo de registros, árvores com menos partições terão performance melhor.

Toda a construção deste índice particionado se dá durante o processamento da primeira consulta, aproveitando-se das leituras feitas para respondê-la. O caráter adaptativo do *Adaptive Merging* reside no procedimento usado para responder às consultas seguintes. O objetivo é executar operações de fusão nas partições durante o processamento das consultas ao transferir os registros que fazem parte das respostas para uma partição dedicada. Suponha-se, por exemplo, que sobre a estrutura mostrada na figura 8 se processe uma consulta que solicita os registros com chave no intervalo $[1,3]$. Após o processamento desta consulta, os registros que compõe a resposta terminarão todos em uma nova partição como mostrado na figura 9.

Figura 9 – Árvore B+ particionada após uma fusão



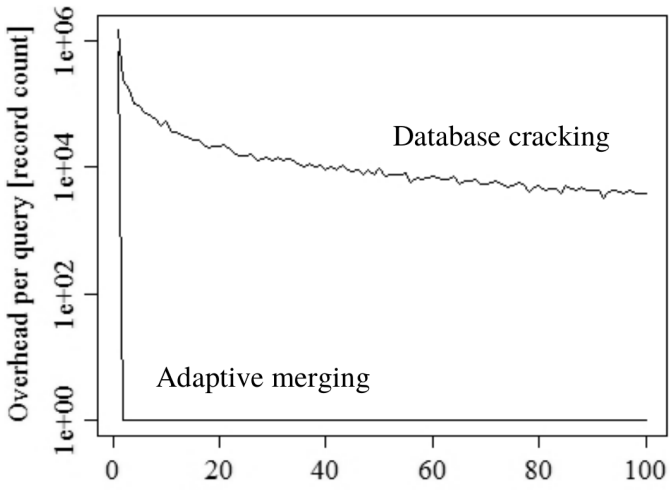
A aparição de uma nova partição pode parecer contrária à ideia de executar operações de fusão ou *merge sort*, mas ocorre que a cada nova consulta mais registros são transferidos para esta partição e eventualmente as partições originais vão se tornando vazias e a partição 2

da figura 9 se torna toda a estrutura (partição hachurada ocupa cada vez mais espaço). Neste ponto as buscas só precisarão percorrer a árvore uma vez e os custos se tornam iguais aos dos algoritmos de uma árvore B+ simples.

2.2.3 Resultados

O *Adaptive Merging* demonstrou ter convergência mais rápida que o *Database Cracking*, isto é, demanda menos consultas até atingir o estado de índice completo. O custo de inicialização porém é maior, ou seja, as operações adicionadas à primeira consulta são mais custosas. Uma comparação de performance não é direta, pois o primeiro concentra-se em dados armazenados em dispositivos de blocos enquanto o segundo a armazenamento colunar em memória principal. Os autores optaram por expressar a comparação em termos de sobrecarga por consulta, isto é, a quantidade de registros acessados a mais, em relação a uma busca pura (que não ajuda na construção dos índices). A figura 10 Mostra esta comparação.

Figura 10 – Adaptive Merging - resultados



Fonte: (GRAEFE; KUNO, 2010)

2.3 Comparação

A tabela 2 a seguir resume as características das principais estratégias de indexação adaptativa e adianta os diferenciais da nossa estratégia, o MetisIDX, que serão discutidas detalhadamente no capítulo seguinte. Outras técnicas, que constituem variações do *Database Cracking*, e que não foram discutidas a fundo neste capítulo aparecem a propósito de completude.

Como mostrado na tabela, todas as técnicas adaptativas têm o processo de construção

Tabela 2 – Comparação das estratégias

Técnica	Orientado à carga	Estritamente adaptativa	Em memória	Aprendizado
Database Cracking	**X**	**X**	**X**	
Adaptive Merging	**X**	**X**		
Stochastic Cracking	**X**		**X**	
Holistic Indexing	**X**		**X**	
MetisIDX	**X**			**X**

da estrutura de dados orientado à carga de trabalho sendo processada. O *Stochastic Cracking* (HALIM *et al.*, 2012) é uma técnica que adiciona um componente aleatório na escolha dos pivôs do *Database Cracking*, fazendo com que os intervalos indexados não sejam necessariamente iguais aos consultados na esperança de indexar intervalos longos melhorando a performance das consultas futuras. O *Holistic Indexing* (PETRAKI *et al.*, 2015) segue um raciocínio semelhante ao da estratégia anterior, mas também é oportunista ao aproveitar-se dos núcleos de processamento ociosos para executar particionamentos aleatórios repetidamente em paralelo com as consultas. Estas são as estratégias que chamamos de não estritamente adaptativas.

O *Database Cracking* e as técnicas derivadas dele são consideradas adequadas para sistemas com armazenamento colunar em memória. A restrição de ser em memória pode ou não ser uma limitação dependendo da aplicação, pois se os dados cabem na memória principal um sistema pensado pra essa situação é desejável. O *Adaptive Merging* é pensado para sistemas nos quais a residência primária dos dados é um disco rígido e até mesmo a coluna ou atributo a ser indexado pode ser muito maior que a memória principal disponível. Esta característica é compartilhada pela nossa estratégia.

O MetisIDX, a ser descrito no capítulo 3 não é uma técnica estritamente adaptativa, pois os intervalos indexados não são idênticos àqueles consultados, mas mantém a característica desejável de se guiar pela carga de trabalho sendo processada. Para isto, lança mão das ferramentas da aprendizagem de máquina, inspirando-se nos resultados recentemente alcançados por diferentes áreas, que estão utilizando esta ferramenta como meio de criar sistemas que não necessitam de administração e tomada de decisão humana para operar.

3 METISIDX

Construir estruturas de acesso para cenários de exploração de dados como um passo anterior ao seu uso não é uma boa estratégia, pois como visto, o conhecimento dos dados necessário para a sua construção é justamente o que se quer obter do processo. Por isto, as técnicas de indexação adaptativa propuseram postergá-la até o momento em que o sistema tiver de processar a carga de trabalho, e assim poder extrair conhecimento dela.

Esta característica é herdada pala estratégia MetisIDX juntamente com a capacidade de evoluir as estruturas de dados continuamente por meio de operações incrementais e de baixo custo. Nossa estratégia é relacionada, e até certo ponto inspirada, pelos índices adaptativos, porém não se trata de uma técnica estritamente adaptativa, pela maneira diferente como tira proveito do conhecimento acumulado da carga de trabalho já processada.

A primeira diferença em relação às técnicas adaptativas é o desacoplamento das operações de processamento de consulta e indexação. Nos índices adaptativos, a consulta atual determina a região dos dados a ser indexada, o intervalo de chave que constitue a resposta da consulta é movido para uma localização de acesso mais fácil (tal como uma nova partição da coluna ou uma sequência de folhas de uma estrutura de árvore). Com isto, causam maior impacto em consultas futuras que requisitarem dados que já foram acessados, pois ao recuperá-los novamente, sua localização é conhecida e novas operações de movimentação de registros não são necessárias.

Acompanhar a carga de trabalho, no entanto, não necessariamente requer que as operações de indexação e processamento de consultas sejam uma só. Torná-las independentes adiciona a vantagem de possibilitar o emprego de paralelismo, um recurso frequentemente disponível e por vezes subutilizado. O motivo disto é que as máquinas atuais contam com múltiplos núcleos de processamento e dificilmente todos são utilizados simultaneamente pelo SGBD (YU *et al.*, 2014). O gargalo de paralelismo não é a disponibilidade de *threads*, mas sim a contenção de bloqueios gerada por acesso concorrente aos items de dados.

Os recursos de processamento que ficam ociosos podem então ser utilizados para outros fins. MetisIDX os utiliza paras as atividades de indexação incremental, que agora não são mais conjuntas com o processamento de consultas, e para decidir quais intervalos de chave indexar a seguir. Esta última atividade é fruto de mais uma flexibilidade oferecida pelo desacoplamento do processamento de consultas: os intervalos de chave indexados não precisam ser idênticos aos já consultados, podendo ser escolhidos de forma a beneficiar mais a

performance ao longo da sequência das consultas. Realizar passos de indexação baseando-se apenas na consulta corrente tem mais uma desvantagem: otimiza o acesso majoritariamente para os registros já acessados e reage mesmo a consultas que se desviem do padrão mais amplo da carga de trabalho.

Neste ponto, MetisIDX tem sua principal adição sobre técnicas de indexação estritamente adaptativas, o foco pode mudar da consulta que está atualmente sendo respondida para as consultas que terão de ser respondidas a seguir. Adicionar este comportamento preditivo ao método de acesso conserva a característica de guiar-se pela carga de trabalho dos índices adaptativos, libera a *thread* que trabalha para responder à consulta atual do esforço de reorganizar fisicamente os dados e ainda aumenta as chances de consultas futuras encontrarem os registros de interesse de forma ótima. Responder à consulta de forma ótima neste contexto significa encontrar os registros que compõe a resposta em sua representação final de índice completo, o que equivale no caso do *Database Cracking* a encontrá-los todos em uma única partição da coluna.

Todas as consultas respondidas pelo sistema podem ser usadas em uma análise do padrão subjacente da carga de trabalho e, consequentemente, nas decisões sobre como modificar as estruturas de dados. Reagir sempre baseando-se na consulta atual isoladamente pode fazer com que se dedique muito tempo de processamento a troco de pouco proveito (particionar um intervalo grande que nunca será consultado novamente, por exemplo) este problema se assemelha às discussões sobre *overfitting*[1] e generalização, comuns nas tarefas de reconhecimento de padrões e aprendizagem de máquina. Este trabalho se propõe a levar esta analogia adiante e de fato empregar uma técnica de aprendizado de máquina para guiar a construção de estruturas de indexação utilizando as consultas previas como dado de treinamento.

Toda aplicação tem uma lógica interna, as consultas não são de fato aleatórias, um padrão subjacente existe e depende tanto dos conteúdos dos dados quanto das intenções dos analistas. Juntando este raciocínio ao fato de que, recentemente, as técnicas de aprendizado de máquina têm apresentado resultados positivos em aprender rapidamente padrões advindos de processos desconhecidos, temos a motivação para avaliarmos sua utilização na construção de métodos de acesso para exploração. No entanto, em momento algum se pode considerar a história futura das consultas conhecida, esta deve ser a premissa em sistemas de gerenciamento de dados para exploração: a mudança é a única constante nestes cenários. A hipótese sobre a qual a técnica MetisIDX se baseia é que a próxima região do dado que será de interesse para a

[1] Ocorre quando um modelo se torna melhor em descrever as características dos dados conhecidos do que para descrever outras amostras da mesma fonte (dados ainda não vistos)

aplicação pode ser prevista com base nas requisições mais recentes. Assim o caráter preditivo existe, mas de forma localizada no tempo, isto é, sempre com vista em um período curto onde determinado padrão de acesso pode se manter, e exige atualização contínua do modelo.

Recentemente, um raciocínio semelhante foi utilizado para responder consultas analíticas (PARK *et al.*, 2017). Este trabalho usa redes neurais para substituir níveis de uma árvore B+, e produz respostas aproximadas ou parciais, o que não é demérito, pois tais respostas são aceitáveis a um grande número de aplicações analíticas e possibilitam um ganho muito expressivo de performance e espaço de armazenamento. O referido trabalho reconhece que:

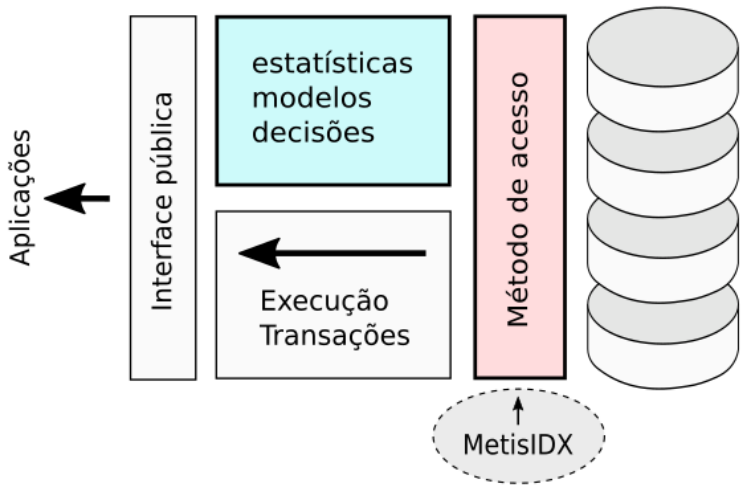
> The answer to each query reveals some fuzzy knowledge about the answers to other queries, even if each query accesses a different subset of tuples and columns (PARK *et al.*, 2017).

