UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

NOVA ORGANIZAÇÃO PARA ESTRUTURA DE DADOS EM BANCOS RELACIONAIS: ESTUDO DE CASO

JOÃO PAULO POFFO

BLUMENAU
2009

2009/1-10

JOÃO PAULO POFFO

NOVA ORGANIZAÇÃO PARA ESTRUTURA DE DADOS EM BANCOS RELACIONAIS: ESTUDO DE CASO

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Marcel Hugo, Orientador

BLUMENAU
2009

2009/1-10

NOVA ORGANIZAÇÃO PARA ESTRUTURA DE DADOS EM BANCOS RELACIONAIS: ESTUDO DE CASO

Por

JOÃO PAULO POFFO

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Marcel Hugo, M.Eng., Orientador - FURB
Membro:	Prof. Alexander Roberto Valdameri, M.Eng. – FURB
Membro:	Prof. Adilson Vahldick, M.Sc. – FURB

Blumenau, 9 de julho de 2009

À Vera Lúcia que me ensinou sobre vida ao abrir meus olhos para a morte.

AGRADECIMENTOS

À minha noiva que soube identificar em seu companheiro os momentos em que ele era seu querido noivo daqueles em que ele era o escritor egoísta, chato e ranzinza desta monografia.

Aos meus amigos, por sempre me lembrarem que falta tão pouco.

Ao meu orientador, Marcel Hugo, por me dar forças me assustando com prazos que pareciam inalcançáveis.

Conhecer não é demonstrar nem explicar, é aceder à visão.

Antoine de Saint-Exupéry

RESUMO

Este trabalho é a implementação e realização de prova de conceito de um tipo de tabela para o Sistema Gerenciador de Banco de Dados (SGDB) MySQL. Um *storage engine* do MySQL é a camada que faz a persistência dos dados e se comunica com o SGDB através de uma *Application Programing Interface* (API). Cada tabela do MySQL pode ser de um *storage engine* diferente e este é identificado através do tipo de tabela. No tipo de tabela desenvolvido é possível criar e remover tabelas, executar seleção, inserção, exclusão e atualização de registros. A arquitetura do tipo de tabela define que todas as colunas de todos os registros são organizadas visando a diminuição de redundância e aumento de desempenho, sem necessidade de criação de índices. A prova de conceito é realizada através da comparação de informações obtidas através de testes com o tipo de tabela desenvolvido e com outros tipos do próprio MySQL, destacando as principais diferenças, limitações e capacidades. O tipo de tabela foi desenvolvido em C++ a partir do exemplo disponibilizado pela MySQL para construção de storage engines.

Palavras-chave: MySQL. Tipo de tabela. *Storage engine*.

ABSTRACT

This work is the implementation and benchmark of a table type to MySQL Data Base Management System (DBMS). A storage engine is the layer of MySQL that implements the persistence of the data and it communicates with the DBMS through an Application Programming Interface (API). Each table on MySQL can be owned by a different storage engine identified by the table type. The developed table type allows table creating and dropping, record selecting, inserting, deleting and updating. The table type architecture defines that each column of all records are sorted to reduce redundancy and increase performance, without creating any secondary indexes. The benchmark is implemented comparing test results of the new table type with other MySQL table types, highlighting the main differences, limitations and capabilities. The development of the table type was in C++ based on the example storage engine available in MySQL sources.

Key-words: MySQL. Table type. Storage engine.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

ACID - *Atomic, Consistency, Isolation, Durability*

API - *Application Programming Interface*

DDD - *Data Display Debugger*

DDL - *Data Definition Language*

DML - *Data Manipulation Language*

DVD - *Digital Video Disc*

EA - *Enterprise Architect*

GCC - GNU *Compiler Collection*

GDB - GNU *DeBugger*

ISAM - *Indexed Sequential Access Method*

LOB - *Large Object*

OSDB - *Open Source Database Benchmark*

RF - Requisito Funcional

RID - *Record IDentification*

RN - Regra de Negócio

RNF - Requisito Não Funcional

SGBD - Sistema Gerenciador de Banco de Dados

SQL - *Structured Query Language*

UML - *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Para um usuário é irrelevante como um programa de computador trata suas informações internamente, o mais importante é o seu resultado visível. Pois, caso o consumo de memória, processador ou espaço em disco forem altos, existem a cada dia novas tecnologias, mais rápidas, poderosas, e com preços mais baixos. Entretanto, há também a necessidade de manipular mais informação em menos tempo. Neste contexto, a evolução do hardware, apesar de contundente, sozinha, é insuficiente para preencher completamente esta lacuna. De tempos em tempos têm-se a necessidade de rever antigos conceitos e analisar se são realmente as melhores alternativas para resolver os problemas de hoje.

Existem várias formas de organizar as estruturas de bancos de dados relacionais. Desde as mais primitivas como arquivos binários, até as mais complexas baseadas em blocos de dados. Dentre todas as estruturas existentes identifica-se um padrão: os dados de um registro estão sempre juntos. Diante do exposto, propõe-se a quebra deste paradigma identificando alguns focos para possíveis ganhos em utilização de espaço de armazenamento e velocidade na obtenção da informação. Para atingir tal meta foi desenvolvida uma arquitetura onde cada coluna de cada tabela estará sempre ordenada. Esta organização visa inibir a criação de índices, evitando redundância de dados e tornando toda informação passível de ser localizada com eficiência. Afinal, não se sabe que informação é realmente importante até o momento em que ela se faz necessária.

É necessário testar se a arquitetura proposta pode realmente trazer tais benefícios. Este teste pode ser efetuado através da realização de uma prova de conceito onde são estabelecidas métricas para comparação com outras arquiteturas existentes. Porém, para possibilitar esta comparação, é necessário que esta nova arquitetura esteja implementada. O desenvolvimento de um Sistema Gerenciador de Banco de Dados (SGBD) compreende muitos problemas como controle de acesso, interfaces de comunicação com o usuário, integridade referencial, entre outros. O SGBD MySQL[1] ajuda a resolver este problema, pois disponibiliza um mecanismo que permite restringir o desenvolvimento somente à nova organização proposta. Este mecanismo é chamado de tipos de tabela. Os tipos de tabela permitem definir estruturas de dados diferentes para cada tabela criada na base de dados mantendo a possibilidade de relacionamento. Cada tipo de tabela é mantida por um *storage engine* o qual funciona como

[1] MySQL é um SGBD de código aberto e portável (DUBOIS, 2008).

uma camada de persistência que é ligada às camadas de controle do banco de dados. A partir deste ponto a implementação do trabalho será referenciada como Tipo de Tabela.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma arquitetura para armazenamento e obtenção de dados em um banco de dados relacional.

Os objetivos específicos do trabalho são:

a) construir um tipo de tabela e seu *storage engine* para o MySQL;
b) obter estatísticas de desempenho e volume a respeito do tipo de tabela proposto e os tipos InnoDB[2] e MyISAM[3] do MySQL;
c) analisar resultados obtidos e traçar comparativo.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho é organizado em cinco capítulos. No primeiro capítulo é apresentada a introdução, os objetivos e estrutura do trabalho. O segundo capítulo fundamenta teoricamente estruturas de dados, sistemas gerenciadores de bancos de dados, tipos de tabela do MySQL, *benchmarks*, a ferramenta *Open Source Database Benchmark* (OSDB) e finaliza com outros trabalhos com temática semelhante. O desenvolvimento do tipo de tabela com seus requisitos, especificação, implementação e resultados são descritos no terceiro capítulo. No quarto, ocorre o desenvolvimento da prova de conceito e é destacada sua especificação, como foi implementada e os resultados obtidos. Finalmente no quinto capítulo estão as conclusões e extensões sugeridas.

[2] InnoDB é um tipo de tabela do MySQL que implementa controle transacional, integridade referencial e bloqueio de alterações no nível de registro (DUBOIS, 2008).

[3] MyISAM é um tipo de tabela do MySQL baseada no formato ISAM (Indexed Sequential Access Method) da IBM© e otimizada para ser especialmente veloz no processo de consulta (DUBOIS, 2008).

2 FUNDAMENTAÇÃO TEÓRICA

São contextualizados nas seções seguintes os principais assuntos com o fim de introduzir conceitos utilizados no decorrer do trabalho. Inicialmente são elucidadas algumas prerrogativas em estrutura de dados, SGBDs e suas estruturas especiais. Adiante é explicado o funcionamento dos tipos de tabelas no MySQL, como é utilizado o *benchmarking* como prova de conceito e a ferramenta para sua melhor execução. Por fim, são relacionados trabalhos correlatos ao proposto.

2.1 ESTRUTURAS DE DADOS

> Em toda e qualquer aplicação computacional, a relação entre as estruturas de dados empregadas e os algoritmos que as manipulam é sempre muito íntima. Portanto, ao se considerar uma opção para a solução de um problema, estes dois aspectos precisam ser igualmente avaliados, pois os reflexos são mútuos e diretos. (HEINZLE, 2006, p. 3).

Uma estrutura de dados não existe sem um fim específico. Elas são criadas com a intenção de facilitar o entendimento, organizar, economizar recursos ou para melhorar o desempenho de algum algoritmo, ou seja, a existência dessas estruturas de dados está tão assegurada quanto a problemas a serem resolvidos. Alguns problemas nem podem ser resolvidos sem uma estrutura de dados adequada. Heinzle (2006, p. 2) afirma que as estruturas de dados precisam retratar as relações existentes entre os dados de forma a preservar as propriedades do mundo real correspondente, e, ainda, considerar e facilitar as operações ou manipulações que serão posteriormente realizadas sobre eles.

Tipos primitivos, estruturas e bancos de dados se confundem por serem simplesmente uma especialização um do outro. Fisicamente existe apenas um tipo primitivo: o *bit*. Porém, com a evolução das diversas linguagens de programação, cada vez mais estruturas foram sendo abstraídas com a finalidade de auxiliar o programador, surgindo assim tipos primitivos mais complexos como: inteiro, ponto flutuante ou data. A partir desses tipos mais evoluídos são criadas estruturas especializadas que quando agrupadas têm a denominação de banco de dados.

Existem vários tipos de estruturas de dados, entre elas: listas, pilhas, filas, árvores,

incluindo algoritmos de pesquisa e ordenação. O registro (*struct* em C) é formado pelo agrupamento de vários tipos primitivos armazenados juntos e linearmente em algum lugar da memória ou do disco.

2.2 SISTEMAS GERENCIADORES DE BANCO DE DADOS

Segundo Elmasri e Navathe (2005, p. 4, grifo do autor), "Um **banco de dados** é uma coleção de dados relacionados". Um SGBD define-se no próprio nome. São programas que em conjunto formam um sistema que controla e mantém uma coleção de dados, seja esta grande ou pequena, garantindo sua consistência, confiabilidade e segurança.

O modelo relacional tem como principal característica a definição de tabelas e o relacionamento entre elas. Por exemplo, a tabela `vendedor` tem relacionamento com a tabela `vendas`, pois, por experiência de negócio do modelador da base de dados, toda venda é feita por um vendedor. Isso denota uma possibilidade de aproximação muito clara entre o modelo e a realidade. Este modelo é o mais utilizado atualmente, tendo como exemplos: PostGreSQL, Firebird, Microsoft SQL Server, entre outros.

2.2.1 Armazenamento

Ramakrishnan (1998, p. 39) divide a memória de computador hierarquicamente em três níveis: a memória (virtual, principal e *cache* - para acesso rápido a dados); dispositivos mais lentos como discos rígidos; e toda a classe de dispositivos ainda mais lentos como o *Digital Video Disc* (DVD).

Armazenar todos os dados de uma grande empresa no armazenamento primário (memória principal) ainda é muito caro. Porém, existem iniciativas que prevêem que em um futuro próximo será relativamente mais barato manter todos os dados da empresa em memória, pois, além do preço da memória volátil estar diminuindo, o desempenho obtido com esta transição é tão grande que justifica, em alguns casos, o aumento destes custos. Um exemplo dessa transição é o Prevayler (WUESTEFELD, 2009), que é um sistema de persistência de dados de código aberto para Java que mantém todos os objetos do persistidos em memória.

A realidade atualmente continua tendo como fardo a utilização em grande escala do armazenamento secundário, deixando ao critério do gerenciador de memória do SGBD descobrir qual a informação mais importante a ser deixada em memória. Portanto, este deve gerenciar adequadamente tanto o armazenamento primário quanto o secundário.

2.2.2 Arquitetura de armazenamento

Com o fim de otimizar o gerenciamento de armazenamento, o SGBD divide-a em páginas de tamanho fixo, normalmente configurável, de forma semelhante ao que o próprio sistema operacional faz com seu sistema de arquivos. Um dos vários problemas desta arquitetura é identificar de forma rápida quais são as páginas livres e evitar a subutilização do espaço disponibilizado em cada uma destas páginas.

Estas páginas armazenam um ou mais registros. Estes registros podem ter tamanho fixo ou variável. Em um registro de tamanho fixo, todos seus campos (ou atributos), também são de tamanho fixo, ou seja, têm um valor máximo e um mínimo fisicamente delimitado. Já os registros de tamanho variável têm um ou mais campos de tamanho variável, normalmente delimitados por um *byte* terminador (RAMAKRISHNAN, 1998, p. 50). Registros de tamanho variável são otimizados para economizar espaço, porém têm uma perda considerável de desempenho em relação aos de tamanho fixo por ser necessário verificar limites.

O SGBD Oracle e o tipo de tabela InnoDB do MySQL têm estrutura de dados baseada em registros de tamanho variável e cada registro possui um identificador único (*Record IDentification* – RID). Porém, o armazenamento reserva diversos outros problemas como otimização do processo de gravação, reaproveitamento de espaço de registros excluídos, exclusão mútua, segurança e confiabilidade dos dados.