A hipótese de que algum grau de erro nas respostas é aceitável é usada e relacionada com os erros do processo de aprendizado da rede. No MetisIDX por outro lado, nos propomos a responder consultas de forma exata. Erros de predição naturalmente ocorrem, mas não se refletem nos resultados das consultas, apenas ocasionam mais acesso à parte ainda não indexada dos dados. Assim a taxa de erro nas predições do modelo têm impacto na performance mas não na corretude.

3.1 Idéias para novos sistemas

MetisIDX é um método de acesso, e como tal, determina a maneira como os diferentes níveis de memória são acessados. A figura 11 mostra onde se encaixa na arquitetura geral de um SGBD e também como parte de uma ideia de nova abordagem de implementação desta arquitetura, onde o componente central é um módulo que monitora continuamente o funcionamento de todo o sistema e se encarrega das diversas decisões com base em conhecimento acumulado e construção de modelos. A ideia central é tornar o sistema de gerenciamento de bancos de dados uma entidade autônoma e, em certo sentido, consciente de suas funções e dos recursos disponíveis. Estes desenvolvimentos são concomitantes com diversas outras classes de sistemas que atualmente estão abraçando a inteligência artificial como forma de superar a dependência de microgerenciamento humano contínuo.

Figura 11 – MetisDB



3.1.1 MetisDB

Os primeiros módulos de um sistema com esta proposta foram desenvolvidos sob o nome de MetisDB que, por contar ainda apenas com o nível de métodos de acesso, não se caracteriza como um SGBD completo. Os componentes desenvolvidos para este sistema têm extensibilidade em mente, podendo ser embarcados em outros sistemas. Esta parte de arquitetura de *micro kernel* para bancos de dados têm recebido a denominação de motor de armazenamento numa tradução livre (*storage engine*). Outros exemplos contemporâneos são o RocksDB [2] e o WiredTiger [3].

O MetisDB conta com um pequeno núcleo que disponibiliza interfaces de programação para carregamento de um número arbitrário de módulos, comunicação e relações de dependência entre módulos e armazenamento e acesso de valores de configurações de sistema. Este último elemento teve atenção especial pela experiência de outros sistemas mais antigos. Bancos de dados atuais têm muitas centenas de configurações ajustáveis (*knobs*) e é interessante disponibilizar uma interface consistente para recuperar e modificar valores e eventos para notificar os módulos sobre modificações de configurações de seu interesse. Com isto o MetisDB se propõe a construir componentes que podem ser combinados de diferentes formas para gerar não um sistema com características fixas mas diferentes sistemas para diferentes cenários. Estes componentes formarão um conjunto de ferramentas particularmente interessante para uso em pesquisa, onde flexibilidade é essencial. Evolução para um sistema de produção não é um obje-

[2] http://www.rocksdb.org/
[3] http://www.wiredtiger.com/

tivo imediato mas decisões de implementação são tomadas tendo em mente esta possibilidade e as necessidades das aplicações atuais.

3.2 Carga de trabalho

Indexar os invervalos de chave de busca tendo em vista os acessos futuros implica um processo preditivo que, por sua vez, requer a construção de um modelo. As variáveis de interesse são aquelas que caracterizam a sequência de consultas analíticas comuns em tarefas de exploração. Seguindo a metodologia utilizada nos trabalhos a respeito de indexação adaptativa já estabelecidos, consideramos a construção de um índice sobre um único atributo cujo tipo de dados suporte a operação de "$<$" (menor que), isto é, cujos valores possam ser ordenados. A carga de trabalho é vista pela camada de método de acesso como uma sequência de consultas por intervalo de chave be busca. A evolução histórica, ou série temporal, que a caracteriza é então expressa pela sequência

$$Q = (l_0, h_0), (l_1, h_1), ..., (l_j, h_j), ... \tag{3.1}$$

de pares de números l e h, onde l é o valor mínimo de chave que atende ao predicado da consulta e h é o valor máximo. Dado que o método de acesso vê todas as consultas como buscas por intervalos de chave, esta sequência contém toda a informação sobre a carga de trabalho. Uma forma equivalente de representar esta sequência é por meio de duas funções $\mathbb{N} \Rightarrow \mathbb{R}$ dadas por

$$\begin{aligned} \phi &= l(j) \\ \psi &= h(j) \end{aligned} \tag{3.2}$$

onde j é a ordem da consulta na sequência. Esta forma é mais conveniente por que a dependência funcional deixa explícitos a entrada do modelo (j), e as funções que devem ser aprendidas por ele (ϕ e ψ).

3.2.1 Atributos de interesse

Do ponto de vista de aprendizado e reconhecimento de padrões, o dado de treinamento é, em alto nível, a carga de trabalho. É necessário então tornar claro quais são os atributos mensuráveis que consideramos caracterizar a carga de trabalho. Em geral, podemos atributos advindos das consultas e também do próprio dado armazenado. Estes últimos são coletados ao fim do processamento de cada consulta e podem ser tão valiosos quanto aqueles

relativos às consultas. Exemplos de atributos originados nos dados são os valores agregados das respostas das consultas já respondidas (extremos, soma, média, etc). Estes atributos podem dar aos algoritmos de aprendizado informação sobre o dado, ou pelo menos sobre o subconjunto mais frequentemente acessado ou mais recentemente acessado.

Dito isto, as funções 3.2 foram selecionadas com o critério da navalha de Occam e são as mais simples relações que se pode encontrar entre os atributos mais imediatos da carga, isto é, os extremos do intervalo de chave de busca requerido pelas consultas, com a sua sequência. É razoável também modelar ϕ e ψ com relações entre si e com as respostas das consultas anteriores, pois isto adiciona informação sobre a distribuição do atributo indexado. Isto envolve uma discussão sobre a capacidade de predição de classes arbitrárias de funções por parte do algoritmo de aprendizado utilizado, ou uma abordagem experimental de avaliar a performance de várias dependências funcionais diferentes no contexto de cargas de trabalho reais. Em ambos os casos a discussão se concentra mais em torno do processo de aprendizado do que em métodos de acesso.

3.3 Aprendizado e predição

Recuperando o fato da disponibilidade de *threads* de processamento, faz sentido dedicar uma delas ao processo de treinar continuamente o modelo da carga de trabalho. Os dados de treinamento para o algoritmo de aprendizagem, que são as sequências de j, ϕ e ψ da seção anterior, são coletados a cada nova consulta, é conveniente esperar pelo processamento de algum número N de consultas e então atualizar o modelo através de um processo de mini batch.

O valor de N precisa ser escolhido, é um hiperparâmetro no jargão de aprendizado de máquina. Escolher um N pequeno significa que as atualizações do modelo serão muito frequentes e que se espera que o padrão de acesso varie muito rapidamente. Já escolher um N grande, implica que a tarefa de treinamento será menos frequente e que no intervalo entre elas o modelo pode se tornar obsoleto e deixar de atender às necessecidades das consultas. Poucos critérios genéricos existem para a escolha deste tipo de parâmetro, mas dada uma aplicação específica, pode acontecer de o tempo típico necessário para responder a uma consulta possa ser estimado ou uma possível periodicidade no comportamento da carga de trabalho seja conhecida. Nestes casos onde conhecimento especialista ou conhecimento sobre a aplicação está disponível, estes recursos devem ser utilizados na escolha de qualquer hiperparâmetro.

3.3.1 Recursos utilizados

Se uma thread for dedicada à atividade de observar e executar o treinamento então a frequência das tarefas de treinamento não são um problema. Seria então conveniente arranjar para que o tempo necessário para executar um mini batch aproxime-se do tempo necessário para processar *N* consultas com as estruturas de dados no seu estado atual. Desta forma uma nova versão do modelo sempre estaria sendo criada enquanto a atual é utilizada para disparar tarefas de indexação. Este arranjo no entanto, é mais facilmente descrito do que implementado, e constitui um estudo à parte. Nos experimentos descritos no próximo capítulo, um número *N* fixo foi utilizado com o critério de permitir mini batches rápidos considerando a técnica de aprendizado utilizada, que será descrita a seguir.

Globalmente, os recursos consumidos para manter um modelo atualizado não são restritivos mesmo considerando que isto ocorrerá como parte integrante do sistema. Além da thread dedicada à atualização contínua, o armazenamento necessário é pouco volumoso, o estado da maioria dos modelos se constitue de uma matriz de números com algumas centenas de linhas e colunas, ou mesmo uma única coluna.

3.3.2 Escolha de algoritmos de aprendizagem

Nem todos os algoritmos de aprendizado são adequados para o fim de guiar o processo de indexação. Convém agora, considerar quais são as limitações a fim de justificar a escolha feita para a implementação do MetisIDX. Muitas das técnicas que são estado da arte no campo da aprendizagem de máquina e em especial em aprendizado profundo requerem grandes quantidades de dados para convergirem e além disso requerem tempos de treinamento que são no mínimo da ordem de horas mesmo fazendo uso de *Graphics Processing Unit* (GPU).

Tais técnicas são claramente inadequadas, pois o modelo usado para construção dos índices precisa ser útil mesmo depois de observar poucas consultas, seria de pouca utilidade por exemplo um modelo que precisa de milhares de consultas observadas para só então ser capaz de oferecer saída significativa, pois até então não se teria suporte de índices, ou o uso de um índice adaptativo já teria atingido nível de performance comparável a um índice completo. Além de demorarem a oferecer utilidade, algoritmos complexos também se tornariam concorrentes por recusos do hardware onde o SGBD opera, assumindo que o modelo e o algoritmo associado são implementados como uma parte integrante de método de acesso que os utiliza que, por sua vez,

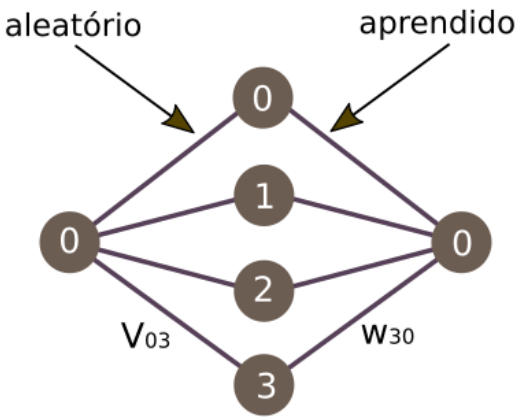
é parte central do SGBD. Portanto, um compromisso existe entre a capacidade dos algoritmos de aprendizagem candidatos de criar modelos complexos e a sobrecarga que eles introduzem no sistema.

Uma vez que não há a intenção de modelar ou prever a carga a longo prazo, também não há a necessidade de se utilizar algoritmos capazes de gerar modelos muito elaborados. O MetisIDX requer previsões aproximadas de umas poucas consultas para manter o processo de indexação executando. Pode se pensar numa analogia com uma expansão em série de Taylor da função objetivo, pode-se utilizar apenas os primeiros termos, desde que se pretenda modelar apenas pontos próximos do ponto base da expansão.

3.4 Máquina de aprendizado extremo

Uma técnica que atende aos requisitos discutidos na seção anterior é o *Extreme Learning Machine* (ELM) (HUANG *et al.*, 2004). Trata-se de uma rede neural artificial que vem sendo utilizada com sucesso em diversos tipos de tarefas (HUANG *et al.*, 2012) e pode teoricamente modelar funções arbitrárias. O ELM é uma rede *feed forward* comum que poupa esforço de treinamento mantendo alguns dos pesos que compõe o modelo fixos.

Figura 12 – Máquina de aprendizado extremo



A figura 12 ilustra uma rede ELM com arquitetura igual à utilizada na experimentação do MetisIDX. Esta rede conta com um neurônio na camada de entrada, que recebe o valor de j, ou seja, a ordem sequêncial da consulta. Uma única camada oculta existe com algum número de neurônios rotulados com os número de 0 a 3 e em seguida a camada de saída, mais uma vez formada por um único neurônio cuja saída é a previsão da rede para o limite mínimo ou máximo

do intervalo de chave consultado. Duas redes independentes são utilizadas, cada uma modula a dependência de um dos limites do intervalo consultado com j, ou de maneira equivalente, uma aprende ϕ e outra aprende ψ.

3.4.1 Treinamento

Cada neurônio computa uma função sigmóide sobre o valor somado de suas entradas, esta função de ativação é utilizada para habilitar o modelo a aprender funções não lineares. Com isto a relação entre a entrada e a saída de cada neurônio é idêntica à de um *perceptron* de múltiplas camadas:

$$x_i^{k+1} = \sigma\left(\sum_j u_{ji}^k x_j^k\right) \tag{3.3}$$

Os pesos das ligações entre os neurônios da camada de entrada e os neurônios da camada oculta são fixados em valores escolhidos aleatoriamente do intervalo [0, 1], esta é a característica que identifica uma ELM. Quatro destes pesos existem, denotados por V_{ij}, significando "peso da ligação entre o neurônio i da camada de entrada e o neurônio j da camada oculta". i só tem um valor, 0, pois apenas um neurônio de entrada existe, j tem quatro valores possíveis correspondendo aos 4 neurônios ocultos. Na figura 12 o peso V_{03} está ilustrado.

Já os pesos das ligações entre os neurônios da camada oculta e os neurônios da camada de saída, W_{ij}, são treinados a partir das observações, isto é, consultas já processadas no contexto do MetisIDX. Este treinamento é executado segundo o algoritmo *gradiente descendente estocástico*, em um passo único, e tem o custo de calcular a pseudo inversa de uma matriz. Esta matriz terá apenas algumas dezenas de colunas e linhas pois esta é a ordem de grandeza do número de neurônios na rede utilizada e do número de consultas observadas até que uma nova etapa de treinamento seja disparada, portanto o treinamento é barato em termos de processamento. Na primeira tarefa de treinamento, quando um modelo prévio ainda não existe, a expressão 3.4 é usada para criar o modelo.

$$\mathbf{W_2} = \sigma(\mathbf{W_1 X})^{+}\mathbf{Y} \tag{3.4}$$

onde $\mathbf{W_1}$ é a matriz dos pesos das ligações entre os neurônios da camada de entrada e os neurônios da camada oculta, $\mathbf{W_2}$ é a matriz dos pesos das ligações entre os neurônios da camada oculta e os neurônios da camada de saída, $\mathbf{X}$ é o vetor dos atributos dos dados (j no contenxto do MetisIDX) e $\mathbf{Y}$ são as saídas observadas (extremos dos intervalos consultados). Após este

primeiro modelo ser treinado, futuros lotes de novas observações (novas consultas) devem apenas atualizar o modelo e não gerar um completamente novo, para que se tome proveito do conhecimento acumulado. Para estas etapas de treinamento seguintes o modelo é atualizado segundo a expressão 3.5 (LIANG *et al.*, 2006), conhecida como ELM online.

$$\begin{aligned} M_{k+1} &= M_k - M_k H_{k+1}^T (I + H_{k+1} M_k H_{k+1}^T)^{-1} H_{k+1} M_k \\ W_{k+1} &= W_k + M_{k+1} H_{k+1}^T (Y_{K+1} - H_{k+1} W_k) \end{aligned} \tag{3.5}$$

onde H é a matriz da saída da camada oculta e M é a covariância dos parâmetros e I é a matriz identidade.

3.5 Construção da estrutura de dados

A proposta apresentada neste trabalho vai além dos índices adaptativos, pois desacopla a construção das estruturas de dados do processamento das consultas, além de aproveitar melhor a informação disponível das requisições já processadas utilizando aprendizagem de máquina. Outras abordagens já foram propostas para acelerar a transição de um índice adaptativo da forma parcial para a forma de índice completo, porém exploraram processos aleatórios (HALIM *et al.*, 2012) indexando intervalos de chave por vezes diferentes dos buscados mas sem critério, ou estratégias oportunistas para meramente indexar intervalos ainda não pesquisados usando *threads* de processamento ociosas (PETRAKI *et al.*, 2015), novamente de forma estocástica. Todas estes esforços foram extenções do *Database Cracking*.

A evolução da estrutura de dados utilizada no MetisIDX herda algumas características do *Adaptive Merging* (AM), a principal delas é a ideia de utilizar uma árvore B+ particionada como estado transitório do índice para evitar custo maior que O(n) em número de acessos a registros em dispositivo de armazenamento secundário. Esta técnica foi escolhida como base por ter sido concebida para bancos de dados maiores do que o volume de DRAM disponível, e usar uma estrutura de dados baseada em árvore B. Estas estruturas e os algoritmos utilizados para acessá-las são adequados a dados armazenados em dispositivos com endereçamento por bloco ou com acesso preferencialmente sequencial como discutido anteriormente, e esta é a realidade da maioria das aplicações de exploração de dados.

A principal contribuição do MetisIDX se resume em acelerar o processamento das buscas antecipando a transição da estrutura de indexação adaptativa parcial até o estado de um índice completo. Isto é, de uma árvore B+ com várias partições de tamanho semelhante para

uma árvore B+ não particionada, cujas buscas requerem apenas uma travessia da raiz até o nível das folhas.

3.5.1 Primeira busca

O processamento da primeira consulta acontece junto com a construção da árvore particionada. Os registros são lidos da localização em que se encontram desordenados, geralmente um arquivo heap, em quantidade suficiente para preencher o espaço de memória disponvel para este fim. Em seguida, são ordenados de acordo com a chave de busca que pode ser escolhida neste momento baseado no predicado da consulta. Neste ponto, paralelismo ou outros artifícios podem ser utilizados para acelerar o processo de ordenação pois o dado está em DRAM. Em seguida o atributo artificial que identifica a partição é adicionado enquanto o dado é escrito no arquivo destinado aos dados indexados e, ao mesmo tempo, o algoritmo de carregamento em massa cria os níveis superiores da árvore. Este processo de responder à primeira busca e construir a árvore B+ particionada é então o mesmo do AM.

Terminada a primeira busca inicia-se o comportamento que distingue a nossa abordagem. Duas estruturas de dados são usadas, a árvore B+ particionada construída durante o processamento da primeira consulta não ganha uma nova partição para onde vão os registros já buscados, como no AM, mas sim uma árvore B+ comum (não particionada) é criada e para ela vão os registros extraídos da árvore particionada. A árvore não particionada, que eventualmente se torna o índice completo é o destino final dos registros e por isto é chamada de *índice definitivo*, enquanto que à estrutura particionada chamamos *índice transitório*, pois existe como parte do processo de construir o índice definitivo.

O uso de duas estruturas para a indexação tem o objetivo de favorecer o paralelismo, agora consideramos de que formas isto ocorre. O índice transitório, sendo uma estrutura temporária, necessita de menos manutenção. Com isto, deixa-se de efetuar verificações por *underflow* e *overflow* nesta estrutura e assim minimiza-se as operações de escrita nela, e consequentemente a contenção de bloqueios. A única escrita que ocorre no índice transitório consiste em marcar como removidos os registros que foram transferidos para o índice final (estratégia *tombstone*). O resultado é que nenhuma operação de *split* ou *merge* de nós da árvore ocorre e diferentes threads podem percorrê-la de forma mais desempedida.

Abdicar de manter o fator de preenchimento do índice transitório por meio de fissões (*split*) ou fusões (*merge*) de nós o transforma basicamente em um índice ISAM, uma estrutura

usada há bastante tempo (CHIN, 1975) (LARSON, 1981) e conhecidamente mais amigável ao paralelismo do que as estruturas B+ sob a ação de tratamento de ocupação. A única seção crítica onde a thread de indexação criará um bloqueio no índice transitório é o período necessário para adicionar os marcadores de exclusão nos registros retirados de lá. Uma operação que pode ser executada de maneira atômica na maioria dos processadores através da instruções especifícas como incremento e recuperação atômico (*atomic add and fetch*), no caso de remoção de uma única entrada.

3.5.2 Bloqueios

Aqui torna-se conveniente uma pequena digressão, a fim de tornar claro o sentido do termo bloqueio no contexto deste trabalho. Duas formas de bloqueio existem em sistemas de bancos de dados. A primeira são os bloqueios que fazem parte dos protocolos de controle de concorrência como o 2PL e servem para proteger itens do banco de dados, tais como registros, páginas ou tabelas. Esta é uma forma forte de bloqueio, que faz parte da arquitetura do sistema e geralmente tem impactos mais significativos na performance. Na literatura de língua inglesa esta forma forte de bloqueio é chamada de *lock*. Bloqueios fortes, controle de concorrência, escalonamento e execução de transações são responsabilidades de níveis que encontram estritamente acima do método de acesso e não são considerados neste trabalho.

A segunda categoria de bloqueios, mais fraca, são aqueles utilizados para proteger estruturas de dados em memória, ou seções críticas no código. Estas formas mais fracas de bloqueio, *latches* na bibliografia inglesa, são detalhes de implementação e oferecem liberdade ao se desenvolver o sistema sobre como implementá-los. Frequentemente, estes bloqueios consistem apenas em *mutexes*, *spin locks* ou, na melhor das hipóteses, operações atômicas nos processadores que oferecem este tipo de instrução. *Latches* são o tipo de bloqueio com que se tem que lidar ao implementar métodos de acesso, e daqui em diante considere-se o termo bloqueio como sinônimo de latch. A tabela 3 resume as diferenças entre os dois tipos de bloqueio.

Tabela 3 – Tipos de bloqueio

	Bloqueio forte	Bloqueio fraco
Separa	Transações	Threads
Protege	Itens de dados	Estruturas em memória
Durante	Toda a transação	Seções críticas
Onde	Gerenciador de transações	Estrutura protegida

Fonte: (GRAEFE *et al.*, 2012)

O principal cuidado que o MetisIDX disperta no que diz respeito a bloqueios é a possibilidade de a thread de indexação bloquear um intervalo de páginas para indexá-la e durante este período de bloqueio uma thread ocupada em processar uma consulta precisar acessar alguma destas páginas e então ter de esperar. Esta é uma possibilidade real, como poderá ser visto nos algoritmos a seguir, neste caso, a performance do método de acesso será a mesma dos índices estritamente adaptativos, pois estes também precisam indexar o intervalo que consultam.