2.2.3 Índices

A estrutura básica de coleções de dados considerada até este ponto armazena registros de forma aleatória e suporta obtenção de todos os registros ou de um registro específico através de seu RID. Algumas vezes é necessário obter registros informando alguma condição nos campos do registro desejado, por exemplo, achar todos os vendedores com comissão maior que R$ 15,00. Para acelerar a busca desta informação, é possível construir estruturas de

dados auxiliares para achar rapidamente os RIDs que satisfaçam a condição informada. Estas estruturas auxiliares são chamadas de índices (RAMAKRISHNAN, 1998, p. 55).

Uma possibilidade de organização dos registros na base de dados seria gravá-los fisicamente de forma ordenada (*clustering*). Se por algum motivo for gravado um registro que se posicione no início de uma tabela, todos os registros subsequentes serão movidos um passo adiante para disponibilizar espaço para este novo registro. Esta estrutura se torna inviável devido ao grande custo de reorganizar repetidamente estes registros fisicamente na base de dados.

Existem várias estruturas de indexação e elas são de responsabilidade de cada SGBD. As mais comuns são a baseada em árvore (Árvore B) e em tabelas de espalhamento (*Hash*). Índices possuem sempre no mínimo dois campos, o campo a ser ordenado e o RID do registro ao qual ele pertence (ELMASRI; NAVATHE, 2005, p. 326).

As tabelas de espalhamento são um tipo de divisão dos registros por categorias. Estas categorias são geradas através do resultado de funções *hash* aplicadas sobre os campos da chave, ou seja, cada busca é efetuada apenas no bloco de dados de uma tabela para onde a categorização apontar. A qualidade desta função *hash* implica diretamente no desempenho da busca. Diferentemente de um índice árvore B, este tipo de índice não é eficiente em consultas por faixa, pois o resultado da função não é necessariamente sequencial.

2.2.4 Índices Árvore B

Os índices estruturados em árvore possuem um nó principal chamado raiz, e cada nó filho da raiz pode ter um ou mais nós filhos, e os nós que não possuem filhos, são denominados nós folha. A cada busca, os nós são percorridos e é escolhido entre os nós filhos qual o que mais se assemelha à busca. Esta operação ocorre recursivamente até extinguir as possibilidades de localização ou até o registro ser encontrado.

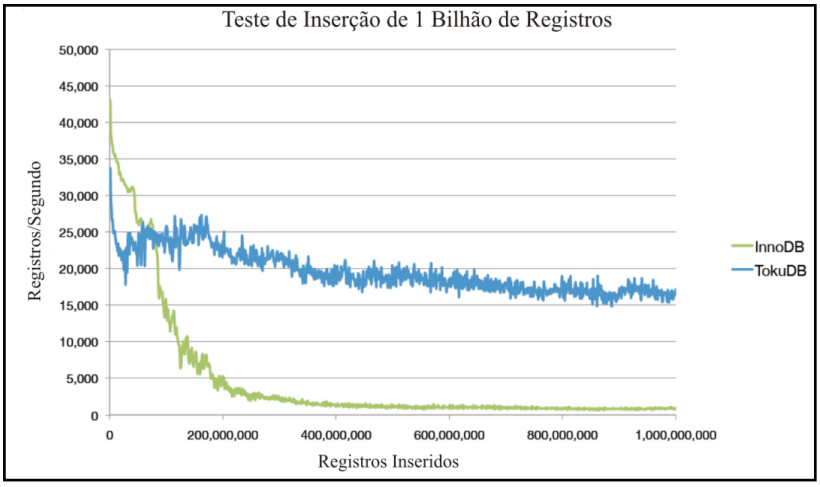
Garcia-Molina, Ullman e Widom (2001, grifo do autor, p. 166) afirmam que, como o nome implica, uma árvore B organiza seus blocos em uma árvore. A árvore é balanceada, o que significa que todos os caminhos desde a raiz até uma folha têm o mesmo comprimento. Em geral, existem três camadas em uma árvore B: a raiz, uma camada intermediária e as folhas, mas é possível haver qualquer número de camadas.

Uma variante da árvore B, denominada árvore B+ (*B+Tree*), mais comumente utilizada em sistemas comerciais, difere essencialmente na localização dos dados. Nesta

última os dados são somente armazenados nos nós folha e nos nós intermediários são apenas chaves que orientam a busca na árvore. Segundo Elmasri e Navathe (2005, p. 336), os nós árvore B são preenchidos entre 50% e 100%, e ponteiros para os blocos de dados são armazenados em ambos os nós internos e nós folhas da estrutura da árvore. Nas árvores B+ os ponteiros para os blocos de dados são armazenados apenas nos nós folhas, implicando em menos níveis e a índices mais densos.

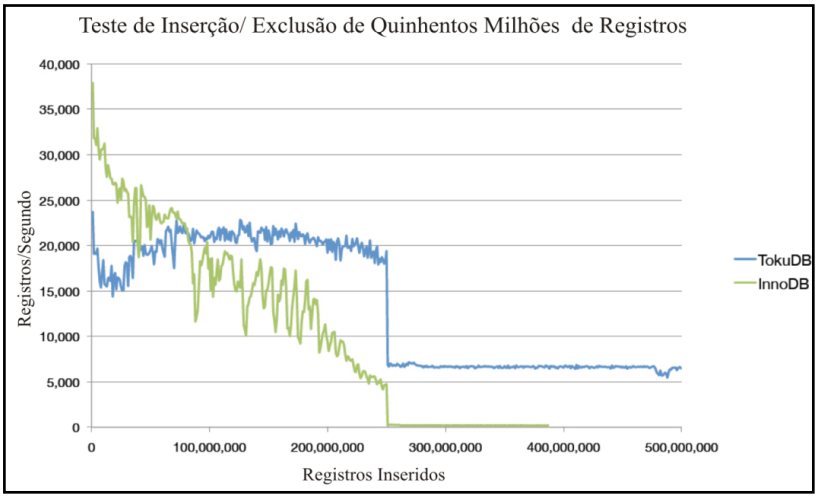
A construção de um índice árvore B leva em consideração o tamanho de cada bloco de dados do banco, sendo este dividido de forma a contabilizar, em relação ao dado a ser organizado, a quantidade de nós filhos que cada nó pode possuir, ou seja, têm uma ramificação com quantidade previsível e fixa de nós filhos. Neste contexto, o espaço das chaves do índice deve sempre prever o maior valor possível de cada campo. Assim, a indexação de um campo texto de tamanho variável sempre ocupará seu tamanho máximo nos nós do índice. Isto pode ser especialmente impactante na indexação de campos de texto de tamanho variável com limites muito grandes.

A empresa Tokutek desenvolveu um tipo de tabela baseado em uma nova tecnologia de indexação em árvores fractais (TOKUTEK, 2009). Este tipo de indexação tem resultados promissores conforme exemplificados: na Figura 1, onde é inserido um bilhão de registros em uma tabela com três índices compostos (múltiplas colunas); na Figura 2, onde ocorre a inclusão de duzentos e cinquenta mil registros e a partir disso, a cada mil registros incluídos, são excluídos outros mil; e na Figura 3, onde é feito um teste de consultas com o crescimento na quantidade de registros, e os tempos de consulta são mais constantes. Porém, por mais que seja um tipo de tabela de um banco de dados de código aberto, o TokuDB (tipo de tabela da Tokutek para o MySQL) não tem seu código aberto.



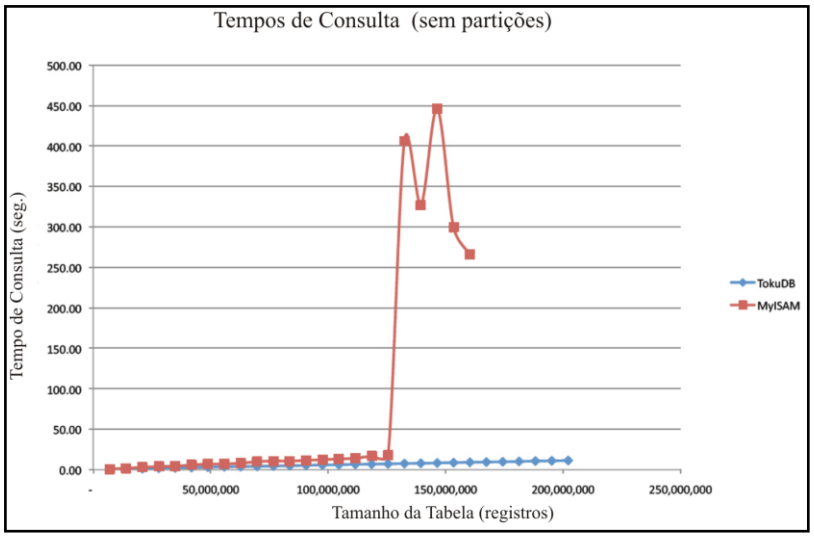
Fonte: adaptado de Tokutek (2009).

Figura 1 - Teste Tokutek: inserção de um bilhão de registros



Fonte: adaptado de Tokutek (2009).

Figura 2 - Teste Tokutek: inserção/exclusão de quinhentos milhões registros



Fonte: adaptado de Tokutek (2009).

Figura 3 - Teste Tokutek: consulta em tabela com índices de árvores fractais

2.3 TIPOS DE TABELA DO MYSQL

MySQL, o mais popular SGBD com linguagem de consulta estruturada (SQL[4]), é desenvolvido, distribuído e mantido por MySQL AB. MySQL AB[5] é uma empresa comercial, fundada pelos desenvolvedores do MySQL. É da segunda geração de companias *Open Source*, pois une os valores da metodologia de código aberto com um modelo de negócio bem

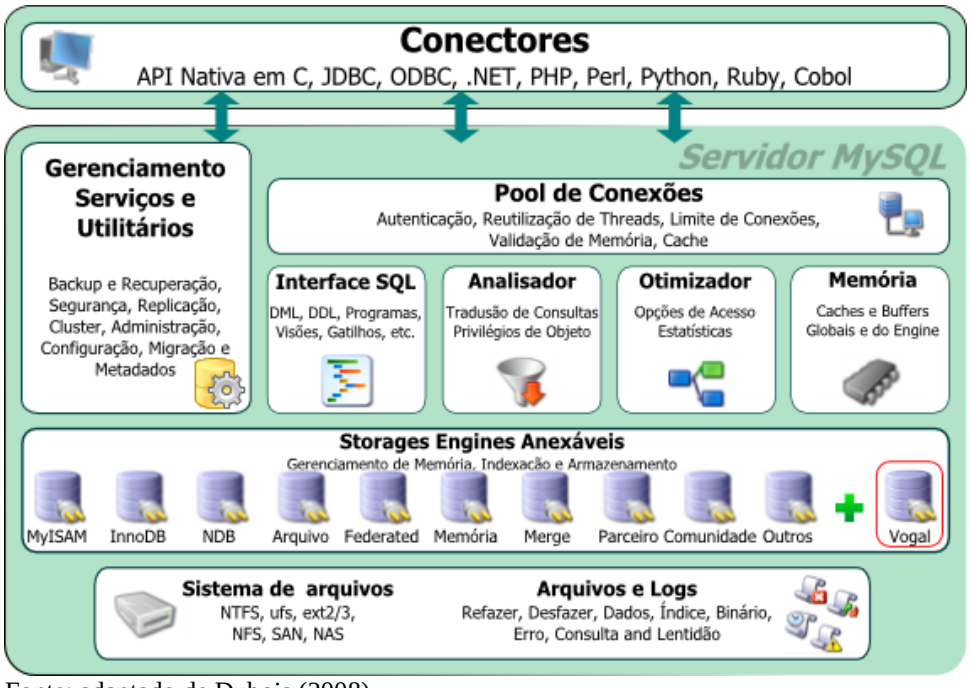
[4] *Structured Query Language* (SQL) é uma linguagem formal baseada na álgebra relacional que se aproxima da linguagem humana com o intuito de facilitar a interação homem-máquina. É amplamente utilizada por SGBDs como meio de comunicação do usuário.

[5] MySQL AB foi comprada pela Sun Microsystems em fevereiro de 2008 e a Sun, em abril de 2009, foi comprada pela Oracle.

sucedido (DUBOIS, 2008). Empresas como Yahoo!, Google, Nokia e YouTube, utilizam o SGBD MySQL em seus sistemas. É altamente portável, possui alta performance, é confiável e fácil de utilizar (MYSQL AB, 2008).

Os *storage engines* de cada tipo de tabela gerenciam o armazenamento e indexação de dados do MySQL. O servidor comunica-se com o *storage engine* através de uma *Application Programming Interface* (API). Esses podem ser construídos de maneira progressiva começando como somente leitura, e então implementando rotinas de gravação e exclusão, e posteriormente indexação e controle transacional (MYSQL COMMUNITY, 2008).

A partir da versão 5.1, foi introduzida no MySQL uma arquitetura que permite utilizar diferentes *storage engines* adicionando-os em um servidor com o serviço de banco de dados em execução sem precisar recompilar[6] os códigos-fonte do servidor MySQL em si, o que facilita o acoplamento de novos tipos de tabela e evita que o desenvolvedor seja obrigado a possuir o código-fonte completo do MySQL para que seja possível implementar um novo tipo de tabela. É possível visualizar a arquitetura global do MySQL na Figura 4.



Fonte: adaptado de Dubois (2008).

Figura 4 - Arquitetura global do MySQL com *storage engines* plugáveis

[6] Recompilar é o ato de compilar mais que uma vez. Compilar é todo o processo desde a validação dos códigos da linguagem de programação até sua tradução para a linguagem de máquina.