3.5.3 Índice final

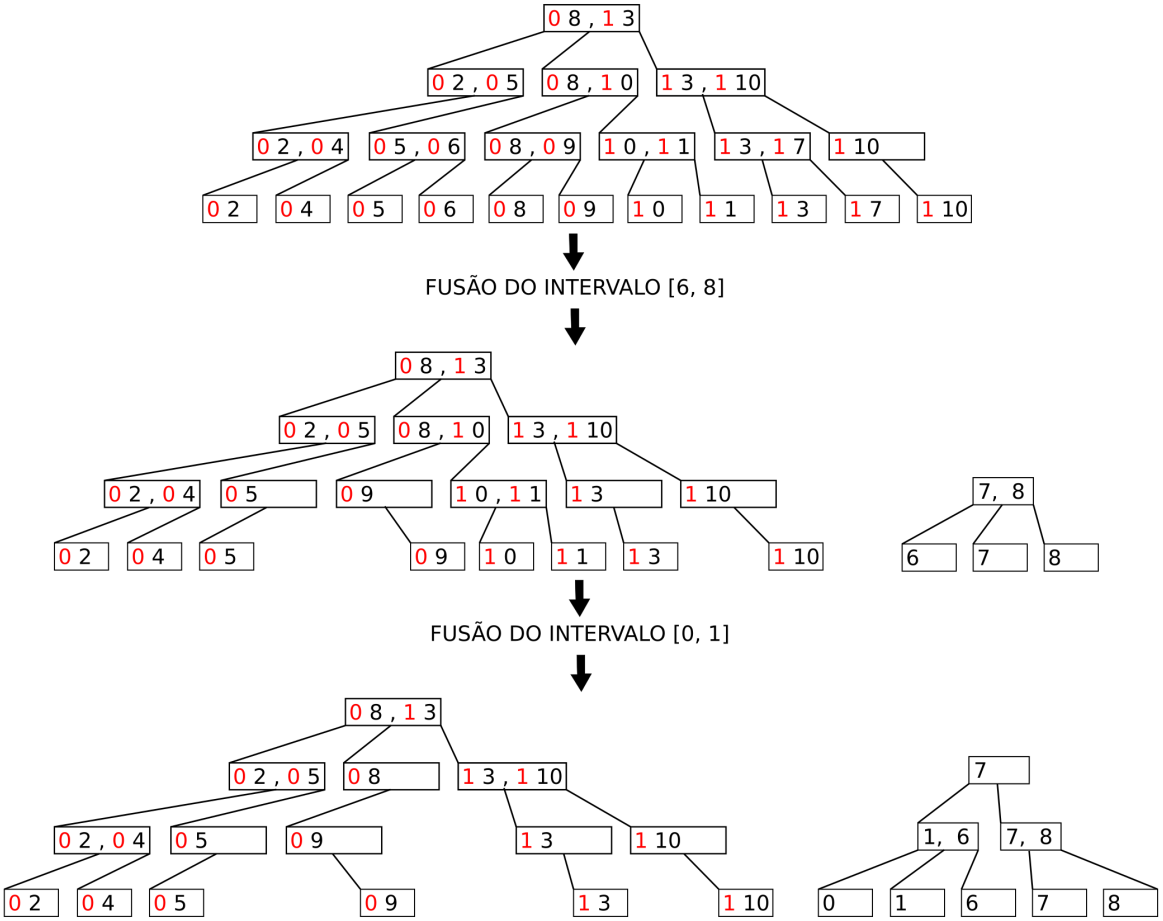
O índice final, que consiste na árvore B+ não particionada para onde vão os registros coletados do índice transitório, por outro lado, passa por ações de fissão e fusão de nós para manter-se fora das condições de *overflow* e *underflow* usuais. Isto é, se N registros cabem em um nó, então considera-se em *underflow* nós com menos que $N/2$ registros, e em *overflow*, nós com N registros. Os bloqueios nesta estrutura são um pouco mais duradores, pois quando a thread de indexação insere um conjunto de registros nesta estrutura, algumas vezes tem de executar as fissões (*splits*) tanto de nós folha quanto de nós índices.

Caso uma thread de consulta precise esperar por uma destas operações, o custo também não será maior so que aquele de um índice adaptativo pois as páginas requisitadas, que estarão indexadas ao final do processo, se encontrarão numa estrutura mais compacta e terão sido recentemente acessadas, aumentando-se as chances de se encontrarem em memória principal. E o processamento da consulta também não terá de executar nenhuma movimentação de dados.

Como exemplo, considere-se um índice transitório cujos nós comportam duas chaves de busca e portanto três ponteiros para nós filhos (esta é a famosa árvore 2 3, caso mais simples de árvore B). A figura 13, em sua parte superior, mostra o estado deste índice transitório. Tal como a árvore mostrada na seção sobre o Adaptive Merging, o número em vermelho representa o atributo artificial inserido durante a execução do algoritmo de carregamento em massa e identifica a partição a que pertence cada valor de chave.

Suponha então que uma operação de *merge* seja executada sobre o intervalo $[6, 8]$. Neste processo, todos os registros do índice transitório que tenham chave de busca neste intervalo (número preto) serão movidos para o índice final, que será criado. Suas posições no índice transitório receberão marcadores de remoção e efetivamente não estarão mais presentes. O resultado final desta operação é mostrado na parte intermediária da figura 13. A parte inferior da mesma figura mostra o estado das estruturas após mais uma busca, desta vez pelo intervalo

Figura 13 – Criação do índice final



$[0, 1]$, a fim de mostrar como o estado de cada estrutura evolui. Em cada operação de indexação os registros são buscados nas partições do índice transitório e inseridos um a um no índice final, possivelmente causando sobrecargas que são tratadas imediatamente por meio de *splits*.

O processo de consultar estas estruturas consiste em percorrer a árvore do índice transitório da raiz até as folhas uma vez para cada partição coletando os registros que lá houverem. Se a busca se destinar a uma operação de indexação não é necessário percorrer o índice final, apenas inserir lá os registros vindos da estrutura transitória. Se tratar-se de uma busca causada por uma consulta, depois de coletar os registros no índice parcial, percorre-se o índice final para coletar os registros que já se encontram lá e que fazem parte da resposta da consulta.

3.5.4 Algoritmos

As buscas em sí têm a mesma complexidade que o processo correspondente no Adaptive Merging, sendo que a travessia da partição final criada no AM é substituída pela travessia do índice final. A diferença de performance vem do fato de a thread de execução da

consulta no MetisIDX não precisar mover dados e do fato de que, na hipótese do processo de predição ter acertado o intervalo a ser consultado, os registros ou boa parte deles estarem no índice final.

Algoritmo 3: Busca nas estruturas de dados

```
Data: Intervalo de chave a ser recuperado
Result: Conjunto de registros com valor chave dentro do intervalo
foreach Partição in Índice transitório do
    travessa_transitorio;
    P = página raiz do índice transitório;
    while P não é folha do
        if P não está bloqueada then
            P = escolhe página filho;
        else
            vá para travessa_transitorio;
        end
    end
    coleta_transitorio;
    if P não está bloqueada then
        while P contém algum registro que satisfaz ao predicado do
            Coleta em P os registros que satisfazem ao predicado;
            P = próxima folha;
        end
    else
        vá para coleta_transitorio;
    end
end
travessa_final;
P = página raiz do índice final;
while P não é folha do
    if P não está bloqueada then
        P = escolhe página filho;
    else
        vá para travessa_final;
    end
end
coleta_final;
if P não está bloqueada then
    while P contém algum registro que satisfaz ao predicado do
        Coleta em P os registros que satisfazem ao predicado;
        P = próxima folha;
    end
else
    vá para coleta_final;
end
```

Se isto ocorrer, mesmo as travessias no índice transitório podem se tornar mais baratas, pois podem ser interrompidas ao encontrar o *tombstone* da chave procurada em um nível intermediário. Este é o caso da chave 8 que se encontrava no primeiro nível acima das folhas na parte superior da figura 13 e já não existe no estado mostrado na parte intermediária da mesma figura.

O algoritmo 3 mostra os passos para a busca de registros nas estruturas de dados. O processo inicia-se pelo índice transitório com travessias sucessivas da raiz ao nível das folhas, uma vez para cada partição. Neste índice, o processo é semelhante a uma busca por chave não prefixo em um índice composto. Em seguida o índice final é percorrido uma única vez, por não ser particionado.

Múltiplas threads de consulta podem estar ativas, correspondendo a diferentes transações. O controle e concorrência dedicado a garantir a consistência dessas buscas concorrentes é resopnsabilidade do gerenciador de transações e do escalonador de tarefas. As verificações por bloqueios que aparecem no algoritmo são bloqueios fracos que visam tornar o sistema livre de condições de corrida relativas à interação entre a thread executando a consulta e a thread dedicada a construção do índice final. Na forma como está escrito o algoritmo 3, quando um bloqueio é encontrado, entra-se em loop verificando o bloqueio até que ele esteja disponível. Esta é uma situação de espera ocupada, pois a CPU continua processando este loop até que o algoritmo possa proseguir. Em uma implementação de produção é mais aconselhável fazer a thread dormir por algum tempo antes de verificar o bloqueio, assim o núcleo de processamento pode ser escalonado para atender a outra tarefa enquanto a página requerida está bloqueada, fazendo-se assim melhor uso dos recursos do sistema.

Como antecipado, a thread que processa uma consulta deve buscar por registros no índice transitório primeiro e, em seguida, no índice final, esta ordem é importante pois se não seguida pode resultar em respostas erradas. A situação onde isto ocorre é a seguinte: suponha-se que a busca se inicie pelo índice final e em seguida volte sua atenção para o índice transitório. Quando isto ocorre, é possível que a thread de indexação tenha bloqueado registros que pertencem ao conjunto solução da consulta. Neste caso, a thread de consulta espera que a tarefa de indexação termine e em seguida continua a busca, mas agora os registros não estão mais lá, foram movidos para o índice final. E como o índice final já foi escaneado, estes registros não serão retornados na resposta.

O algoritmo 4 descreve a atividade da thread de indexação, apenas uma thread é usada

para este fim. O procedimento começa verificando se já há consultas observadas o suficientes para executar uma atualização do modelo e caso afirmativo executa-se o treinamento da rede neural. Caso negativo, utiliza-se a rede para predizer o intervalo a ser solicitado em seguida e executa-se a transferência dos registros neste intervalo para o índice final. Este procedimento se repete ininterruptamente até que o índice transitório tenha sido exaurido. Sendo a indexação independente do processamento de consultas, várias execuções deste algoritmo podem ocorrer durante o processamento de uma única transação ou caso o sistema passe por períodos de pouca demanda ou ociosidade. Este último caso é mais uma vantagem do MetisIDX em relação a índices adaptativos, pois estes param de melhorar a estrutura de acesso se as consultas parem, enquanto nossa estratégia continua indexando intervalos não consultados até eventualmente alcançar o estado de índice completo.

Algoritmo 4: Predição e Indexação

```
Data: Predicados das últimas N consultas
Result: Indexação contínua dos intervalos preditos
while O índice transitório não for exaurido do
    if Pelo nemos N consultas foram observadas then
        treina a rede neural (atualiza o modelo);
        limpa a pilha de consultas observadas;
    else
        indexa_intervalo;
        usa a rede neural para prever o próximo intervalo;
        if intervalo não está bloqueado then
            adquire bloqueio no intervalo de folhas;
            move registros para o índice final;
            libera o bloqueio;
        else
            vá para indexa_intervalo;
        end
    end
end
```

No algoritmo 4 não há espera ocupada, pois quando a execução verifica que o intervalo a ser indexado está bloqueado por alguma thread que executa consultas, não espera que seja desbloqueado mas sim tenta indexar um outro intervalo predito pelas redes neurais.

Apenas uma thread é usada para indexação, é possivel usar multiplas threads mas é preciso ponderar sobre os benefícios. Se o sistema tiver que servir a multiplos clientes pode ser preferível dedicar a maioria das threads à tarefa de processar às consultas. Além disto, múltiplas

threads executando movimentação de dados vão competir por vazão no gerenciador de buffer e nos dispositivos de armazenamento mais lentos, e também bloquearão multiplos intervalos de chave ao mesmo tempo competindo com o processamento das consultas. Embora este trabalho não avalie o cenário de múltiplas threads de indexação, é aparente que esta abordagem traz poucas vantagens.

Como adiantado, algumas oportunidades de ganho de performance existem ao se transformar os algoritmos apresentados anteriormente em implementações da técnica em um sistema de gerenciamento de dados real. As mais diretas e mais significativas são o uso de um fluxo de trabalho assíncrono a fim de evitar esperar por bloqueios e por transferências demoradas. Esta otimização não favorece uma dada busca, mas pode melhorar significativamente a vazão de um sistema que atende a múltiplos clientes por manter os núcleos de processamento ocupados apenas com trabalho útil.

4 EXPERIMENTOS

Uma avaliação experimental da capacidade do MetisIDX de acelerar as buscas foi realizada seguindo procedimentos semelhantes aos que foram executados para os índices adaptativos. Os experimentos conduzidos nos trabalhos originais anteriores, apresentados como resultados das estratégias adaptativas, utilizaram cargas de trabalho sintéticas compostas de seguidas consultas por intervalo. Os dados utilizados nestes trabalhos se compõem de 10 milhões de valores inteiros, que são utilizados como chave de busca.

MetisIDX e *Adaptive Merging* foram ambos implementados como componentes do MetisDB para fins de comparação. A inserção de tais métodos de acesso em sistemas de gerenciamento de bancos de dados completos seria valorosa a fim de avaliar integralmente o comportamento deles sob o efeito de cargas de trabalho mais complexas e advindas de aplicações reais ou de *benchmarks* já padronizados, em particular o TPC-H já que as estratégias se destinam a contextos mais próximos de OLAP. Podendo-se assim comparar também com abordagens não adaptativas.

No entanto, como apontado pelos autores do *Adaptive Merging* (GRAEFE; KUNO, 2010), os SGBDS tradicionais fazem uma distinção muito rígida entre consultas e atualizações de índices, e exigem que as primeiras só façam acessos de leitura. A introdução de métodos de acesso que incluem movimentação de dados a cada consulta, como é o caso dos índices adaptativos e do MetisIDX, requer modificações em diferentes componentes do sistema. Com isto, a implementação no núcleo de sistemas tradicionais, embora factível, representa um esforço maior que a recompensa. Isto se dá por que tais estratégias foram pensadas para sistemas cuja estrutura interna seja inteiramente pensada para receber componentes adaptativos, e tal necessidade ainda está em aberto. E foi uma das motivações para o início do desenvolvimento do MetisDB.

Dado que a nossa estratégia não é um índice estritamente adaptativo, pode ficar aparente que a melhor comparação seriam estratégias como o *Holistic Indexing* (PETRAKI *et al.*, 2015), que também vai além de indexar apenas o intervalo de chave da consulta atual. No entanto, estes trabalhos são todos extensões do *Database Cracking* e, sendo assim, são destinados a sistemas com armazenamento baseado em colunas e com os dados residentes em memória principal. O MetisIDX se propõe a melhorar o acesso a dados armazenados em forma de registros completos e residentes em dispositivos secundários endereçados a nível de blocos. Por isto a comparação mais direta é com o *Adaptive Merging*, com o qual também compartilhamos a

estrutura de dados (árvore B+ particionada). Por estes motivos esta estratégia foi tomada com referência para os algoritmos de busca e construção, e também para comparação de performance.

O hardware utilizado nos experimentos se compõe de um disco rígido Seagate de 7200RPM com sistema de arquivos formatado segundo o esquema EXT4, um processador Intel i5 com 8 núcleos de 3.10GHz, e duas memórias Kingston DDR4 de 4GB cada. O ambiente de software consiste de um sistema operacional Debian GNU/Linux 9.3 com compilador C++ GNU 6.3.0 com o qual o MetisDB foi compilado usando usando o padrão C++14 sem otimizações.

4.1 Conjuntos de dados

Dois conjuntos de dados foram utilizados nos experimentos, ambos compostos de registros contendo um número inteiro de 64 bits usado como chave de busca e indexação e um campo textual de 64 bytes contendo caracteres aleatórios, adicionado com o intuito de dar volume. A tabela 4 mostra as dimensões destes conjuntos de dados.

Tabela 4 – Conjuntos de dados

Conjunto de dados	Tamanho	Número de registros
D1	500MB	5.106.208
D2	50GB	727.483.873

Dois tamanhos foram utilizados para observar o efeito da extensão dos intervalos no processo e também a diferença de comportamento entre um conjunto de dados que cabe inteiramente na memória principal (500MB) e um conjunto de dados que precisa passar ativamente por gerenciamento de *buffer* (50GB). Note-se que mesmo estando aquém dos volumes citados para algumas das aplicações atuais de maior escala, estes conjuntos ainda são equivalentes ou maiores que os utilizados na avaliação dos índices adaptativos (10 milhões de valores inteiros $\approx$ 610MB).

Estes registros foram inicialmente gerados e gravados em arquivo não ordenado (HEAP). A disposição dos registros no arquivo foi a de páginas de 8KB. Cada uma destas páginas recebeu um preenchimento aleatório entre 50% e 100%, porcentagens estas que representam a fração entre o número de registros que cabem em uma página e o número que de fato foi gravado. A distribuição usada para gerar os preenchimentos foi a uniforme, por isto, o arquivo *heap* ficou efetivamente 75% ocupado. Este arquivo só é acessado uma vez, durante a primeira busca, por que durante o processamento desta busca todos os registros são transferidos para um outro arquivo contendo o índice transitório.

O arquivo contento o índice transitório também é organizado em páginas de 8KB mas, devido a se saber de antemão que não haverá inserção de novos registros nesta estrutura, o fator de preenchimento utilizado é 100%. Esta estrutura ser compacta é conveniente, tanto por ser ela um elemento temporário, quanto pelo fato de conter em cada registro o atributo extra da árvore B+ particionada. Já o índice final, que fica em um terceiro arquivo, tem também suas páginas de 8KB correspondendo a nós da árvore B+, mas neste usa-se também fator de preenchimento igual á 75%, pois em cenários com inserções o índice final é a localização mais conveniente para recebê-los.

4.2 Consultas

A carga de trabalho a que os dois métodos de acesso foram submetidos consiste em 1000 buscas por intervalos simples. Em SQL, estas buscas tem a forma

`SELECT ... WHERE A>=`Q_{LOW} `AND A<=`Q_{HI}`;`

Q_{LOW} e Q_{HI} são os valores mínimo e máximo da chave de busca que a consulta admite respectivamente. E são também os valores ϕ e ψ das funções que devem ser aprendidas pelas ELM.

Os trabalhos anteriores em índices adaptativos selecionavam valores aleatórios a cada vez para Q_{LOW} e Q_{HI}. Na experimentação do MetisIDX no entanto, esta abordagem não é adequada pois a maneira como a técnica atua é aprendendo padrões nestes valores. Como discutido anteriormente, a hipótese de que a carga de trabalho é desconhecida precisa ser mantida, mas é razoável considerar que as consultas disparadas por aplicações reais não sejam aleatórias e que padrões subjacentes existam. Por este motivo, nos experimentos aqui descritos, uma dependência funcional bem definida foi utilizada para estes limites. A forma desta dependência é uma função quadrática. Logo, a expressão que gera os intervalo a serem utilizados em cada busca é dado pela equação 4.1 a seguir

$$Q(j) = M \cdot (j/C)^2 \tag{4.1}$$

onde M é o valor máximo da chave de busca presente nos dados e C é o número total de buscas a serem efetuadas. Esta distribuição quadrática das buscas corresponde a uma varredura sobre o conjunto de dados com leituras localizadas cujo espaçamento aumenta ao longo da sequência. Alguma aleatoriedade foi adicionada a esta carga através do acrécimo de um ruído gaussiano aos limites dos intervalos consultados. A forma final para os limites das consultas, Q_{LOW} e Q_{HI} é

então

$$\begin{aligned} Q_{LOW}(j) &= Q(j) - R_1 \\ Q_{HI}(j) &= Q(j) + R_2 \end{aligned} \tag{4.2}$$

onde R_1 e R_2 são valores gerados a partir de uma distribuição normal com valor médio igual a zero e desvio padrão igual a 0.5% da extensão do domímio da chave. Um caso particular pode ocorrer onde a predição da rede afirma que a busca futura solicitará um intervalo grande o bastante para causar a indexação de um intervalo muito grande ou mesmo do dado inteiro. Neste caso, um índice adaptativo só retornaria a resposta para a aplicação após ter indexado o intervalo inteiro, o que no caso extremo de o intervalo abranger o dado inteiro equivale ao custo de criar um índice completo e executar a busca sobre ele. Nossa estratégia pode, mais uma vez usando da flexibilidade de indexação e consulta serem independentes, escolher uma política de mais economia de operações de entrada e saída. A política escolhida a princípio é a de indexar a cada vez no máximo a quantidade de registros que cabem na memória, além disto. Com isto, uma busca não precisa esperar uma atividade de indexação reorganizar todos os registros da resposta para retornar para o usuário.

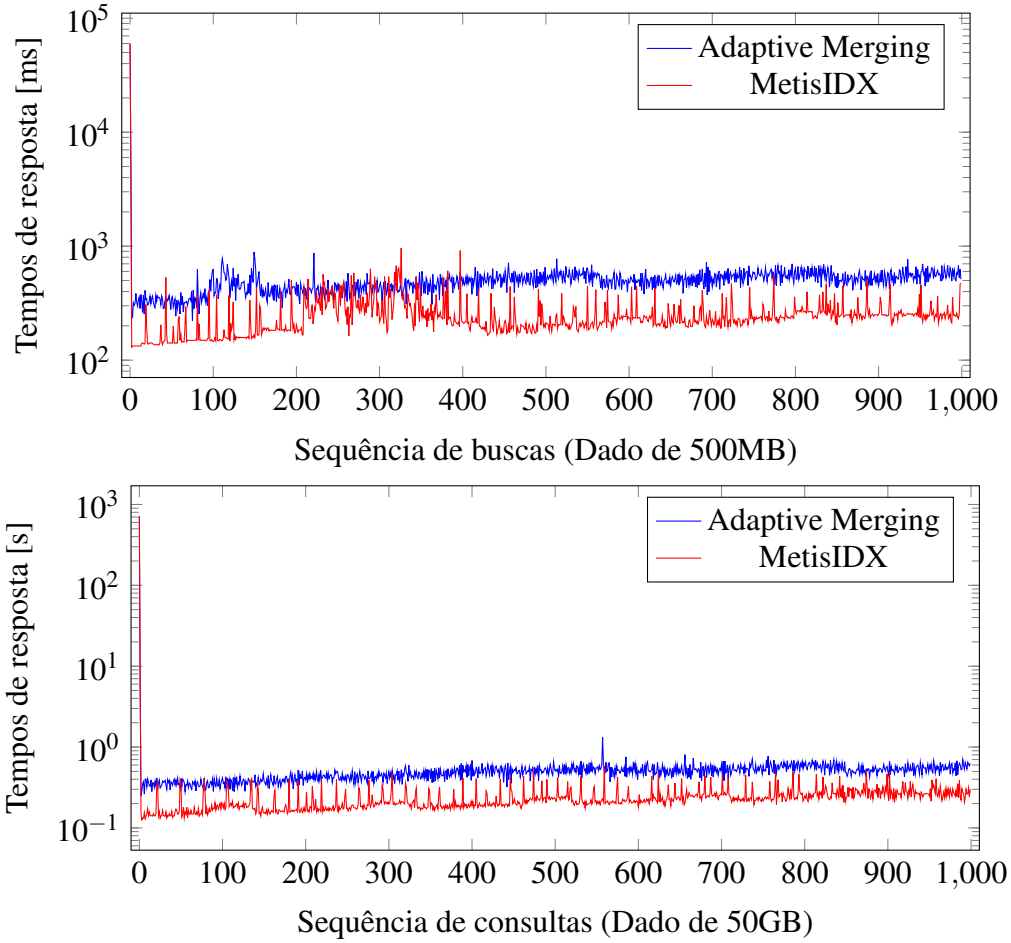
4.3 Tempos de busca

No contexto de processamento de consultas sobre estruturas em disco é comum utilizar-se os número de páginas lidas e escritas como métrica de performance primária, enquanto tempos de resposta são indicadores menos importantes, apesar de esta ser a quantidade observada pelas aplicações ou clientes. No entanto, quando a indexação ocorre em paralelo e de forma independente do processamento das buscas, o número de transferências deixa de refletir a ideia de "rapidez"no processamento da carga de trabalho. Isto se dá pelo caráter preditivo da construção da estrutura. Isto é, a quantidade de páginas trocadas pela thread de indexação não se reflete no tempo de consulta pois frequentemente ocorrem, pelo menos em parte, antes da consulta ser processada.