2.3.1 Arquitetura de *storage engines* do MySQL

Alguns dos recursos que podem estar presentes em um *storage engine* são: concorrência, controle transacional, integridade referencial, armazenamento físico, indexação, gerenciamento de memória, dentre outros (DUBOIS, 2008). Estes recursos e a qualidade apresentada por cada um são os diferenciais procurados pelo administrador de banco de dados para escolher um tipo de tabela ou outro.

Além dos códigos-fonte dos *storage engines* estarem disponíveis por serem de código aberto, existe uma versão exemplo do MySQL disponibilizada pela MySQL AB na linguagem C++ e um manual de passos a serem seguidos para a sua implementação.

O *storage engine* exemplo não faz nada. Seu propósito é ilustrar ao desenvolvedor como começar a construir um tipo de tabela. O tipo de tabela se chama *EXAMPLE*, disponível junto aos códigos fonte do MySQL, possui a especificação de uma classe com as informações básicas que um *storage engine* deve possuir.

2.4 *BENCHMARKING* COMO PROVA DE CONCEITO

Um dos métodos de prova de conceito é o *benchmarking*. Este método tem como prerrogativa a execução de testes a partir de sistemas automatizados em dois ou mais programas, fornecendo como saída algumas métricas para comparação entre eles. Segundo Ramakrishnan (1998, p. 465), *benchmarks* deveriam ser portáveis, de fácil compreensão, naturalmente escaláveis para maiores faixas de problemas e deveriam medir performance de pico (por exemplo, transações/segundo) para volumes típicos de trabalho no domínio do sistema.

Dificilmente será criado um comparativo que obtenha resultados simples e diretos. De acordo com as estatísticas e resultados das ferramentas utilizadas para auxiliar o processo de análise, o ser humano é quem chegará às devidas conclusões. São levados em consideração alguns pontos facilmente delineáveis como a capacidade da infra-estrutura, roteiro dos testes, entre outros. Mas também pontos mais subjetivos como otimizações e implementações distintas existentes nos artefatos testados.

Desta forma, o *benchmarking*, como prova de conceito, é utilizado por sua

característica sistemática e informacional, possibilitando ao analista projetar, acompanhar e traçar conclusões de menor risco de refutação.

2.5 OSDB

Conforme estudo realizado por Pires, Nascimento e Salgado (2006), o OSDB é uma ferramenta de *benchmark* de código aberto que cresceu a partir de um pequeno projeto de Andy Riebs na Compaq Computer Association com o objetivo inicial de avaliar a taxa de *In/Out* (I/O) e o poder de processamento da plataforma GNU Linux/Alpha[7]. O OSDB é compatível com vários SGBDs, entre eles o MySQL, e possibilita a comparação através de métricas como tamanho máximo suficiente do banco de dados para completar o teste em menos de 12 horas, tempo de resposta das consultas e número de linhas retornadas por segundo.

Benchmarks são raramente confiáveis e úteis. Estes são vistos com desconfiança, pois os indivíduos que apresentam os resultados podem ter motivações e competência suspeitas. Também são considerados inúteis, pois dificilmente o modelo é equiparável aos trabalhos que se deseja comparar (RIEBS, 2004). Portanto, para um *benchmark* estar acima de qualquer suspeita e sempre se adequar a necessidade do desenvolvedor, o ideal é ser transparente e acessível a qualquer interessado, como o OSDB que é de código aberto.

2.6 TRABALHOS CORRELATOS

Existem diversas iniciativas correlatas ao trabalho proposto, sendo: um novo tipo de tabela com o intuito de substituir o MyISAM, o Maria do MySQL (DUBOIS, 2008); o tipo de indexação *Bitmap*, que é um método aplicável apenas para algumas necessidades específicas (CYRAN et al., 2005); e um comparativo de desempenho entre SGBDs de código aberto (PIRES; NASCIMENTO; SALGADO, 2006).

[7] GNU Linux/Alpha é uma versão do GNU Linux que funciona sobre a arquitetura Alpha criada pela Compaq. Alpha é o nome dado para o projeto de processadores com a arquitetura *Reduced Instruction Set Computer* (RISC) de 64 bits criada pela empresa norte americana Digital Equipment (CROOK, 1997).

2.6.1 Maria: tipo de tabela

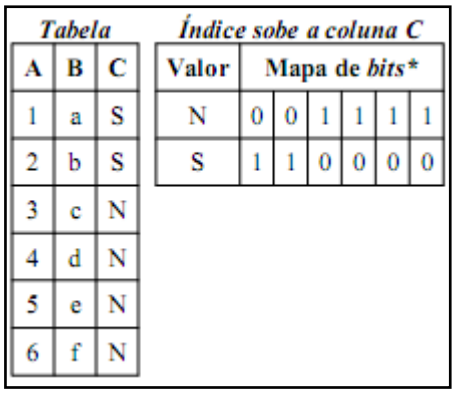
O *storage engine* do tipo de tabela Maria foi introduzido no MySQL a partir da versão 6.0.6 e é uma especialização do MyISAM com o recurso de operações atômicas (operações iniciam bloqueando a tabela para alteração e só liberam outras alterações com o término da operação corrente, com o desbloqueio da tabela). Este tipo de tabela suporta todas as principais funcionalidades do *storage engine* MyISAM, porém inclui facilidades de recuperação (em caso de um problema de sistema), gravação completa de históricos (incluindo criação e limpeza de tabelas), entre outros. O Maria mantém a velocidade e a flexibilidade de seu predecessor combinando estas características com suporte transacional (DUBOIS, 2008).

No MyISAM existem três formatos de linha: fixo, variável e comprimido. O primeiro e o segundo são selecionados automaticamente dependendo do tipo dos campos da tabela, ou seja, caso a tabela tenha algum campo de tamanho variável o formato da linha muda para atender essa necessidade. O formato de linha comprimida, que é uma configuração opcional que visa redução de utilização de espaço em disco, possui um processo intermediário de compressão e descompressão. Porém, estes três formatos de linha existem por compatibilidade e para serem usados em modelos não transacionais. Portanto, foi implementado um novo formato de linha no tipo de tabela Maria que é formado por páginas, exatamente para possibilitar um fim transacional otimizado.

2.6.2 Indexação *Bitmap*

O propósito de um índice é prover ponteiros para as linhas de uma tabela correspondentes a uma chave. Em um índice regular, isso é alcançado gravando uma lista de RIDs para as chaves de cada linha. O SGBD Oracle grava as chaves repetidamente para cada RID. Em um índice *Bitmap*, é utilizado um mapeamento de *bits*, ao invés de uma lista de RIDs (CYRAN et al., 2005). Cada *bit* deste mapeamento corresponde a um possível RID, ou seja, se o *bit* estiver ligado, a posição dele corresponde à posição do registro desejado na tabela (Figura 5). Isso é eficiente pelo fato de não precisar acessar os dados físicos da tabela para restringir as informações que serão acessadas. O sistema de indexação permite a realização de intersecção entre índices do tipo *Bitmap* sobre diferentes colunas para aumentar

ainda mais a performance e a pontualidade da obtenção da informação.



Tabela

A	B	C
1	a	S
2	b	S
3	c	N
4	d	N
5	e	N
6	f	N

Índice sobe a coluna C

Valor	Mapa de *bits**					
N	0	0	1	1	1	1
S	1	1	0	0	0	0

Figura 5 - Tabela com índice do tipo *Bitmap*

Porém, esta eficiência tem um alto custo de atualização. Para cada gravação efetuada na tabela, existe a necessidade de reconstrução total do índice. Isso se faz imprescindível pela necessidade de atualização do mapa de *bits*, tornando-se inviável para tabelas com grande volume de atualização. Portanto, este tipo de índice deve ser utilizado sobre colunas com muita repetição do conteúdo e sobre tabelas com baixo índice de atualização.

2.6.3 Comparativo de desempenho entre bancos de dados de código aberto

O objetivo da proposta de Pires, Nascimento e Salgado (2006, p. 1) é apresentar um estudo comparativo de desempenho entre os SGBD MySQL e PostGreSQL, em plataforma GNU/Linux, utilizando o *Open Source Database Benchmark* (OSDB). O estudo consiste em analisar as métricas geradas pelo OSDB e estimular melhorias nas funcionalidades dos SGBDs relacionadas com desempenho.

É inicialmente feita uma análise entre alguns *benchmarks*, chegando-se à conclusão de que o OSDB é o que melhor se adequa às suas necessidades de testes pela abrangência destes testes e possibilidades de variação.

As configurações de hardware, sistema operacional e banco de dados em que seriam realizados todos os testes são descritas de forma detalhada. Estes testes são resumidos visando auxiliar a análise e a inferência de um resultado. Em seguida são apresentados detalhadamente apenas os testes onde houve maior divergência. Alguns destes testes são:

a) carga e estrutura: criação de tabelas, carga das tabelas e criação de índices;
b) seleções: capacidade do SGBD de escolher a melhor forma de consulta à tabela;
c) junções: teste de junção sem utilizar índices;
d) projeções: seleção utilizando uma cláusula de unicidade de linhas (cláusula

`distinct` do SQL);

e) agregações: utilização de agrupamento (cláusula `group by`) e funções de agregação em colunas indexadas (por exemplo, a função `sum` do SQL).

Como conclusão é definida onde há possibilidade de melhoria em cada um dos SGBDs e qual que apresentou melhores resultados nos testes executados. São então descritos possíveis novos testes em outras configurações de hardware, sistema operacional e SGBD.

3 DESENVOLVIMENTO DO TIPO DE TABELA

O desenvolvimento do projeto divide-se em duas etapas: o desenvolvimento do Tipo de Tabela e o desenvolvimento da prova de conceito. Neste capítulo são apresentados os requisitos do sistema, especificação, implementação e os resultados obtidos em relação ao desenvolvimento do Tipo de Tabela.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Simplificando a estrutura de uma tabela a uma estrutura formada por linhas e colunas onde cada linha é um registro e cada coluna é um campo, é representada na Figura 6 esta estrutura com uma exemplificação de como ficariam os índices, caso estes existissem. Nesta estrutura o registro é atômico (linha em destaque na Figura 6), ou seja, está sempre junto fisicamente. Na Figura 7 é representado esquematicamente como será a nova estrutura de tabelas, tornando cada campo independente, ordenado e inter-relacionável.

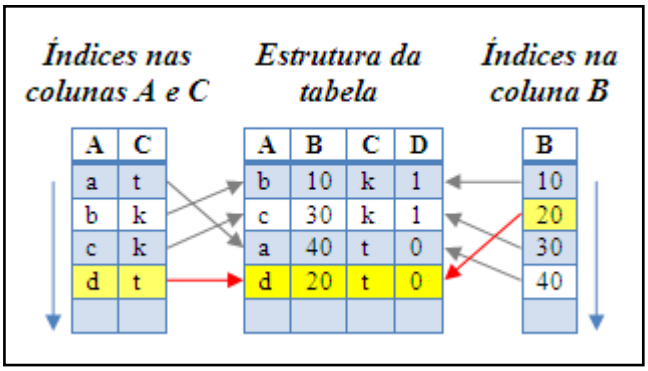


Figura 6 - Estrutura comum de tabela

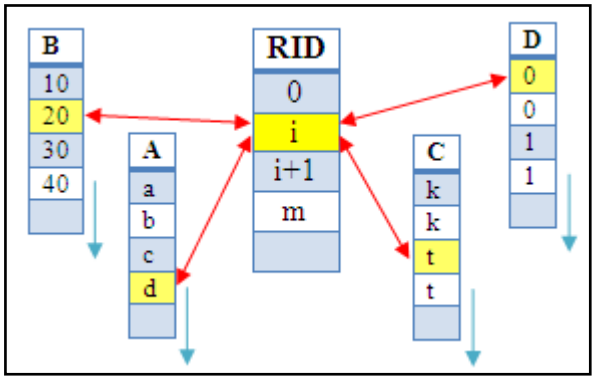


Figura 7 - Nova estrutura de tabela

A arquitetura do Tipo de Tabela proposto deverá:

a) permitir ao usuário[8] criar e remover tabelas (Requisito Funcional - RF);
b) permitir ao usuário inserir, atualizar e remover registros, das tabelas criadas (RF);
c) permitir ao usuário extrair registros, das tabelas criadas, utilizando a linguagem SQL, limitando-se a uma tabela por extração e sem recursos de agrupamento (RF);
d) permitir ao usuário projetar quaisquer campos da tabela desejada e criar condições para restringir a extração (RF);
e) ser implementado utilizando a linguagem de programação C++ com a biblioteca GNU *Compiler Collection* (GCC) em Linux com o pacote Ubuntu (Requisito Não Funcional – RNF);
f) ser implementado compreendendo o armazenamento dos dados e metadados[9] no dispositivo secundário e implementando um gerenciamento de memória primitivo, não implementando controle transacional nem de concorrência (RNF);
g) As tabelas devem ser definidas apenas com campos de tipo texto ou numérico inteiro (RNF);
h) As instruções SQL devem obedecer ao padrão definido no manual de referência do MySQL (DUBOIS, 2008) (Regra de Negócio - RN);
i) Os tipos texto serão limitados a 127 caracteres (RN);
j) Os tipos numéricos deverão estar entre -2147483648 e 2147483647 (inteiro de 32 *bits*) (RN);
k) As limitações de tamanho na definição do campo não serão validadas (RN);
l) As tabelas devem ser definidas apenas com os campos e seus tipos. Qualquer outra definição não será validada. Como por exemplo, chaves primárias, obrigatoriedade, auto-incremento, etc. (RN).

3.2 ESPECIFICAÇÃO

O projeto do Tipo Tabela é especificado empregando o processo de análise e projeto orientado a objetos. A ferramenta Enterprise Architect (EA) é utilizada para o

[8] Usuário, no contexto desta monografia, significa o ator que utilizará o SGBD. Na maioria dos casos este ator é outro sistema.

[9] Metadados é a definição básica de um esquema em um banco de dados, ou seja, é a estrutura primordial que identifica os objetos existentes e orienta o SGDB.

desenvolvimento de todos os diagramas da *Unified Modeling Language* (UML).

O Tipo de Tabela foi denominado Vogal (AEIOU - Armazenamento e Extração de Informação Otimizada por Unicidade) e seu código fonte está disponível no endereço <http://code.google.com/p/vogal/>.

3.2.1 Estrutura do arquivo de dados

O tipo de tabela Vogal é basicamente um grande arquivo de 100MB dividido em cem mil blocos de 1KB (Figura 8) onde as tabelas dos metadados ocupam os oito blocos iniciais. A quantidade de blocos ocupada por uma tabela varia de acordo com a quantidade de colunas e quantidade de dados existentes.

Cada bloco possui um cabeçalho e um corpo (Figura 9). No cabeçalho é definido o tipo do bloco e se ele é válido ou não. Quando um arquivo de dados for inicializado todos os blocos são marcados como não válidos, e a partir do momento em que vão sendo utilizados, são marcados como válidos. A partir do momento que um bloco não é mais utilizado, ele é marcado novamente como não válido. Essa informação de validade do bloco é importante para, ao inicializar o banco de dados, sejam identificados quais blocos estão livres.

Existem dois tipos de bloco: `tabela` e `coluna`. No corpo de ambos existe uma árvore B. Em decorrência da variabilidade do tamanho dos registros, há a necessidade de manter a informação de quantidade que significa a quantidade de nós da árvore que estão no nó bloco atual.

Quando o tipo de bloco for `tabela` a árvore é ordenada pelo RID (Figura 10) que mantém junto ao nó do RID as informações de deslocamentos de suas colunas. Quando o tipo de bloco for `coluna` a árvore é ordenada pelo valor da coluna e que é complementada pelo RID do registro (Figura 11), possibilitando assim o inter-relacionamento.

A informação de deslocamento das colunas é formada pelo deslocamento físico do bloco no arquivo de dados complementado pela informação de deslocamento do nó da árvore no bloco.

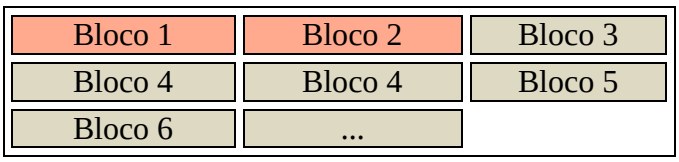


Figura 8 - Estrutura do arquivo de dados

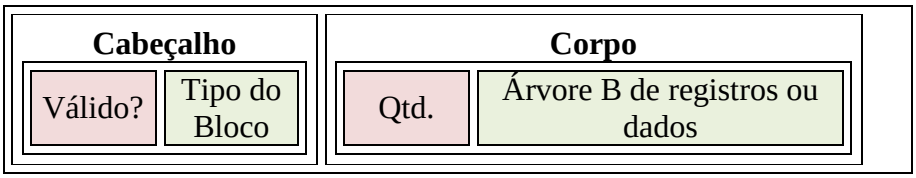


Figura 9 - Estrutura de um bloco genérico

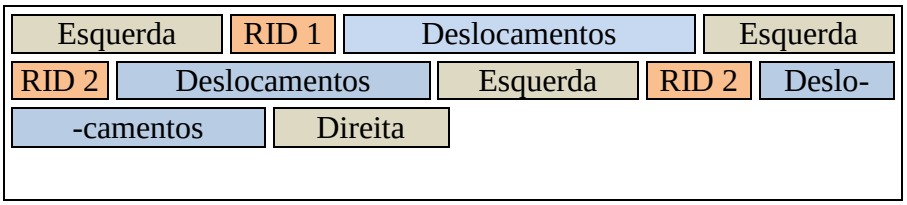


Figura 10 - Estrutura do nó da árvore de registros

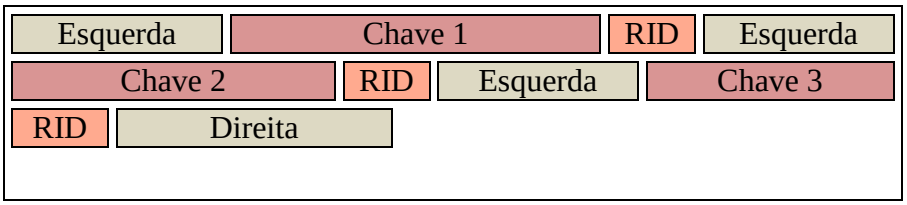


Figura 11 - Estrutura do nó da árvore de dados

Todos os dados gravados na base de dados são de tamanho variável, ou seja, a gravação do valor 1 em um campo numérico com precisão 8 ocupará apenas 2 *bytes*. Isso ocorre porque todos os dados têm 1 *byte* de identificação de tamanho. Este *byte* tem 7 *bits* de informação de tamanho e 1 *bit* que identifica se deve ser lido um novo byte de definição de tamanho. Dessa forma, com 1 *byte* é possível medir campos de até 127 *bytes* (2^7), com 2 *bytes*, campos de até 16384 *bytes* (2^{14}), e assim por diante, com um tamanho teoricamente ilimitado.

Como o objetivo da proposta é evitar redundância e desperdício de espaço em disco, o espaço que seria subutilizado é, dessa forma, diminuído drasticamente. Como a operação de conversão destes dados ocorre em memória o ganho de desempenho na leitura da base de dados potencialmente supera a perda na verificação dos limites. A indexação em árvore B foi preferida em relação a árvore B+ em virtude da necessidade de menor redundância e complexidade.

O dicionário de dados segue o mesmo padrão de estrutura e é formado pelas tabelas:

a) `OBJS`: coleção dos objetos do Tipo de Tabela, neste caso tabelas, composta pelas colunas `NAME` (Nome do objeto), `TYPE` (Tipo do objeto. Como somente são tratadas tabelas, o valor sempre será `TABLE`) e `LOCATION` (Localização do bloco inicial da árvore de dados da tabela no arquivo de dados);

b) `COLS`: coleção das colunas das tabelas do Tipo de Tabela, composta pelas colunas `TABLE_RID` (RID da tabela mãe da coluna), `NAME` (Nome da coluna), `TYPE` (Tipo da

coluna. Apenas dois tipos são tratados: `NUMBER` e `VARCHAR`) e `LOCATION` (Localização do bloco inicial da árvore de dados da coluna no arquivo de dados).

3.2.2 Diagrama de casos de uso

Na Figura 12 são demonstrados os casos de uso do projeto.

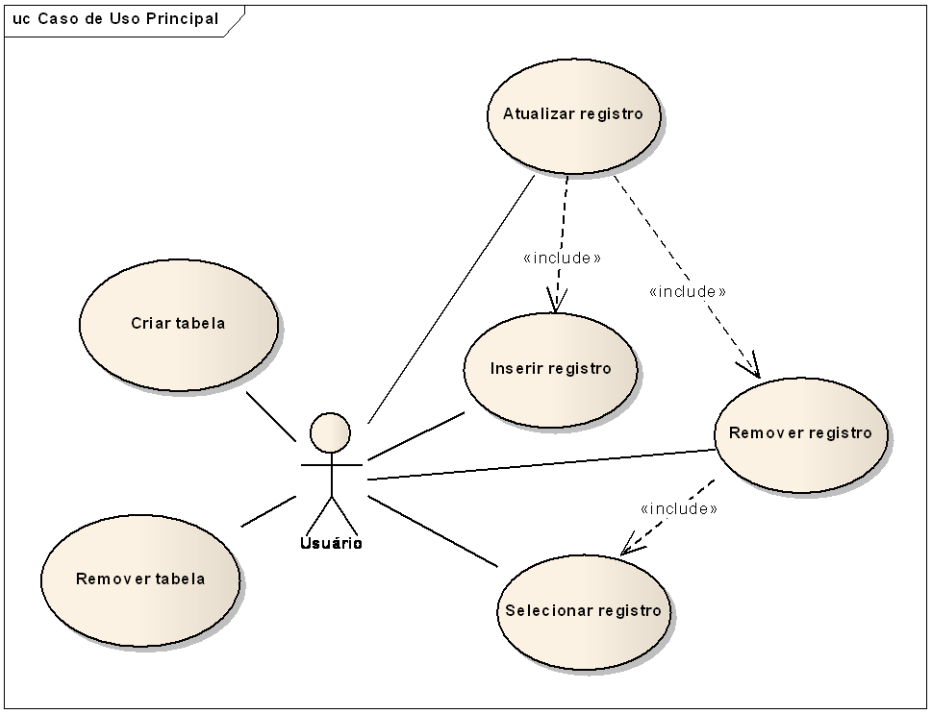


Figura 12 - Diagrama de casos de uso

Os casos de uso visualizados na Figura 12 são:

a) `Criar tabela`: permite ao usuário criar uma tabela (Quadro 1);
b) `Remover tabela`: permite ao usuário remover uma tabela (Quadro 2);
c) `Inserir registro`: permite ao usuário inserir um registro (Quadro 3);
d) `Atualizar registro`: permite ao usuário atualizar um registro (Quadro 4);
e) `Remover registro`: permite ao usuário remover um registro (Quadro 5);
f) `Selecionar registro`: permite ao usuário selecionar um registro (Quadro 6).

Caso de Uso	**`Criar tabela`**
Resumo	Através de um cliente[10] o usuário redige uma instrução SQL para criação de uma tabela do tipo Vogal.
Ator	Usuário
Pré-condição	O usuário deve estar conectado a um banco de dados MySQL existente através de qualquer cliente que permita instruções SQL.
Pós-condição	Tabela criada no banco de dados.
Fluxo principal	1. O usuário cria uma instrução *Data Definition Language* (DDL) [11] para criação de uma tabela do tipo Vogal; 2. O usuário comanda a execução desta instrução DDL no servidor através de opção disponível no cliente; 3. O sistema cria a tabela no esquema.
Exceções	2. O sistema gera erro na criação de uma tabela caso outra tabela possua mesmo nome; 2. O sistema gera erro na criação de uma tabela caso a tabela tenha colunas com nomes iguais.

Quadro 1 - Caso de uso `Criar tabela`

Caso de Uso	**`Remover tabela`**
Resumo	Através de um cliente o usuário redige uma instrução SQL para remoção de uma tabela do tipo Vogal.
Ator	Usuário
Pré-condição	O usuário deve estar conectado a um banco de dados MySQL existente através de qualquer cliente que permita instruções SQL.
Pós-condição	Tabela removida do banco de dados.
Fluxo principal	1. O usuário cria uma instrução DDL para remoção de uma tabela do tipo Vogal; 2. O usuário comanda a execução desta instrução DDL no servidor através de opção disponível no cliente; 3. O sistema remove a tabela no esquema.
Exceções	2. O sistema gera erro na remoção de uma tabela que não existe.

Quadro 2 - Caso de uso `Remover tabela`

[10] Cliente, no paradigma Cliente/Servidor, significa uma aplicação que envia solicitações para um servidor, que processa a solicitação, gera uma resposta e a envia ao cliente. O MySQL é um banco de dados cliente/servidor. Portanto, cliente neste contexto representa uma aplicação que se comunica com o servidor através de uma linguagem comum, no caso: SQL.

[11] DDL é um subconjunto de instruções SQL que tem como característica a definição/alteração da esquema de dados.

Caso de Uso	**`Inserir registro`**
Resumo	Através de um cliente o usuário redige uma instrução SQL para inserção de um registro em uma tabela do tipo Vogal.
Ator	Usuário
Pré-condição	O usuário deve estar conectado a um banco de dados MySQL existente através de qualquer cliente que permita instruções SQL.
Pós-condição	Registro inserido na tabela.
Fluxo principal	1. O usuário cria uma instrução *Data Manipulation Language* (DML) [12] para inserção de um registro em uma tabela do tipo Vogal; 2. O usuário comanda a execução desta instrução DML no servidor através de opção disponível no cliente; 3. O sistema insere o registro na tabela.
Exceções	2. O sistema gera erro na inclusão do registro caso a tabela não exista.

Quadro 3 - Caso de uso `Inserir registro`

Caso de Uso	**`Atualizar registro`**
Resumo	Através de um cliente o usuário redige uma instrução SQL para atualização de um registro em uma tabela do tipo Vogal.
Ator	Usuário
Pré-condição	O usuário deve estar conectado a um banco de dados MySQL existente através de qualquer cliente que permita instruções SQL.
Pós-condição	Registro atualizado na tabela.
Fluxo principal	1. O usuário cria uma instrução DML para atualização de um registro em uma tabela do tipo Vogal; 2. O usuário comanda a execução desta instrução DML no servidor através de opção disponível no cliente; 3. O sistema exclui o registro na tabela; 4. O sistema insere o registro na tabela.
Fluxo alternativo	A partir do item 2 do fluxo principal: 2.1. O sistema não consegue localizar o registro de acordo com a instrução recebida; 2.2. O sistema não exclui nem insere o registro na tabela.
Exceções	2. O sistema gera erro na atualização do registro caso a tabela não exista.

Quadro 4 - Caso de uso `Atualizar registro`

[12] DML é um subconjunto de instruções SQL que tem como característica a manipulação de dados em uma tabela existente no esquema de dados.