O gráfico na figura 14 mostra os tempos de resposta da mesma sequência de 1000 buscas para o *Adaptive Merging* e para o MetisIDX. A parte superior corresponde à massa de dados de 500MB enquanto o gráfico abaixo mostra os resultados relativos ao conjunto com 50GB. Os eixos X mostram a sequência com que as buscas foram disparadas e o eixo Y o tempo de resposta em escala logarítmica. Em milisegundos no caso dos dados de 500MB e em segundos no caso dos dados de 50GB.

A característica mais marcante destes gráficos é a diferença entre os tempos necessários para o processamento da primeira busca e as demais. O motivo é que nas duas técnicas o processamento da primeira consulta é dependente da construção do índice transitório inteiro, por isto, é a única consulta que necessariamente acessa todas as páginas, folha ou não, das estruturas.

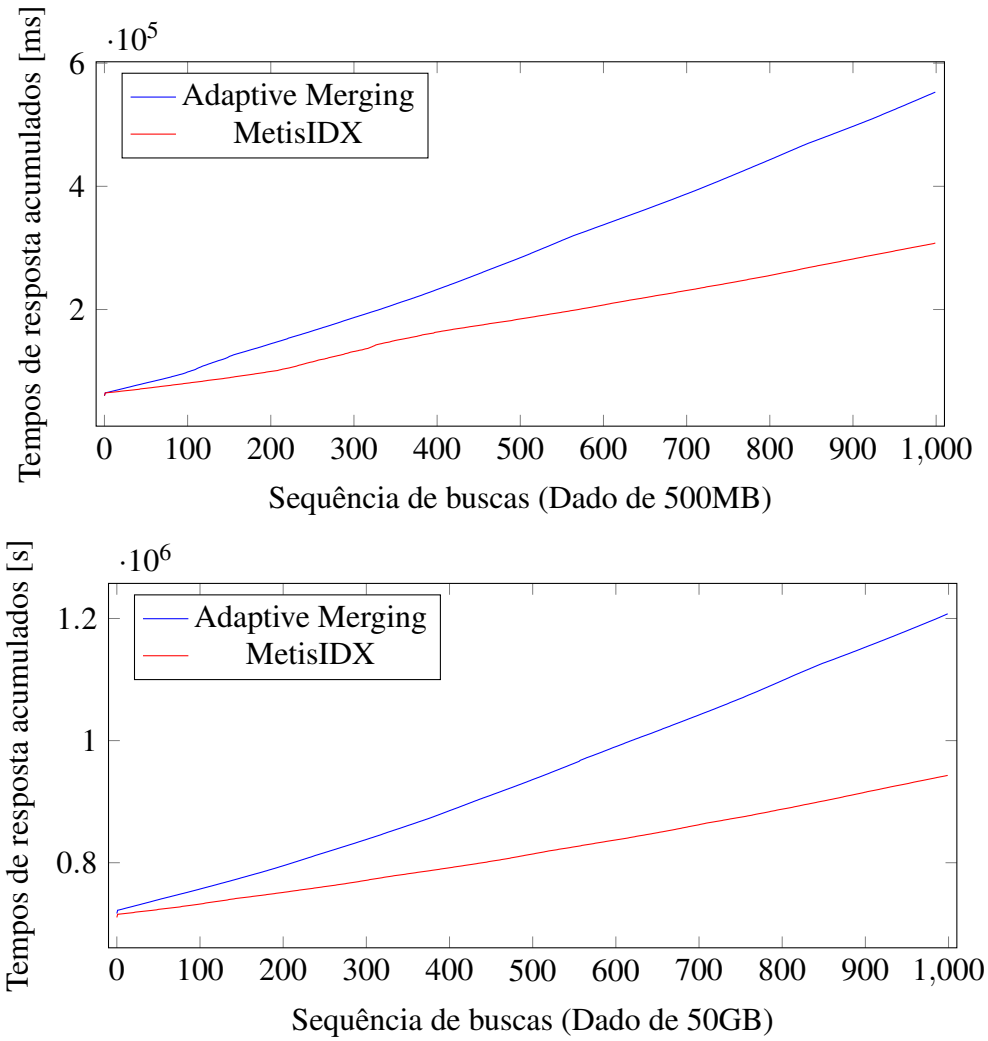
Figura 14 – Tempos de resposta



O gráfico do experimento com 500MB tem mais ruído enquanto o de 50GB tem um comportamento mais consistente. Isto se dá pelo fato de que as buscas no primeiro são muito rápidas e por isto fatores externos afetam de forma significativa o tempo de resposta. O principal destes fatores externos é o escalonamento de processos por parte do sistema operacional, pois o tempo que o SO dedicar de uma *thread* física para outro processo e a troca de contexto se reflete no tempo da busca. Mesmo estando a máquina dedicada para o experimento e que os demais processos se limitem a componentes do sistema (*daemons*) apenas o tempo da troca de contexto já tem ordem de grandeza próxima dos tempos das buscas, que neste caso é de décimos de segundo.

Os resultados com o conjunto de dados de 50G tem um comportamento mais regular,

Figura 15 – Tempos para se responder a N buscas



por que os tempos necessários para se executar as buscas são grandes o bastante para que as interferências de outros processos sejam desconsideradas. Deste gráfico, é possível observar que os tempos do MetisIDX são sistematicamente mais baixos, com exceção de algumas buscas que têm tempos semelhantes ao do *Adaptive Merging*.

Nestes casos, o que ocorre é que a *thread* de busca acompanha a *thread* de indexação, isto é, um intervalo é buscado enquanto ainda está sendo indexado. Isto ocorre quando a *thread* que coleta os registros pertencentes à resposta precisa esperar que o processo de indexação libere o intervalo de páginas folha desejado. Esta é a única situação onde uma *thread* que executa buscas precisa esperar e o custo, neste caso, é idêntico ao das buscas numa estratégia adaptativa. Este aspecto também pode ser observado nos gráficos da figura 14, quando há um pico no tempo de busca da nossa estratégia, este fica no mesmo nível que o *Adaptive Merging*.

4.3.1 *Ganho cumulativo*

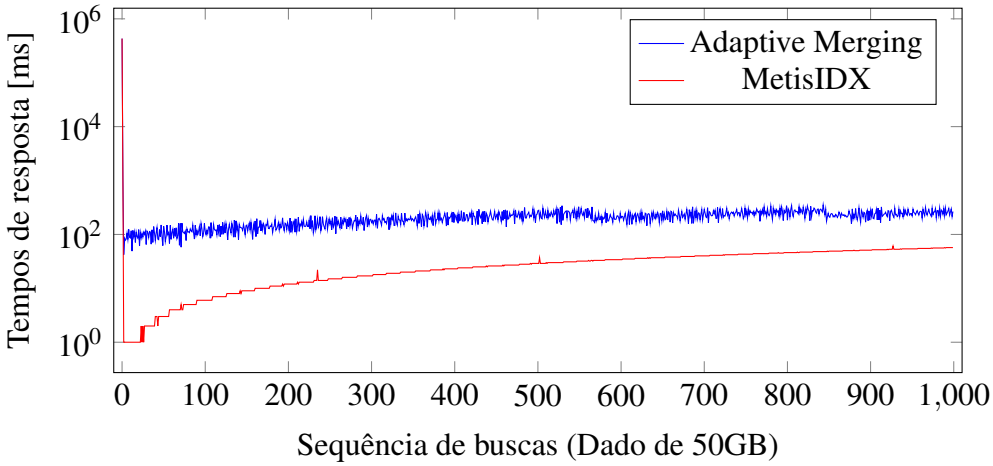
O ganho de tempo proporcionado pelo uso da técnica MetisIDX se torna mais aparente se observarmos a diferença acumulada em relação à estratégia de referência. Os gráficos na figura 15 mostram esta diferença para os dois conjuntos de dados discutidos anteriormente. O que estes gráficos expressam é o tempo necessário para se responder a N consultas como uma função de N.

Pode ser observado que a diferença na performance das estratégias é aproximadamente linear, logo, quanto mais buscas forem executadas, mais vantagem o MetisIDX apresentará. Após as usuais 1000 consultas, a diferença já é de 31% de ganho em relação ao *Adaptive Merging*. Esta vantagem tem um limite, uma vez alcançado o estado de índice completo as duas técnicas se tornam idênticas. Neste estado, o índice transitório está vazio e todas as buscas consistem em uma travessia da árvore B+ não particionada do índice final. Com os custos usuais de índices completos.

4.3.2 *Meios de armazenamento diferentes*

A avaliação de desempenho de consultas semelhantes com os dados armazenados em meios com propriedades diferentes é interessante a fim de se poder inferir mais detalhes a respeito do comportamento de eventuais sistemas implementados com a nossa estratégia. Em uma nova execução das mesmas consultas da seção anterior sobre o conjunto de dados de 50GB, um SSD foi utilizado como dispositivo armazenador para todos os arquivos, isto é, para armazenar o arquivo heap onde os registros inicialmente residem, e para armazenar os índices parcial e final.

Figura 16 – Tempos de resposta no SSD



Este dispositivo, um Intel P3600 com capacidade para 250GB, tem a característica de apresentar acentuada assimetria entre leituras e escritas, sendo as primeiras aproximadamente 10 vezes mais rápidas. Outra diferença importante em relação ao HDD é que, como em todo SSD, os acessos em ordem aleatória às páginas de dados têm essencialmente o mesmo custo que acessos sequenciais, embora escritas sequenciais ainda sejam ligeiramente mais eficientes que escritas aleatórias.

O gráfico na figura 16 mostra os tempos de resposta para essa sequência de 1000 buscas. A forma geral do gráfico é a mesma daquele resultante do experimento com o HDD. Uma diferença é que os tempos de resposta são menores, pelo fato de o dispositivo ser mais rápido.

Uma distinção adicional reside no fato de as primeiras buscas apresentarem uma diferença maior entre o MetisIDX e o *Adaptive Merging*, devida a não haver mais perdas devidas à aleatoriedade dos acessos. Durante esta primeira parte da sequência, é possível observar também um aspecto de degraus na curva, é provável que o motivo seja a manutenção da tabela de endereçamento das páginas em memória. Esta tabela ainda está crescendo e por vezes precisa passar por um novo espalhamento (*rehash*) de algum *bucket*. *Hashing linear* é usado para esta tabela, provido pelo objeto `std::unordered_map` da biblioteca padrão da linguagem C++. Isto sugere a conclusão de que a este ponto, com a rapidez das transferências entre o SSD e a DRAM, o processamento já começa a se tornar o gargalo. Se este for o caso, uma maneira de resolver este problema é utilizar uma implementação de tabela mais eficiente para esta aplicação, isto é, uma tabela de espalhamento de tamanho estático. Pois o número máximo de páginas mantidas em memória é conhecido.

4.3.3 Avaliação do processo preditivo e localidade dos registros

Os resultados apresentados anteriormente se concentram na medida de performance mais diretamente observada pelas aplicações, o tempo de resposta. Isto se justifica pelo fato de esta métrica carregar os efeitos de todos os fatores envolvidos no funcionamento do sistema, incluindo transferências, concorrência e compartilhamento de recursos. Inclui também a taxa de acerto das predições das redes neurais, embora uma avaliação deste fator, que é uma das principais distinções do nosso método seja um pouco mais sutil, pois influencia mas não determina sozinho a performance.

A maneira usual de se avaliar o desempenho de algoritmos de aprendizagem de

máquina é através de validação cruzada. No entanto, a aplicação da ELM na nossa técnica tem como objetivo modelar um intervalo, e o faz através do aprendizado de duas funções que correspondem aos extremos deste intervalo, assim erro ou acerto não é uma decisão binária. Se os extremos não coincidirem exatamente mas houver sobreposição e uma parte dele for indexada, a performance já será beneficiada. Além disto, o processo de aprendizado é contínuo, assim não há estado "pronto" ou "treinado" do modelo para que se possa aplicar a um conjunto de dados de validação.

Outro ponto que faz a validação cruzada pouco útil neste experimento é o uso de funções com comportamento conhecido, como a função quadrática, e é fato conhecido que a ELM pode modelá-las (HUANG *et al.*, 2012). Uma quantidade que tem uma relação com o desempenho e também com o processo preditivo é a razão do número de páginas encontradas pela thread que executa as buscas em memória principal pelo total de buscas por páginas, ou seja, o *cache hit rate*. Estes números, para as duas técnicas a partir da segunda busca, aparecem na tabela 5 e correspondem ao conjunto de dados de 50GB.

Tabela 5 – Cache hit rates no conjunto de 50GB

Técnica	*Hit rate*
Adaptive Merging	32%
MetisIDX	58%

Os valores que aparecem nesta tabelas são globais, isto é, dão os *hit rate* de todas as solicitações de páginas para os índices transitório e final, e para páginas folha e páginas índice. O valor para o AM é razoável, pois considerando as buscas efetuadas, que vão todas solicitar intervalos ainda não solicitados, cada uma acessará 6 páginas índice (altura da árvore B+ particionada) que estarão na maioria das vezes em memória por serem acessadas frequentemente. Em seguida uma ou duas dezenas de páginas folha, contendo os registros, serão acessadas. Com isto é possível estimar-se que o hit rate fique em torno de 30% como observado. A taxa de *cache hit* para o conjunto de dados de 500GB não tem significância por que estes dado cabem completamente na memória principal e por isto todas as páginas são trazidas para o cache quando são requisitadas pela primeira vez e nas demais sempre estarão presentes. 2GB de memória foram utilizados como capacidade para o cache.

A localidade dos dados mais elevada do MetisIDX é a fonte primária da melhoria da performance e consequência direta do comportamento preditivo. A política de substituição

de páginas utilizada é o *Least Recently Used* (LRU) a fim de priorizar páginas recentemente acessadas. Com isto, quando a rede acerta o intervalo a ser consultado, ou parte dele, este intervalo é indexado e neste processo trazido para a memória. Em seguida, quando consultado, os registros estarão em sua posição final com acesso de índice completo e também em memória principal. No caso de um intervalo distinto ser previsto, este será indexado, trazendo para a memória registros diferentes daqueles de interesse e este comportamento de *pré-fetching* não se observará. Por isto, os valores relativos ao MetisIDX na tabela 5 podem ser tomados como indicativo da taxa de acerto nas predições.

A possibilidade de utilizar outros algoritmos de aprendizagem de máquina existe, observando-se as restrições listadas da subseção 3.3.2, MetisIDX não precisa ser implementado necessariamente com uma ELM como mecanismo de regressão. Na avaliação experimental, a ELM foi escolhida por se encaixar nestas restrições e particularmente por se adequar a aprendizado online com atualização de modelo de baixo custo.

5 CONCLUSÕES E TRABALHOS FUTUROS

A ideia principal e noção motivadora por trás do MetisIDX é a de que adicionar comportamento preditivo é o próximo passo natural na evolução das estratégias de indexação adaptativa. Isto é possibilitado pela capacidade de construir as estruturas de dados através de operações incrementais onde cada uma tem baixo custo por ter curta duração. Guiar estas operações por meio de algoritmos de aprendizado para mover o foco da requisição atual para o padrão emergente da carga de trabalho faz sentido, por permitir ao sistema se preparar com alguma antecedência para as demandas vindouras sem sacrificar o caráter orientado à carga, continuando capaz de adequar-se de forma autônoma caso a demanda mude e sem fazer suposições prévias sobre ela.

Uma generalização desta ideia para outros componentes de um SGBD motivou também a idealização de um novo sistema ainda em desenvolvimento, o MetisDB. Este se propõe a lançar as bases de uma arquitetura de gerenciamento de dados onde cada etapa do processamento de consultas gera conhecimento, que é igualmente explorado por todos os componentes no planejamento de suas ações. Em uma analogia alegórica, isto corresponde a ver o sistema de gerenciamento de bancos de dados como um organismo vivo e que precisamos adicionar a ele um sistema nervoso, a fim de dar-lhe consciência de si próprio, de suas partes e de sua função. Tal ideia, da qual este trabalho é apenas um passo e que parece ser compartilhada por outros membros da comunidade (PARK *et al.*, 2017), trás a ciência e a técnica de gerenciar armazenamento e processar consultas para a contemporaneidade. Com estes desenvolvimentos, a área se torna par com várias outras, que estão se enveredando pelos caminhos de construir sistemas autônomos e capazes de aprender com a experiência ao abraçar a inteligência artificial.

5.1 Contribuições

Além do avanço de paradigma para os índices adaptativos, nossa técnica possui pontos particulares de melhoria com respeito às anteriores, que chamamos de estritamente adaptativas. A primeira delas, necessária para que um mecanismo externo possa guiar a construção do método de acesso, é o desacoplamento das operações de indexação do processamento das consultas. Com isto, a escolha dos intervalos de chave a serem indexados se torna livre das consultas que estão sendo processadas. A partir deste ponto, pode se focar em beneficiar às consultas futuras.

Flexibilidade é outra característica interessante do MetisIDX, índices adaptativos advogam que é um mérito indexar apenas os registros que de fato são tocados pelas consultas evitando empreender esforço de indexação com dados de menor interesse, e isto é verdade para a maioria das aplicações de exploração. Outros casos há, no entanto, em que é interessante atingir o índice completo, adicionando eventualmente todos os registros na estrutura final do método de acesso. Para estes cenários é possível fazer com que a *thread* de indexação do MetisIDX continue a executar mesmo em períodos de ociosidade ou menor carga do sistema, a fim de construir um índice completo.

O esquema apresentado para o MetisIDX pode ser adaptado para ter qualquer um dos dois comportamentos. Para a primeira opção, a de indexar apenas o que é buscado, basta adaptar o algoritmo 4 de forma a executar apenas uma operação de *merge* para cada busca processada. Para a abordagem alternativa, isto é, esforçar-se para atingir o índice completo, pode-se disparar continuamente ações de indexação. Neste último caso, pode ser necessário incluir fusões de intervalos aleatórios, dentre aqueles ainda não indexados, para incluir os registros não requisitados ainda. Isto se deve ao fato de que a saída do mecanismo de regressão pode não incluir todos os intervalos de chave, e também pode não haver novas consultas para atualizar o modelo.

O caráter preditivo do MetisIDX é capaz de acelerar o acesso às sequências de registros buscados por aplicações de exploração por permitir que o processamento das consultas acessem registros que se encontram todos, ou em parte, em estruturas de índice eficientes. As buscas também se tornam mais céleres por que a *thread* que realiza a busca não precisa efetuar movimentação de dados, pois esta atividade é de responsabilidade da *thread* dedicada a atualizar o modelo e construir o índice.

5.2 Resultados

A experimentação realizada mostra que o emprego da estratégia MetisIDX proporciona melhora de desempenho em relação a esquemas de indexação adaptativa estado da arte. Em particular, uma comparação com o *Adaptive Merging* foi efetuada pelo fato de a estrutura de dados usada ter aspectos em comum com esta estratégia e os dispositivos de armazenamento que as duas têm em mente são do mesmo tipo. Após as 1000 consultas por intervalo de chave de busca usuais utilizadas para teste na maioria dos trabalhos sobre índices adaptativos conseguimos um ganho de cerca de 30% em tempo. As consultas individuais foram mais rápidas na larga

maioria das vezes e aquelas que demoraram mais do que as demais tiveram tempos basicamente idênticos ao da estratégia de referência.

A construção da estrutura de dados anterior ao tempo de busca faz com que parte dos registros, ou até mesmo todos eles em algumas situações, sejam encontrados em camadas mais elevadas da hierarquia de memória quando buscados, pois é necessário carregá-los para a memória principal para indexá-los. Nos experimentos descritos anteriormente, isto pode ser notado através da maior taxa de *cache hit*. Neste caso, o acesso a dados na camada de armazenamento secundário (endereçado a nível de blocos e muito lento) é poupado, pois um subconjunto dos registros foi previamente levado para o *buffer* em memória DRAM. O LRU como política de substituição de páginas é adequado por que permite que as páginas recentemente carregadas pela *thread* de indexação permaneçam até serem requisitadas por uma *thread* de busca.

Um aparante gargalo de processamento, devido ao gerenciamento de *buffer*, começa a aparecer se utilizado um dispositivo de armazenamento que forneça acesso mais rápido em relação a discos rígidos. Este fato não contradiz a hipótese de núcleos de processamento serem um recurso frequentemente disponível, pois o gargalo no presente caso diz respeito ao desempenho de uma *thread* e não da quantidade de *threads* disponíveis. Muitas oportunidades de otimização de desempenho existem no entanto, que podem fazer com que o gargalo volte a ser as transferências. E a principal delas é justamente o uso de paralelismo em diferentes etapas do processo, em particular na ordenação dos dados executada durante o processamento da primeira consulta a fim de construir o índice transitório. Assim, a construção da estrutura de acesso baseada em processo preditivo, que é também um *pre-fetching*, aliada a um meio de armazenamento amigável a acessos aleatórios, é uma opção para esconder a lentidão ocasionada pelo uso de armazenamento secundário, além de fornecer resultados melhores que as técnicas estritamente adaptativas.

5.3 Perspectivas e sugestões de pesquisa futura

Os princípios utilizados na elaboração da nossa técnica de indexação constituem contribuição na construção de estruturas de dados em métodos de acesso para aplicações de exploração de dados. Mas sua aplicabilidade não se restringe a este nicho. Aprendizado contínuo em mais etapas do funcionamento do sistema pode oferecer outras melhorias de performance e ainda amplificar aquelas trazidas pelo MetisIDX.

A diferença nas ordens de magnitude dos tempos necessários para se acessar registros nos diferentes níveis da hierarquia de memória sempre fizeram com que economizar transferências ou executá-las antecipadamente fosse o alvo primário de otimizações para os SGBDs. Uma contribuição valorosa para estes problemas, que usa a ideia de aprendizado contínuo, é dedicar um agente a manter atualizado um modelo do padrão de transferências e efetuá-las baseadas nas predições do modelo na esperança de concluí-los antes de as aplicações requisitá-los, tal como a thread de indexação faz para os intervalos de chave de busca no MetisIDX.

Em um sistema que conta com muitos níveis desta hierarquia, como fitas, HDDs e memória, um modelo poderia sugerir quais fitas deverem ser trazidas para a leitora e até onde devem ser rebobinadas para atender às próximas consultas. Este esquema, em conjunto com o uso de braços mecânicos para a movimentação dos rolos de fita, pode proporcionar ganhos de performance dos mais significativos. E isto é verdade mesmo se a taxa de acertos for baixa para os padrões da comunidade de aprendizagem de máquina (menor que 50%), porque nos casos onde acertos ocorrerem, o passo mecânico do processo é poupado. E são estas operações mecânicas necessárias a alguns dispositivos que constituem os maiores custos de desempenho.

Acesso concorrente também é uma possível área a ser abordada com as ferramentas do aprendizado automático. Suponha-se, por exemplo, que um esquema de construção de estrutura de dados semelhante ao MetisIDX seja utilizada com múltiplas conexões de clientes solicitando intervalos de chave de busca diferentes. Neste caso é mais benéfico aprender um modelo que prediga quais intervalos de chave serão buscados por mais de uma aplicação ao longo do tempo, e então indexar estes intervalos. Com isto, múltiplos clientes serão beneficiados com o *pre-fetching* efetuado pela *thread* de de indexação, e com o isto, o *throughput* do sistema é beneficiado ainda mais.