Caso de Uso	**`Remover registro`**
Resumo	Através de um cliente o usuário redige uma instrução SQL para remoção de um registro em uma tabela do tipo Vogal.
Ator	Usuário
Pré-condição	O usuário deve estar conectado a um banco de dados MySQL existente através de qualquer cliente que permita instruções SQL.
Pós-condição	Registro removido na tabela.
Fluxo principal	1. O usuário cria uma instrução DML para remoção de um registro em uma tabela do tipo Vogal; 2. O usuário comanda a execução desta instrução DML no servidor através de opção disponível no cliente; 3. O sistema localiza o registro de acordo com a instrução recebida; 4. O sistema remove o registro da tabela.
Fluxo alternativo	A partir do item 2 do fluxo principal: 2.1. O sistema não consegue localizar o registro de acordo com a instrução recebida; 2.2. O sistema não remove o registro na tabela.
Exceções	2. O sistema gera erro na remoção do registro caso a tabela não exista.

Quadro 5 - Caso de uso `Remover registro`

Caso de Uso	**`Selecionar registro`**
Resumo	Através de um cliente o usuário redige uma instrução SQL para seleção de um registro em uma tabela do tipo Vogal.
Ator	Usuário
Pré-condição	O usuário deve estar conectado a um banco de dados MySQL existente através de qualquer cliente que permita instruções SQL.
Pós-condição	Registro removido na tabela.
Fluxo principal	1. O usuário cria uma instrução DML para remoção de um registro em uma tabela do tipo Vogal; 2. O usuário comanda a execução desta instrução DML no servidor através de opção disponível no cliente; 3. O sistema localiza o registro de acordo com a instrução recebida; 4. O sistema remove o registro da tabela.
Fluxo alternativo	A partir do item 2 do fluxo principal: 2.1. O sistema não consegue localizar o registro de acordo com a instrução recebida; 2.2. O sistema não remove o registro na tabela.
Exceções	3. O sistema gera erro na remoção do registro caso a tabela não exista.

Quadro 6 - Caso de uso `Selecionar registro`

3.2.3 Diagrama de classes

A Figura 13 ilustra o diagrama de classes do *storage engine* Vogal e para valorizar o aspecto legibilidade foram omitidos os atributos de cada classe e seus métodos. A interface `handler` é a estrutura de ligação entre o MySQL e o *storage engine*, ou seja, nela estão representados os métodos que são suportados e necessários para que integração ocorra corretamente e na classe `ha_vogal` estes métodos são implementados.

Em seguida são detalhadas algumas classes consideradas essenciais para o sistema. No apêndice A o diagrama representado na Figura 13 pode ser consultado sem omissões.

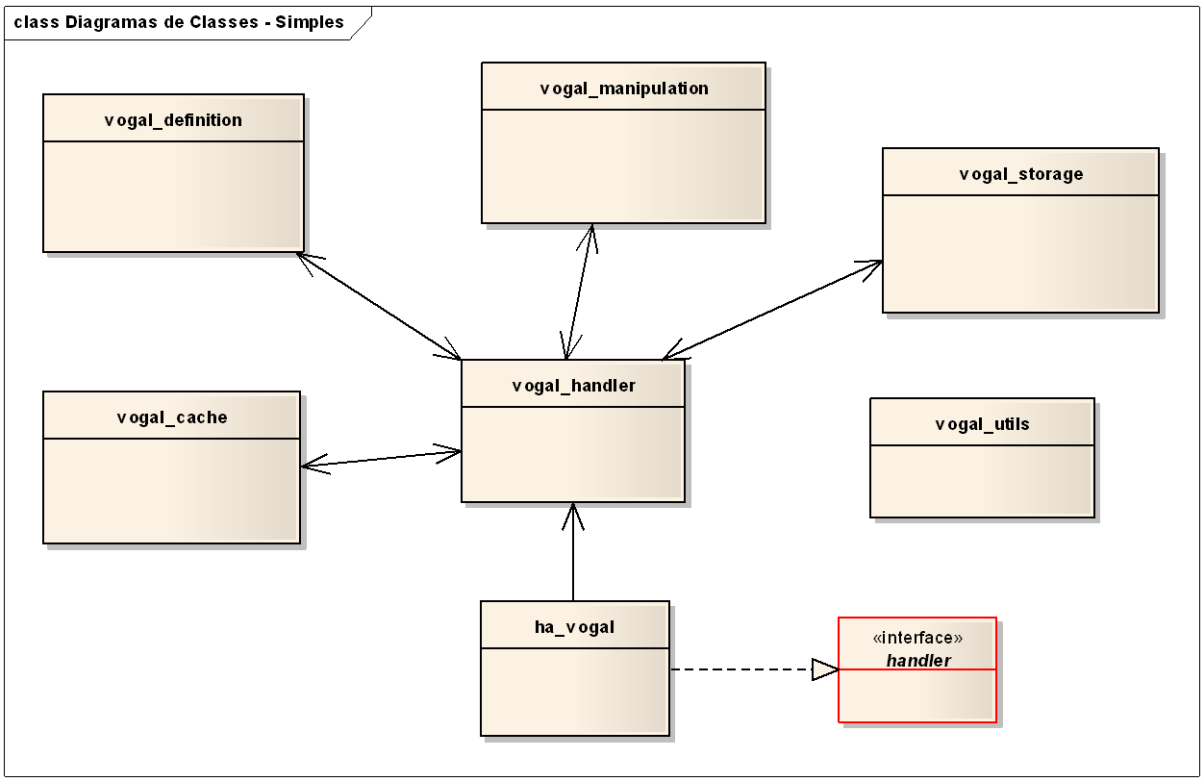


Figura 13 - Diagrama de classes do *storage engine*

Na Figura 14 é apresentado o diagrama de classes das classes auxiliares que são utilizadas como meio de interação entre as classes principais.

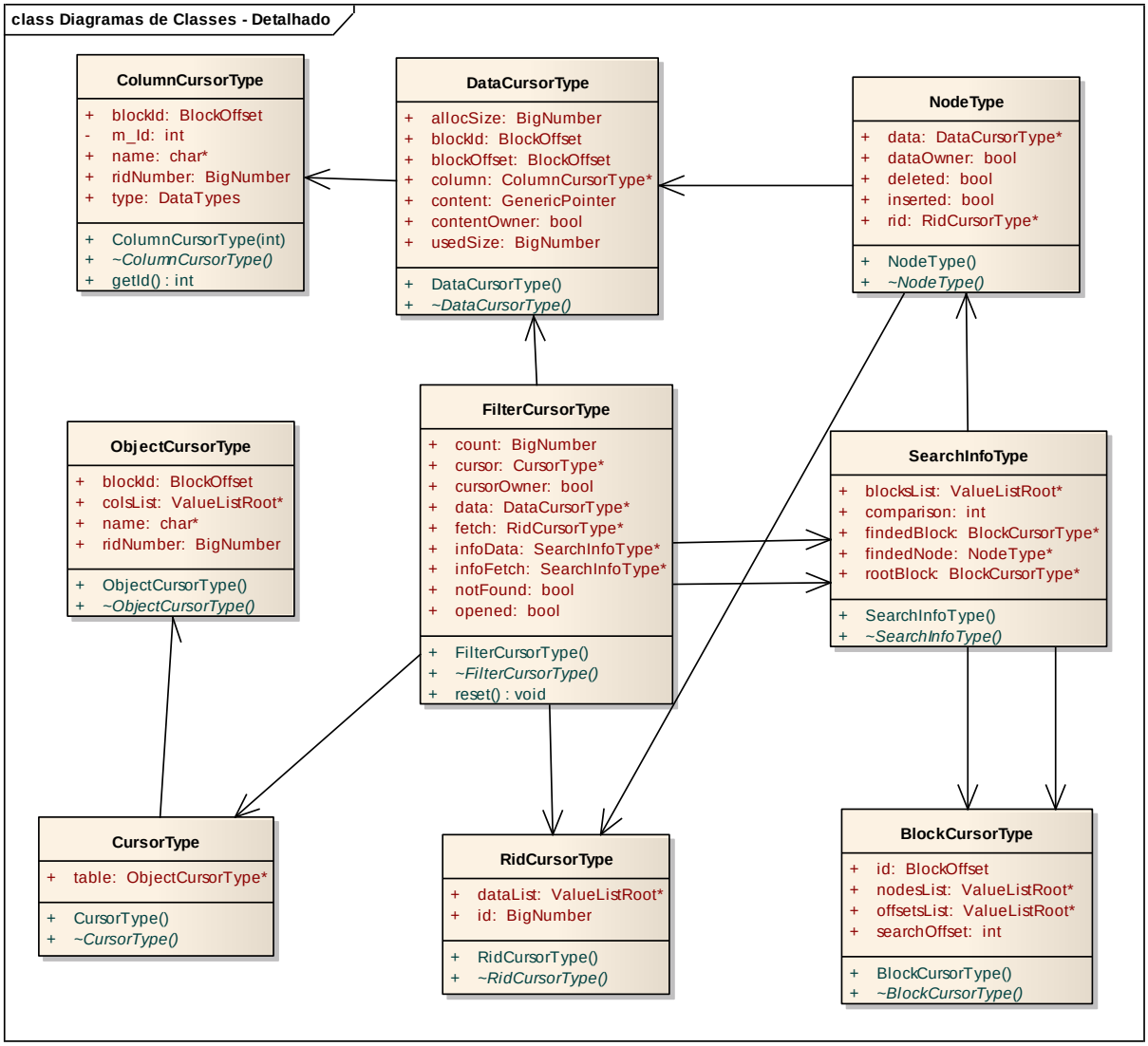


Figura 14 - Diagrama de classes das classes auxiliares do *storage engine*

3.2.3.1 Classes de gerenciamento

As classes de gerenciamento são: `ha_vogal` (Figura 15), `vogal_definition` (Figura 17), `vogal_manipulation` (Figura 16), `vogal_storage` (Figura 18) e `vogal_cache` (Figura 19).

A classe `ha_vogal` é a implementação da interface de comunicação com a API dos *storage engines*. Toda requisição do banco de dados passa por este componente que gerencia a comunicação entre o banco de dados com as classes especializadas: `vogal_definition` (definição das tabelas e organização destas informações dentro dos blocos de dados), `vogal_manipulation` (manipulação dos registros nas tabelas), `vogal_storage` (armazenamento e recuperação dos blocos em disco) e `vogal_cache` (otimização de

memória). Ao receber a chamada `open` da API do *storage engine* a classe `ha_vogal` aciona o método `openDatabase`. Este, caso o arquivo do banco de dados não exista, cria-o através da classe `vogal_storage` e então inicia o processo de criação do dicionário de dados através do método `createDataDictionary` que gerencia a interação entre as classes `vogal_storage` e `vogal_definition`. Assim que o dicionário de dados estiver criado, ou o arquivo de banco de dados já existir, o método `openDatabase` abre este arquivo e inicia o processo `initialize` que inicializa a operação de gerenciamento de memória na classe `vogal_cache` que por sua vez inicializa o método `bufferize` que carrega em memória informações relevantes ao banco de dados para otimizar seu desempenho.