Cuidados foram tomados a fim de minimizar a contenção de bloqueios por parte da *thread* de indexação ao priorizar a *thread* de busca fazendo-a esperar apenas no caso em que o intervalo de chave requisitado por ela estiver sendo indexado. Porém, quando mais de uma *thread* acessa uma árvore B+, e pelo menos uma delas efetua operações que não são somente de leitura, sempre há concorrência. O caso ideal para estratégias como o MetisIDX é o uso de estruturas de dados livres de bloqueios, como a Bw-Tree (LEVANDOSKI; SENGUPTA, 2013), baseada na árvore B+. Estas estruturas têm prometido acesso por múltiplas *threads*, sem a necessidade de bloqueios e com menor invalidação de caches. Atualmente as primeiras implementações em sistemas reais estão aparecendo e se mostrando viáveis. Uma contribuição futura interessante é

adaptar os algoritmos apresentados aqui para uma destas estruturas de dados.

Portanto, estabelecemos a indicação de que a construção de índices baseada nas características da carga de trabalho ainda é uma direção de pesquisa frutífera, levando-se em conta a mudança de paradigma proposta. Esta mudança consiste em modelar continuamente o padrão dinâmico dos acessos a fim de manter o sistema apto a responder às solicitações que chegam a ele. Conclui-se também que a estratégia proposta tem impacto positivo no desempenho do método de acesso, e que as ideias desenvolvidas podem ser estendidas para outros componentes do sistema.

REFERÊNCIAS

ATHANASSOULIS, M.; IDREOS, S. Design tradeoffs of data access methods. In: **Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016**. [S.l.: s.n.], 2016. p. 2195–2200.

ATHANASSOULIS, M.; KESTER, M. S.; MAAS, L. M.; STOICA, R.; IDREOS, S.; AILAMAKI, A.; CALLAGHAN, M. Designing access methods: The RUM conjecture. In: **Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016.** [S.l.: s.n.], 2016. p. 461–466.

BENDER, M. A.; FARACH-COLTON, M.; JOHNSON, R.; MAURAS, S.; MAYER, T.; PHILLIPS, C. A.; XU, H. Write-optimized skip lists. In: **Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017**. [S.l.: s.n.], 2017. p. 69–78.

BRUNO, N.; CHAUDHURI, S. An online approach to physical design tuning. In: **Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007**. [s.n.], 2007. p. 826–835. Disponível em: <https://doi.org/10.1109/ICDE.2007.367928>.

CHAUDHURI, S.; NARASAYYA, V. R. An efficient cost-driven index selection tool for microsoft SQL server. In: **VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece**. [s.n.], 1997. p. 146–155. Disponível em: <http://www.vldb.org/conf/1997/P146.PDF>.

CHIN, Y. H. Analysis of vsam's free-space behavior. In: **Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA.** [s.n.], 1975. p. 514–515. Disponível em: <http://doi.acm.org/10.1145/1282480.1282529>.

FAERBER, F.; KEMPER, A.; LARSON, P.; LEVANDOSKI, J. J.; NEUMANN, T.; PAVLO, A. Main memory database systems. **Foundations and Trends in Databases**, v. 8, n. 1-2, p. 1–130, 2017.

GILBERT, S.; LYNCH, N. A. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. **SIGACT News**, v. 33, n. 2, p. 51–59, 2002. Disponível em: <http://doi.acm.org/10.1145/564585.564601>.

GRAEFE, G. Sorting and indexing with partitioned b-trees. In: **CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings**. [s.n.], 2003. Disponível em: <http://www-db.cs.wisc.edu/cidr/cidr2003/program/p1.pdf>.

GRAEFE, G. Sorting in a memory hierarchy with flash memory. **Datenbank-Spektrum**, v. 11, n. 2, p. 83–90, 2011. Disponível em: <https://doi.org/10.1007/s13222-011-0062-6>.

GRAEFE, G.; HALIM, F.; IDREOS, S.; KUNO, H. A.; MANEGOLD, S. Concurrency control for adaptive indexing. **PVLDB**, v. 5, n. 7, p. 656–667, 2012. Disponível em: <http://vldb.org/pvldb/vol5/p656_goetzgraefe_vldb2012.pdf>.

GRAEFE, G.; KUNO, H. A. Self-selecting, self-tuning, incrementally optimized indexes. In: **EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings**. [s.n.], 2010. p. 371–381. Disponível em: <http://doi.acm.org/10.1145/1739041.1739087>.

GUPTA, H.; HARINARAYAN, V.; RAJARAMAN, A.; ULLMAN, J. D. Index selection for OLAP. In: **Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.** [s.n.], 1997. p. 208–219. Disponível em: <https://doi.org/10.1109/ICDE.1997.581755>.

HALIM, F.; IDREOS, S.; KARRAS, P.; YAP, R. H. C. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. **PVLDB**, v. 5, n. 6, p. 502–513, 2012. Disponível em: <http://vldb.org/pvldb/vol5/p502_felixhalim_vldb2012.pdf>.

HUANG, G.-B.; ZHOU, H.; DING, X.; ZHANG, R. Extreme learning machine for regression and multiclass classification. **IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)**, IEEE, v. 42, n. 2, p. 513–529, 2012.

HUANG, G.-B.; ZHU, Q.-Y.; SIEW, C.-K. Extreme learning machine: a new learning scheme of feedforward neural networks. In: IEEE. **Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on**. [S.l.], 2004. v. 2, p. 985–990.

IDREOS, S. Big data exploration. In: TAYLOR AND FRANCIS. **Big Data Computing**. [S.l.]: Taylor and Francis, 2013.

IDREOS, S.; GROFFEN, F.; NES, N.; MANEGOLD, S.; MULLENDER, K. S.; KERSTEN, M. L. Monetdb: Two decades of research in column-oriented database architectures. **IEEE Data Eng. Bull.**, v. 35, n. 1, p. 40–45, 2012. Disponível em: <http://sites.computer.org/debull/A12mar/monetdb.pdf>.

IDREOS, S.; KERSTEN, M. L.; MANEGOLD, S. Database cracking. In: **CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings**. [s.n.], 2007. p. 68–78. Disponível em: <http://www.cidrdb.org/cidr2007/papers/cidr07p07.pdf>.

IDREOS, S.; PAPAEMMANOUIL, O.; CHAUDHURI, S. Overview of data exploration techniques. In: **Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial**. [S.l.: s.n.], 2015.

KASTRATI, F.; MOERKOTTE, G. Optimization of disjunctive predicates for main memory column stores. In: **Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017**. [s.n.], 2017. p. 731–744. Disponível em: <http://doi.acm.org/10.1145/3035918.3064022>.

KEIM, D. A. Exploring big data using visual analytics. In: **Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014.** [s.n.], 2014. p. 160. Disponível em: <http://ceur-ws.org/Vol-1133/paper-26.pdf>.

KERSTEN, M. L.; SIDIROURGOS, L. A database system with amnesia. In: **CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings**. [s.n.], 2017. Disponível em: <http://cidrdb.org/cidr2017/papers/p58-kersten-cidr17.pdf>.

LARSON, P. Analysis of index-sequential files with overflow chaining. **ACM Trans. Database Syst.**, v. 6, n. 4, p. 671–680, 1981. Disponível em: <http://doi.acm.org/10.1145/319628.319665>.

LEHMAN, P. L.; YAO, S. B. Efficient locking for concurrent operations on b-trees. **ACM Trans. Database Syst.**, v. 6, n. 4, p. 650–670, 1981.

LEVANDOSKI, J. J.; SENGUPTA, S. The bw-tree: A latch-free b-tree for log-structured flash storage. **IEEE Data Eng. Bull.**, v. 36, n. 2, p. 56–62, 2013. Disponível em: <http://sites.computer.org/debull/A13june/bwtree1.pdf>.

LIANG, N.; HUANG, G.; SARATCHANDRAN, P.; SUNDARARAJAN, N. A fast and accurate online sequential learning algorithm for feedforward networks. **IEEE Trans. Neural Networks**, v. 17, n. 6, p. 1411–1423, 2006. Disponível em: <https://doi.org/10.1109/TNN.2006.880583>.

LIAROU, E.; IDREOS, S. dbtouch in action database kernels for touch-based data exploration. In: **IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014**. [s.n.], 2014. p. 1262–1265. Disponível em: <https://doi.org/10.1109/ICDE.2014.6816756>.

NANDI, A. Querying without keyboards. In: **CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings**. [s.n.], 2013. Disponível em: <http://cidrdb.org/cidr2013/Papers/CIDR13_Paper37.pdf>.

PARK, Y.; TAJIK, A. S.; CAFARELLA, M. J.; MOZAFARI, B. Database learning: Toward a database that becomes smarter every time. **CoRR**, abs/1703.05468, 2017. Disponível em: <http://arxiv.org/abs/1703.05468>.

PETRAKI, E.; IDREOS, S.; MANEGOLD, S. Holistic indexing in main-memory column-stores. In: **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015**. [s.n.], 2015. p. 1153–1166. Disponível em: <http://doi.acm.org/10.1145/2723372.2723719>.

PIRK, H.; PETRAKI, E.; IDREOS, S.; MANEGOLD, S.; KERSTEN, M. L. Database cracking: fancy scan, not poor man's sort! In: **Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014**. [s.n.], 2014. p. 4:1–4:8. Disponível em: <http://doi.acm.org/10.1145/2619228.2619232>.

PSAROUDAKIS, I.; SCHEUER, T.; MAY, N.; SELLAMI, A.; AILAMAKI, A. Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. **PVLDB**, v. 10, n. 2, p. 37–48, 2016. Disponível em: <http://www.vldb.org/pvldb/vol10/p37-psaroudakis.pdf>.

SCHNAITTER, K.; POLYZOTIS, N. A benchmark for online index selection. In: **Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China**. [s.n.], 2009. p. 1701–1708. Disponível em: <https://doi.org/10.1109/ICDE.2009.166>.

SCHUHKNECHT, F. M.; JINDAL, A.; DITTRICH, J. The uncracked pieces in database cracking. **PVLDB**, v. 7, n. 2, p. 97–108, 2013. Disponível em: <http://www.vldb.org/pvldb/vol7/p97-schuhknecht.pdf>.

SESHADRI, P.; SWAMI, A. N. Generalized partial indexes. In: **Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan**. [S.l.: s.n.], 1995. p. 420–427.

STONEBRAKER, M.; ABADI, D. J.; BATKIN, A.; CHEN, X.; CHERNIACK, M.; FERREIRA, M.; LAU, E.; LIN, A.; MADDEN, S.; O'NEIL, E. J.; O'NEIL, P. E.; RASIN, A.; TRAN, N.; ZDONIK, S. B. C-store: A column-oriented DBMS. In: **Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005**. [s.n.], 2005. p. 553–564. Disponível em: <http://www.vldb2005.org/program/paper/thu/p553-stonebraker.pdf>.

VARTAK, M.; RAHMAN, S.; MADDEN, S.; PARAMESWARAN, A. G.; POLYZOTIS, N. SEEDB: efficient data-driven visualization recommendations to support visual analytics. **PVLDB**, v. 8, n. 13, p. 2182–2193, 2015. Disponível em: <http://www.vldb.org/pvldb/vol8/p2182-vartak.pdf>.

XIE, Z.; CAI, Q.; JAGADISH, H. V.; OOI, B. C.; WONG, W. Parallelizing skip lists for in-memory multi-core database systems. In: **33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017**. [S.l.: s.n.], 2017. p. 119–122.

YU, X.; BEZERRA, G.; PAVLO, A.; DEVADAS, S.; STONEBRAKER, M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. **PVLDB**, v. 8, n. 3, p. 209–220, 2014. Disponível em: <http://www.vldb.org/pvldb/vol8/p209-yu.pdf>.