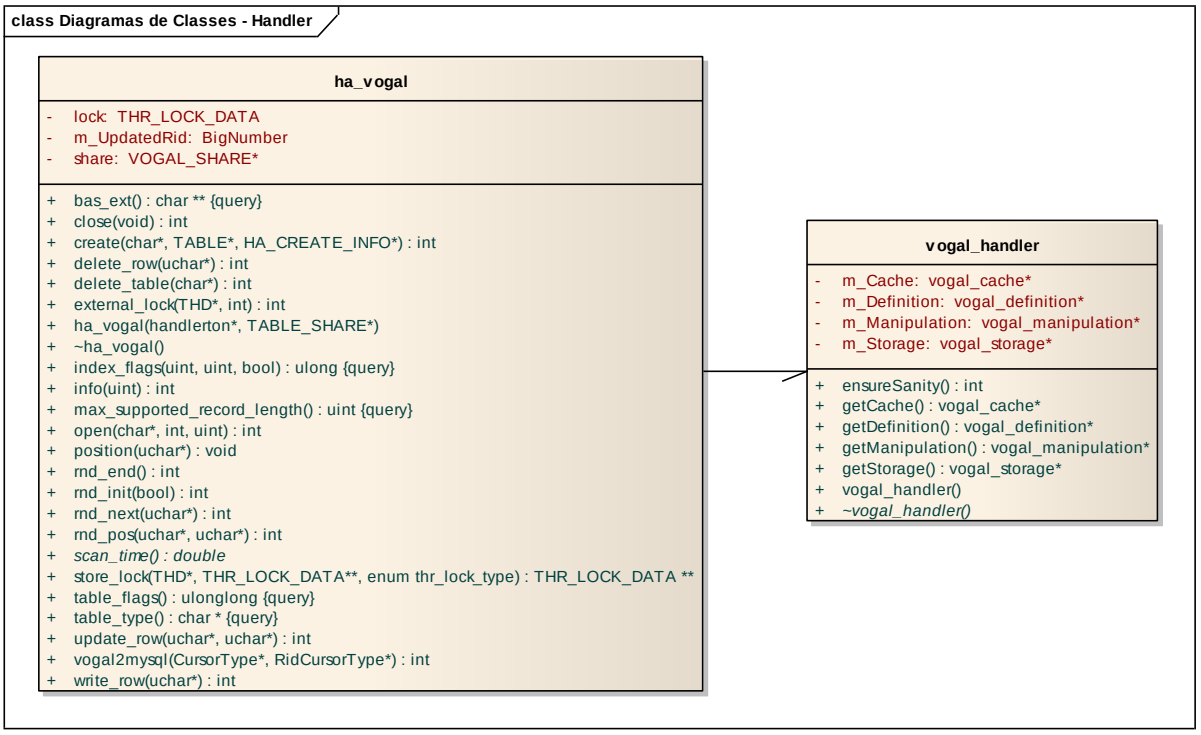


Figura 15 - Definição da classe `ha_vogal` e `vogal_handler`

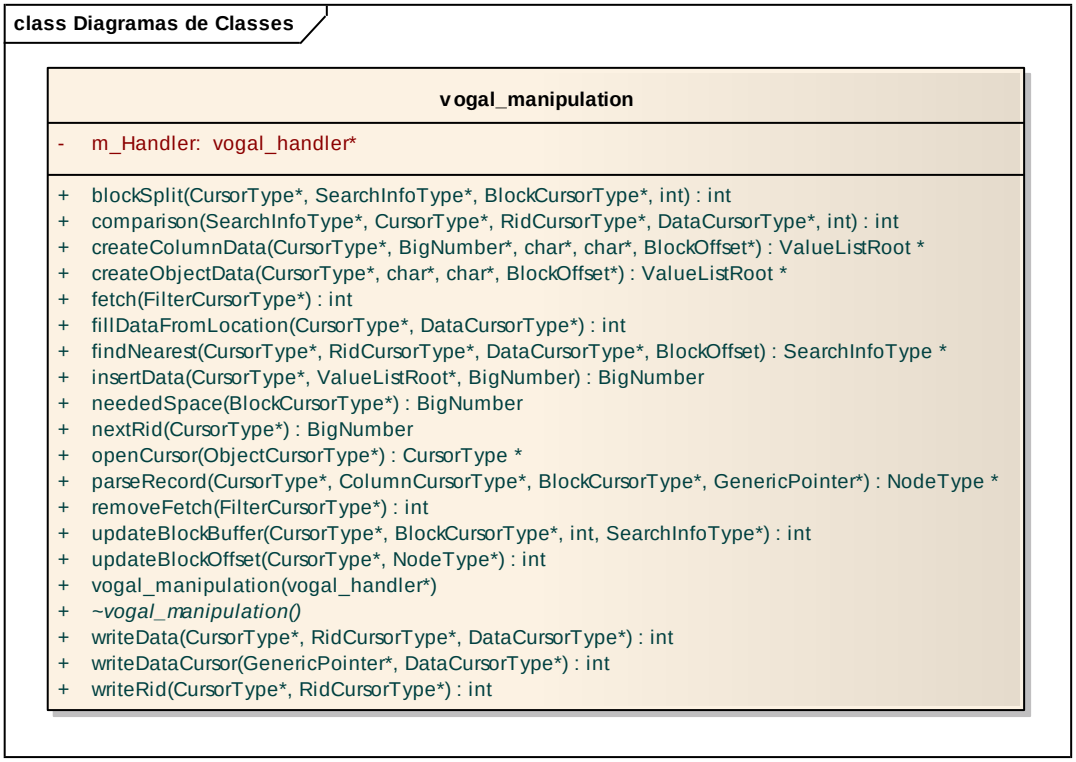


Figura 16 - Definição da classe `vogal_manipulation`

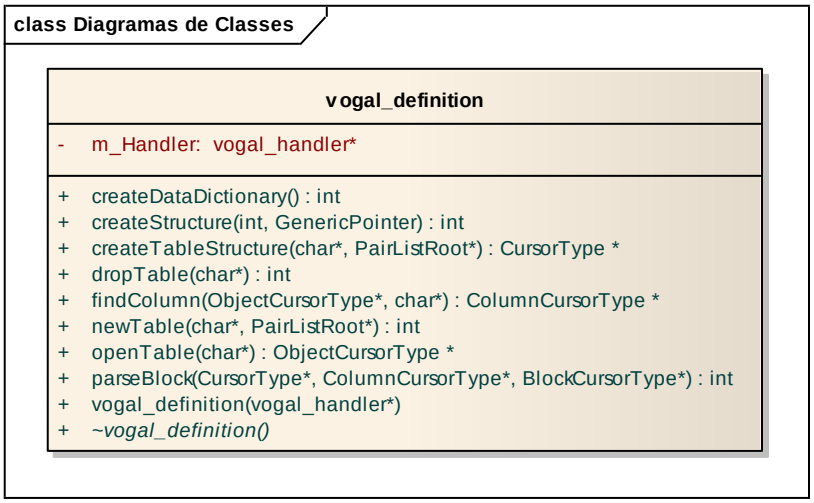


Figura 17 - Definição da classe `vogal_definition`

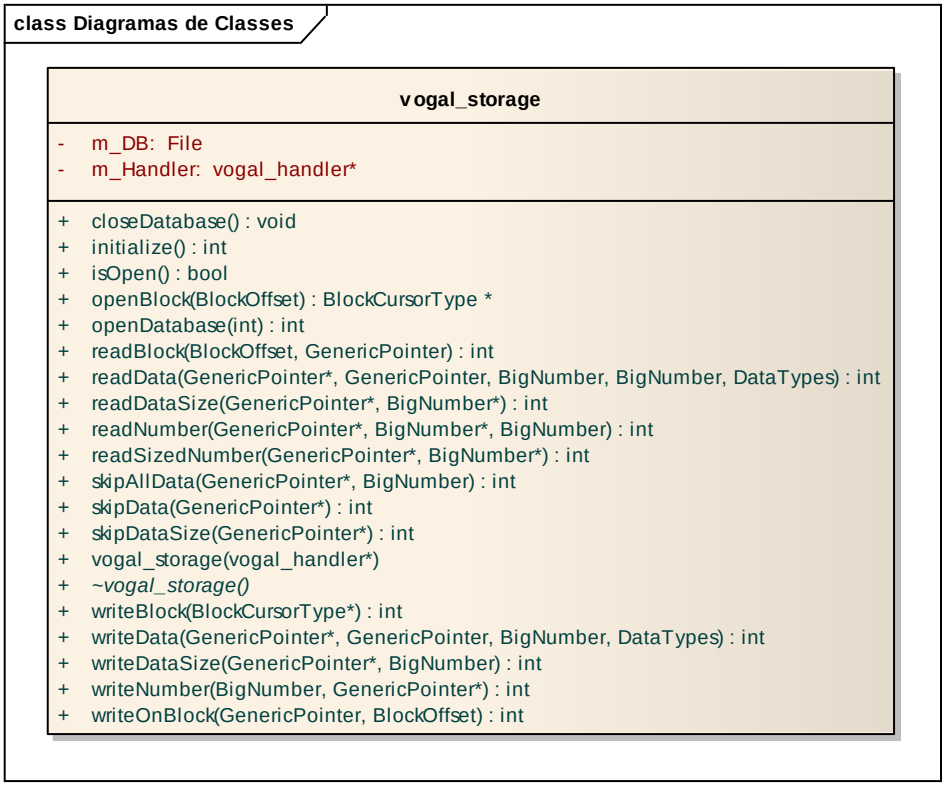


Figura 18 - Definição da classe `vogal_storage`

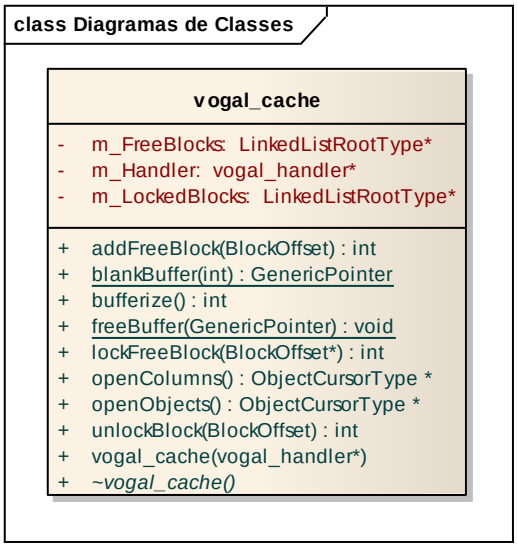


Figura 19 - Definição da classe `vogal_cache`

Os métodos da interface que representam alguma interação com índices não serão implementados, pois o projeto Vogal não permite a criação de índices secundários e o mecanismo de indexação das colunas implementado pelo sistema é apresentado à API como um registro atômico comum, ou seja, para a API o tipo de tabela Vogal é uma coleção de registros sem índices.

A partir do momento em que é executado o método `ensureSanity` da classe `vogal_handler`, é executado processo de inicialização da base de dados utilizando o método

`openDatabase` da classe `ha_vogal`, fica a critério do usuário quais operações ele deseja realizar no banco, ou seja, as inserções, exclusões, criação de tabelas, etc. ocorrem na ordem que o usuário desejar, impossibilitando a definição de uma sequência lógica.

Quando usuário criar uma instrução DDL de criação de tabela em um cliente, o método `create` da API é chamado e então o método `newTable` da classe `vogal_definition` é disparado. Este método verifica no dicionário de dados, utilizando informações da classe `vogal_cache`, se a tabela existe ou não e caso não exista, cria a tabela através do método `createTableStructure`, o qual obtém o espaço de um bloco de memória e define a estrutura da tabela e repassa essa informação para a classe `vogal_storage` para persistência. Da mesma forma, caso o método `delete_table` da API for executado, é então chamado o método `dropTable` da classe de `vogal_definition`, a qual localiza a tabela no banco e viabiliza sua remoção.

Se o usuário executar uma instrução DML para qualquer operação sobre algum registro, é executado o método `open` da API. Este por sua vez, mediado pela classe `ha_vogal`, chama o método `openTable` da classe `vogal_definition`, o qual abre as tabelas do dicionário de dados para verificar a existência da tabela e obter sua definição e ponteiros para localização da tabela e suas colunas dentro do arquivo do banco de dados. Existindo a tabela, através da `vogal_manipulation` é executado a escrita e leitura do registro de acordo com a operação solicitada. No caso da exclusão e consulta a leitura dos índices é feita para localização do registro, e se a operação for de exclusão, este é removido. No caso da inserção e atualização é feito uma leitura nos índices pra escrita no local apropriado de cada coluna dentro de seu índice, se for atualização, é feito primeiro a operação de exclusão do registro. Estas operações ocorrem através da classe `vogal_manipulation` e principalmente pelos métodos `openCursor`, `fetch`, `insertData`, `removeFetch`, `readRead`, `writeRid`, entre outros.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas, parte do código desenvolvido e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

Todo o ambiente de desenvolvimento se encontra em uma máquina virtual disponível através da ferramenta VirtualBox da Sun Microsystems. Esta máquina virtual tem o Linux instalado na distribuição do Ubuntu. O protótipo, independente do MySQL, foi implementado em C no ambiente Eclipse. Deste protótipo surgiram as principais estruturas que foram integradas ao MySQL utilizando o editor de texto VIsual (VI) que está disponível em praticamente todas as distribuições do sistema operacional Linux e o editor visual Geany. O código fonte foi compilado utilizando o Make e o GCC, todos disponibilizados junto à distribuição do Linux. A depuração, quando necessária, foi efetuada através da ferramenta Data Display Debugger (DDD) que implementa um ambiente visual de depuração através do utilitário GNU DeBbuger (GDB). As figuras dos códigos fonte apresentados neste trabalho foram feitas usando o editor Notepad++.

O MySQL na sua versão 5.1.34-0 foi primordial para implementação pela infra-estrutura por causa da existência de *storage engines* e pelos testes por causa da possibilidade acesso ao banco de dados. Como ferramenta de execução de instruções SQL no banco de dados foi utilizada o MySQL Navigator.

3.3.2 Código desenvolvido

É importante salientar que os códigos apresentados nos quadros deste tópico podem ter sofrido modificações no seu leiaute quando comparados ao código original visando destacar alguns aspectos, porém sem nunca perderem sua lógica básica de funcionamento.

O desenvolvimento iniciou com a implementação de um protótipo que abrangia grande parte dos requisitos do sistema. Porém, não integrado à API dos *storage engines* do MySQL. Neste protótipo foram criadas as estruturas de controle e tipos padrão para utilização no sistema (Quadro 7). Estas estruturas são importantes, pois não foram modeladas como classes e são utilizadas como tipos básicos em grande parte do sistema.

```
// Tipos de blocos
#define C_BLOCK_TYPE_MAIN_TAB 0
#define C_BLOCK_TYPE_MAIN_COL 1

typedef enum {NUMBER,FLOAT,DATE,VARCHAR,UNKNOWN} DataTypes; // Tipos dos campos

typedef unsigned long long int BigNumber; // Campo numerico com maior tamanho possível
typedef unsigned char * GenericPointer; // Ponteiro generico para os dados
typedef BigNumber BlockOffset; // Localização dos blocos

typedef struct {
        unsigned char valid:1; // true/false
        unsigned char type  :3; // tabela/coluna
        unsigned char       :4; // 4 bits sobram
} BlockHeaderType;

typedef struct {
        unsigned char size:7; // 0 - 127
        unsigned char more:1; // true/false
} VariableSizeType;

typedef struct {
        BlockOffset id; // posição do bloco
        BlockOffset offset; // offset do registro no bloco
} BlockPointerType;
```

Quadro 7 - Estruturas de controle e tipos padrão

Antes de implementar as classes gerenciadoras do sistema, há a necessidade de criação de mecanismo de coleções. Foram criados quatro tipos de coleção: lista de faixas (`LinkedListRoot`), lista de valores (`ValueListRoot`), lista de nomes e valores (`PairListRoot`) e árvores (`StringTreeRoot`). A lista de faixas procura otimizar listas de números, economizando memória e melhorando o desempenho de busca e atualização. A lista sequencial é uma lista dinâmica simples cuja funcionalidade foi estendida na lista de nomes e valores. A árvore é uma estrutura de árvore binária em memória que tem um nome como chave.

A primeira necessidade em uma estrutura de armazenamento é a definição de seu dicionário de dados e existe um problema relativo à sua criação. Este problema de criação ocorre pois o dicionário de dados é a informação básica para criação de todas as estruturas, ou seja, não é possível criar uma rotina de criação de tabelas e utilizá-la para criar o dicionário de dados pois ela, por definição, consulta o dicionário de dados para criar novas estruturas. Desta forma é necessário que o dicionário de dados seja criado de forma independente do método de criação de tabelas. Mas antes de criar o dicionário de dados, há a necessidade de definição do formato de escrita e leitura destas informações. Os principais métodos de escrita são exemplificados no Quadro 8. Os métodos de leitura são análogos aos de escrita e são o `readNumber`, `readString`, `readSizedDada` e `readData` e estão na classe `vogal_storage`.

```
// Escreve buffer de um bloco no arquivo
int vogal_storage::writeOnBlock(GenericPointer buf, BlockOffset block){
        // Posiciona no arquivo
        if (my_seek(m_DB, BI2BO(block), SEEK_SET, MYF(MY_WME)) == BI2BO(block))
                // Grava no arquivo
                if (my_write(m_DB, buf, C_BLOCK_SIZE, MYF(MY_WME)) == C_BLOCK_SIZE)
                        // Remove da pilha de pendências
                        if (m_Handler->getCache()->unlockBlock(block))
                                return true; }

// Escreve número no buffer
int vogal_storage::writeNumber(BigNumber number, GenericPointer* where){
        int ret = writeData(where, (GenericPointer)&number, sizeof(BigNumber), NUMBER);}

// Escreve dado generico no buffer
int vogal_storage::writeData(GenericPointer* dest, GenericPointer src, BigNumber
                             dataBytes, DataTypes dataType){
        // Em números bytes zerados à direita significam zeros à esquerda.
        if(dataType == NUMBER)
                while ((dataBytes>1) && (!(*(src + dataBytes - 1))))
                        dataBytes--; break;
        // Grava o tamanho do dado
        if (!writeDataSize(dest, dataBytes))
                ERROR("Erro ao gravar tamanho do dado!");
        // Grava o dado
        memcpy((*dest), src, dataBytes);
        (*dest) += dataBytes; }

// Grava buffer de um bloco no arquivo
int vogal_storage::writeBlock(BlockCursorType* block){
        return writeOnBlock(block->buffer, block->id); }

// Escreve o tamanho dinâmico dos campos
int vogal_storage::writeDataSize(GenericPointer* dest, BigNumber size){
        do {
                VariableSizeType * var_p = (VariableSizeType *)(*dest);
                var_p->size = size & (unsigned char)127; // Tamanho máximo
                size >>= 7; // Desloca 7 bits pra direita
                var_p->more = size > 0;
                (*dest)+=sizeof(VariableSizeType);
        } while (size > 0); }
```

Quadro 8 - Métodos de escrita de dados nos blocos

Definido os métodos de escrita e leitura pode-se criar o dicionário de dados. Em seguida devem ser coletadas informações do arquivo de banco de dados criado de forma a otimizar o desempenho. No Quadro 9 é exemplificado o método onde são lidos todos os blocos do arquivo e que armazena essa informação em uma lista de faixa.

```
// Lê os blocos vazios do arquivo e armazena em memória
int vogal_cache::bufferize(){
        // Bufferizando blocos vazios de dados
        GenericPointer buf = blankBuffer();
        // Posiciona no primeiro bloco que não é de controle
        for (long int offset = C_OBJECTS_COLS_COUNT + C_COLUMNS_COLS_COUNT + 2;
             m_Handler->getStorage()->readBlock(offset, buf); offset++) {
                BlockHeaderType * header = (BlockHeaderType *) buf;
                // Verifica se o primeiro bit está ativo, se não estive, está vazio
                if (!header->valid)
                        addFreeBlock(offset);
        }
        freeBuffer(buf);
        // Lista blocos livres
        llDump(m_FreeBlocks); }
```

Quadro 9 - Método de coleta dos blocos livres

Em diversas partes do sistema é necessário reservar alguns blocos para gravação e evitar que outros métodos utilizem-no antes que o método solicitante efetive sua gravação.

Esta reserva é feita na classe `vogal_storage` nos métodos `lockFreeBlock` (reserva bloco) e `unlockBlock` (cancela reserva do bloco). A reserva é feita movimentado o bloco solicitado entre duas coleções na classe `vogal_cache`: `freeBlocks` (lista de faixas de blocos livres) e `lockedBlocks` (lista de faixas de blocos reservados).

Uma vez o dicionário de dados criado e o método de escrita definido é possível efetuar todas as operações ao qual este trabalho se propõe: criar e remover tabelas, consultar, inserir, alterar e remover registros. Antes de qualquer uma destas operações o método de `openTable` da classe `vogal_definition` (Quadro 10). No caso da criação de tabelas, este método é chamado para garantir sua inexistência. Na remoção de tabelas, exatamente o contrário. Porém, nas outras operações este método é essencial para organizar em memória as definições da tabela na classe `ObjectCursorType`, as informações que foram previamente persistidas no armazenamento secundário. Desta forma, a tabela de tabelas, denominada `OBJS`, e a tabela de colunas, denominada `COLS`, são acessadas como tabelas normais e podem ser consultadas inclusive pelo usuário final.

```
// Abertura e inicialização de tabelas
ObjectCursorType * vogal_definition::openTable(char * tableName) {
        // Começa a inicializar a tabela
        ObjectCursorType * table = new ObjectCursorType();
        table->name = tableName; table->colsList = NULL;
        // Se está tentando abrir a tabela de colunas, a inicialização deve ser manual
        if (!strcmp(tableName,C_COLUMNS)) { ### SUPRIMIDO ### }
        // Se está tentando abrir a tabela de objetos, a inicialização deve ser manual
        if (!strcmp(tableName,C_OBJECTS)) { ### SUPRIMIDO ### }
        // Cria objeto de filtro
        FilterCursorType objsFilter = new FilterCursorType();
        objsFilter->cursor = m_Handler->getManipulation()->openCursor
               (m_Handler->getCache()->openObjects());
        objsFilter->data = new DataCursorType();
        objsFilter->data->content = (GenericPointer) tableName;
        objsFilter->data->usedSize = strlen(tableName);
        objsFilter->data->allocSize = objsFilter->data->usedSize + 1;
        objsFilter->data->column = findColumn(objsFilter->cursor->table, C_NAME_KEY);
        // Procura a tabela
        if (!m_Handler->getManipulation()->fetch(objsFilter)) ERROR("Erro no fetch!");
        if (objsFilter->notFound) goto freeOpenTable;
        ColumnCursorType colLocation = findColumn
               (objsFilter->cursor->table, C_LOCATION_KEY);
        DataCursorType dataLocation = (DataCursorType *) vlGet
               (objsFilter->fetch->dataList, colLocation->getId());
        // Achando, abre o bloco da tabela
        table->blockId = (*(BigNumber*) dataLocation->content);
        table->ridNumber = objsFilter->fetch->id;
        // Obtém as colunas da tabela
        FilterCursorType colsFilter = new FilterCursorType();
        colsFilter->cursor = m_Handler->getManipulation()->openCursor
               (m_Handler->getCache()->openColumns());
        colsFilter->data = new DataCursorType();
        colsFilter->data->content = (GenericPointer) &objsFilter->fetch->id;
        colsFilter->data->usedSize = sizeof(BigNumber);
        colsFilter->data->allocSize = colsFilter->data->usedSize;
        colsFilter->data->column = findColumn
               (colsFilter->cursor->table, C_TABLE_RID_KEY);
        // Varre as colunas da tabela
        table->colsList = vlNew(true);
        do {
                if (!m_Handler->getManipulation()->fetch(colsFilter))
                        ERROR("Impossível efetuar o fetch das colunas!");
                if (colsFilter->notFound)
                        break; // Fim dos dados
                ColumnCursorType * column = new ColumnCursorType
                      (vlCount(table->colsList));
                column->ridNumber = colsFilter->fetch->id;
                for (int i = 0; i < vlCount(colsFilter->fetch->dataList); i++) {
                        DataCursorType * data = (DataCursorType *) vlGet
                             (colsFilter->fetch->dataList, i);
                        if (!strcmp(data->column->name, C_NAME_KEY)) {
                                column->name = (char*) data->content;
                        } else if (!strcmp(data->column->name, C_TYPE_KEY)) {
                                column->type = vogal_utils::str2type
                                     ((char*) data->content);
                        } else if (!strcmp(data->column->name, C_LOCATION_KEY)) {
                                column->blockId = (*(BigNumber *) data->content ); } }
                if (!vlAdd(table->colsList, column))
                        goto freeOpenTable;
        } while (true);
freeOpenTable:
        return table; }
```

Quadro 10 - Definição do método de abertura de tabelas

Com a tabela aberta é possível efetuar as operações desejadas sobre ela. Para consultas, alterações e exclusões de registros é chamado o método `fetch` (Quadro 11) da classe `vogal_manipulation`. Este método movimenta o objeto `ObjectCursorType` pela tabela até localizar o registro desejado sendo então possível efetivar a operação desejada. Caso a

operação seja apenas consulta, o método `rnd_next` da API será chamado para efetuar a consulta de cada registro obtido e o `ha_vogal` gerenciará adequadamente a chamada.

Os métodos de escrita e leitura são principalmente o `writeRid` (Quadro 12) e o `readRid` (Quadro 13), respectivamente, e estão localizados na classe `vogal_manipulation`.

```
// Efetua a leitura das linhas da tabela
int vogal_manipulation::fetch(FilterCursorType * filter){
	if (!filter || !filter->cursor || !filter->cursor->table)
		ERROR("Cursor de filtro inválido para efetuar o fetch!");
	filter->fetch = NULL;
	if (filter->notFound)
		goto fetchEmpty;
doNext:
	if (filter->data) {
		if (!filter->opened) {
			// Obtém o rid do dado a ser localizado
			filter->infoData = findNearest(filter->cursor, NULL, filter->data,
				filter->data->column->blockId);
			if (!filter->infoData)
				ERROR("Erro ao tentar localizar o dado!");
			filter->opened = true;
		} else {
			// Varredura da árvore
			if (!filter->infoData->findedBlock)
				ERROR("Nenhum bloco aberto na busca inicial");
			if (!comparison(filter->infoData, filter->cursor, NULL, filter->data))
				ERROR("Erro durante o processo de comparação dos dados "); }
	} else {
		if (!filter->opened) {
			// Obtém o rid do dado a ser localizado
			filter->infoFetch = findNearest(filter->cursor, NULL, NULL,
				filter->cursor->table->blockId);
			if (!filter->infoFetch)
				ERROR("Erro ao tentar localizar o rid!");
			filter->opened = true;
		} else {
			// Varredura da árvore
			if (!filter->infoFetch->findedBlock)
				ERROR("Nenhum bloco aberto na busca inicial!");
			if (!comparison(filter->infoFetch, filter->cursor, NULL, NULL))
				ERROR("Erro durante o processo de comparação dos dados!"); } }
	// Verifica grau de identificação, no caso, tem que ser igual!
	if (filter->infoData) {
		if (filter->infoData->notFound)
			goto fetchEmpty;
		// Obtém as localizações de todos os dados do rid localizado
		filter->infoFetch = findNearest(filter->cursor,
		    filter->infoData->findedNode->rid, NULL, filter->cursor->table->blockId); }
	if (filter->infoFetch->notFound)
		goto fetchEmpty;
	filter->fetch = filter->infoFetch->findedNode->rid;
	if (filter->infoFetch->findedNode->deleted) // Excluído não vale
		goto doNext;
	// Por enquanto obtém todas as colunas
	// TODO: Pegar somente as colunas solicitadas
	for (int d = 0; d < vlCount(filter->fetch->dataList); d++) {
		DataCursorType* data = (DataCursorType*) vlGet(filter->fetch->dataList, d);
		if (!fillDataFromLocation(filter->cursor, data))
			ERROR("Erro ao preencher as colunas com os dados!"); }
	filter->count ++;
fetchEmpty:
	filter->notFound = true;
	return true }
```

Quadro 11 - Definição do método para consulta de registros

```
// Grava cada dado da tabela
int vogal_manipulation::writeData(CursorType * cursor, RidCursorType * rid,
                                  DataCursorType * data) {
        // Procura o dado mais próximo
        SearchInfoType *info = findNearest(cursor, rid, data,
                data?data->column->blockId:cursor->table->blockId);
        // Cria novo nodo e insere no local adequado
        NodeType *node = new NodeType();
        node->rid = rid;
        node->data = data;
        node->inserted = true;
        vlInsert(info->findedBlock->nodesList, node, info->findedBlock->searchOffset);
        // Esquerda e direita apontando para o vazio!
        BlockOffset *neighbor = new BlockOffset();
        (*neighbor) = 0;
        vlInsert(info->findedBlock->offsetsList, neighbor, info->findedBlock->searchOffset);
        // Somente insere vizinho da direita se for o primeiro rid do bloco
        if (vlCount(info->findedBlock->offsetsList) ==
            vlCount(info->findedBlock->nodesList) == 1) {
                neighbor = new BlockOffset();
                (*neighbor) = 0;
                vlInsert(info->findedBlock->offsetsList, neighbor,
                        info->findedBlock->searchOffset + 1); }
        if (!updateBlockBuffer(cursor, info->findedBlock, info))
                ERROR("Erro ao atualizar o buffer do bloco!");
        if (!m_Handler->getStorage()->writeBlock(info->findedBlock))
                ERROR("Erro ao gravar bloco de dados!");
        if (info)
                info->~SearchInfoType(); }
```

Quadro 12 - Definição do método para escrita de registros

```
// Lê buffer carregado do arquivo para a memória e separa as informações relevantes
int vogal_definition::parseBlock(CursorType * cursor, ColumnCursorType * column,
                                 BlockCursorType * block) {
        // Monta lista de rids e offsets do bloco
        block->nodesList = vlNew(true);
        block->offsetsList = vlNew(true);
        // Posição atual
        GenericPointer p = block->buffer + sizeof(BlockHeaderType);
        // Lê a quantidade de registros do bloco
        BigNumber dataCount;
        if (!m_Handler->getStorage()->readDataSize(&p, &dataCount))
                ERROR("Erro ao ler a quantidade de registros no block");
        if (dataCount > 0)
                for (int i = 0; i < dataCount; i++) {
                        // Auxiliares
                        BlockOffset *neighbor = new BlockOffset();
                        // Lê primeiro nó a esquerda
                        if (!m_Handler->getStorage()->readSizedNumber(&p, neighbor))
                                ERROR("Erro ao ler nó a esquerda da árvore de registros");
                        // Adiciona informação à lista
                        vlAdd(block->offsetsList, neighbor);
                        NodeType *node = m_Handler->getManipulation()->parseRecord
                                (cursor, column, block, &p);
                        // Adiciona informação à lista
                        vlAdd(block->nodesList, node); }
        // Se a quantidade for maior que zero, depois do último nó sempre tem o nó a direita
        neighbor = new BlockOffset();
        if (!m_Handler->getStorage()->readSizedNumber(&p, neighbor))
                ERROR("Erro ao ler nó a direita da árvore de registros");
        // Adiciona informação à lista
        vlAdd(block->offsetsList, neighbor); }
```

Quadro 13 - Definição do método de leitura do registro

Existem outros métodos utilizados no sistema Vogal, porém, de menor relevância à tecnologia proposta, portanto, aqui, não destacados.

3.3.3 Operacionalidade da implementação

Inicialmente o usuário deve acessar um cliente que disponibilize uma interface de comunicação com um servidor MySQL e permita a elaboração de instruções SQL, e conectar a um banco de dados previamente criado. O servidor MySQL ao qual o cliente estiver conectado deve dar suporte ao tipo de tabela Vogal.

Para criar uma tabela o usuário deve elaborar uma instrução DDL semelhante ao definido no Quadro 14 e comandar a execução da instrução no servidor. É de suma importância que o usuário informe o tipo da tabela a ser criada (cláusula `TYPE = Vogal`), caso contrário a tabela será criada com o tipo de tabela padrão do MySQL (normalmente MyISAM).

Após criar uma tabela, o usuário pode incluir um registro através de uma instrução DML semelhante ao Quadro 15. Assim que a tabela estiver populada com dados é possível alterar (Quadro 16) e excluir registros (Quadro 17). Para excluir uma tabela deve ser executada uma instrução como a do Quadro 18. No Quadro 19 há um exemplo de uma instrução DML para consultar algum registro. Executando esta consulta após a criação da tabela e a inserção do dado apresentado nos exemplos, tem-se o resultado apresentado na Figura 20. E na Figura 21 é exibido o conteúdo binário do arquivo de dados. O trecho apresentado possibilita identificar os nomes das colunas existentes no metadados.

```
CREATE TABLE Vendedor (
        Codigo INT,
        Descricao VARCHAR(50)
) TYPE = Vogal
```

Quadro 14 - DDL de criação de tabela do tipo Vogal

```
INSERT INTO Vendedor
        (Codigo, Descricao)
VALUES (10, 'José')
```

Quadro 15 - DML de inserção de registro

```
UPDATE Vendedor
   SET Descricao = 'João'
 WHERE Codigo = 10
```

Quadro 16 - DML de alteração de registro

```
DELETE FROM Vendedor
 WHERE Codigo = 10
```

Quadro 17 - DML de exclusão de registro

```
DROP TABLE Vendedor;
```

Quadro 18 - DDL de remoção de tabela

```
SELECT *
  FROM Vendedor
 WHERE Codigo = 10
```

Quadro 19 - DML de consulta de registro

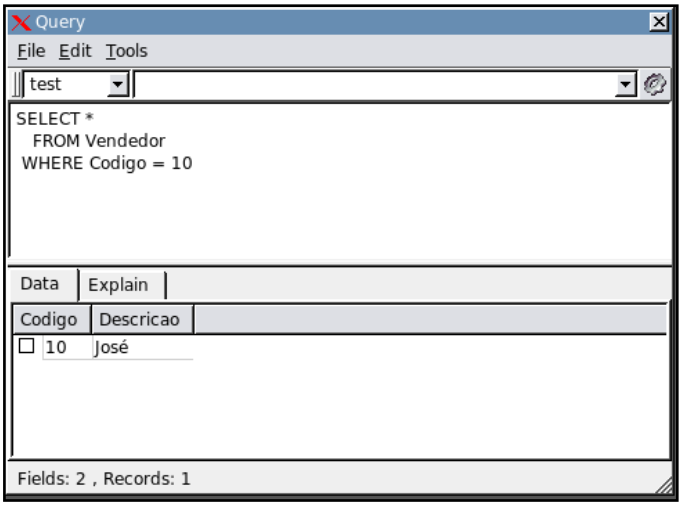


Figura 20 - Consulta e resposta do banco de dados

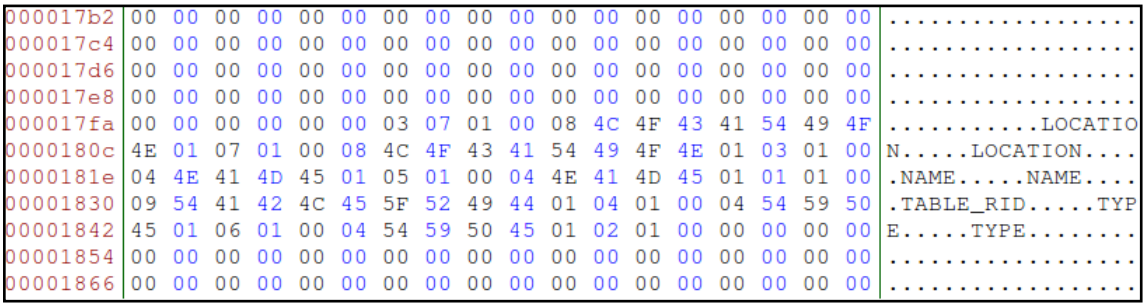


Figura 21 - Estrutura binária do arquivo de dados

3.4 RESULTADOS E DISCUSSÃO

O tipo de tabela Vogal foi construído com o intuito de diminuir a redundância e aumentar a performance em consultas pontuais. Inicialmente foi escolhido o tipo de indexação Árvore B+ para as colunas. Porém, por definição, este tipo de indexação gera informação redundante. Dessa forma, foi utilizado o tipo de indexação Árvore B normal. Este traz mesmo grau de balanceamento, porém com mais desperdício de espaço nos blocos utilizados, contudo, sem redundância.

Outro problema foi de atualização de ponteiros, pois no conceito do projeto, as colunas são como índices independentes, mas esses índices devem estar ligados de alguma forma uns aos outros para possibilitarem a formação de um registro. Dessa forma, uma tabela, além dos

índices correspondentes às colunas, há um índice correspondente aos RIDs, ou seja, as colunas apontam para um RID e o RID aponta para as colunas. Como o RID poderia apontar para as colunas? Caso as colunas também tivessem uma numeração sequencial, seriam necessários dois índices para manter uma só coluna, o que tornou essa metodologia inviável. A melhor forma encontrada foi o RID apontar para a posição física do nó da árvore do índice, ou seja, seu bloco e deslocamento. Esta metodologia permite um acesso direto aos dados da coluna em caso de não utilização desta em nenhuma busca. Porém traz um problema bastante grave: este deslocamento pode ser atualizado para cada atualização da coluna, ou seja, se um bloco possuir dados de vinte registros, os RIDs desses registros devem ter o deslocamento referente às colunas modificadas atualizado, e é bastante comum que estes RIDs estejam em blocos independentes. Portanto, segundo o exemplo, para a atualização de um registro, pode ser necessário gravar vinte e um blocos. Os testes mais adiante trarão resultados mais claros.

O gerenciamento de memória também trouxe problemas, pois, ao contrário do que se imaginava inicialmente, o MySQL não faz este gerenciamento para os *storage engines*. Estes devem implementar um gerenciamento individual. Contudo, para uma completa otimização de um sistema, o gerenciamento de memória não pode ser ignorado, e este é um dos grandes segredos dos grandes bancos de dados. Portanto, não há tempo hábil para criação de um gerenciamento de memória tão eficiente quanto as grandes construções (InnoDB, MyISAM, entre outros), mesmo estas sendo de código aberto, pois estas estruturas estão fortemente acopladas à arquitetura de escrita e leitura. Em virtude disso, o gerenciamento de memória é mínimo, procurando otimizar os mecanismos mais simples, como a obtenção de blocos livres.

A documentação disponível no site do MySQL não deixa claro se este é desenvolvido em C ou C++. Apenas no momento que é lida a documentação que vem junto ao código fonte do sistema que é destacada a informação que o MySQL é desenvolvido em C++ e é orientado a objetos. Também é possível constatar isso estudando o código fonte. Quando do início do desenvolvimento do protótipo ainda não se tinha esta informação e assim o protótipo foi desenvolvido em C em programação estruturada. Para não perder o que foi desenvolvido no protótipo, parte da programação estruturada foi mantida, ou seja, a programação do *storage engine* não é completamente orientada a objetos em virtude do tempo disponível para seu desenvolvimento.

Em consequência aos problemas encontrados e do tempo disponível, foi necessário estabelecer algumas limitações ao projeto. Algumas destas limitações são comparadas ao tipo de tabela Maria no Quadro 20.

Limitações \ Tipo de tabela	Maria	Vogal
controle transacional	Implementa	Não implementa
integridade referencial	Implementa	Não implementa
ACID[13]	Não implementa	Não implementa
tamanho máximo do banco de dados	256TB	100MB
tamanho máximo de cada registro	64KB	1KB
máximo de colunas por tabela	4096	1024
tamanho máximo de campos LOB	4GB	N/A
tamanho máximo de campos de texto	64KB	127B
tamanho máximo de campos numéricos	64 *bits*	32 *bits*
menor data válida	1000	N/A
maior data válida	9999	N/A

Quadro 20 - Comparativo de limitações dos tipos de tabela Vogal e Maria

A arquitetura de gravação dos campos torna-os virtualmente ilimitados em uma tabela do tipo Vogal. Tanto o tamanho dos registros, quanto a quantidade de registros ou o tamanho máximo do banco de dados. Porém, as limitações são aqui destacadas até o ponto onde puderam ser testadas e confirmadas.

[13] ACID é acrônimo de *Atomic, Consistency, Isolation, Durability*, cuja tradução é Atômico, Consitente, Isolado e Durável. Transações implementadas de forma apropriada satisfazem as características ACID sendo que, para uma transação ser atômica significa que é na base de tudo ou nada; Para ser isolada, a transação deve ser executada como se nenhuma outra transação estivesse em execução ao mesmo tempo; Tem a característica de durável a transação que sua conclusão garante a persistência da informação no banco de dados; Para ser consistente a informação não pode sofrer mudanças sem que seja sobre ação direta do usuário (GARCIA-MOLINA; ULLMAN; WIDOM, 2001, p. 10).

4 DESENVOLVIMENTO DA PROVA DE CONCEITO

Neste capítulo é apresentado o desenvolvimento da prova de conceito que compreende o levantamento dos requisitos da prova, sua implementação e os resultados obtidos.

4.1 REQUISITO PRINCIPAL DO PROBLEMA A SER TRABALHADO

A prova de conceito deve traçar uma análise comparativa entre o tipo de tabela proposto e os tipos InnoDB e MyISAM do MySQL (RF).

4.2 ESPECIFICAÇÃO

Para atender o requisito do projeto é necessário definir o que é relevante ser testado e a que resultados chegar. Baseando-se nas comparações de NG (2009), inicialmente é necessário delinear claramente quais são as limitações e quais são as capacidades dos tipos de tabela testados. Como limitações serão verificados os seguintes itens:

a) tamanho máximo do banco de dados;
b) tamanho máximo de cada registro;
c) máximo de colunas por tabela;
d) tamanho máximo de campos *Large Object* (LOB);
e) tamanho máximo de campos de texto;
f) tamanho máximo de campos numéricos;
g) menor data válida;
h) maior data válida.

Os seguintes itens são capacidades do Tipo de Tabela. As capacidades consideradas básicas, como consulta, inserção, exclusão e atualização não são consideradas.

a) união e diferença (o banco de dados MySQL não implementa intersecção);
b) junção interna;
c) junção externa;

d) consulta interna (*sub-select*);
e) LOBs;
f) domínio;
g) programa;
h) ACID;
i) integridade referencial;
j) controle transacional.

Embora as limitações e capacidades sejam importantes para escalar um tipo de tabela, o foco desta prova de conceito é a performance do mesmo. Para tal, baseado nos testes realizados por Pires, Nascimento e Salgado (2008), serão feitos testes de carga, estrutura e consulta. Este teste será realizado com volume de cem mil registros. Como testes de carga e estrutura serão realizados as seguintes verificações:

a) criação de tabela;
b) carregamento da mesma com dados;
c) atualização de um registro;
d) atualização de 100% dos registros;
e) exclusão de um registro;
f) exclusão de 100% dos registros.

Como testes de consulta serão realizados as seguintes validações:

a) consulta de 100% dos registros com projeção total das colunas;
b) consulta de 100% dos registros com projeção parcial (metade das colunas);
c) consulta de 10% dos registros dos registros com projeção total das colunas;
d) consulta de 10% dos registros com projeção parcial;
e) consulta de um registro com projeção total das colunas;
f) consulta de um registro com projeção parcial.

Todos os testes serão realizados em um servidor MySQL local instalado em uma máquina virtual com o sistema operacional Linux na distribuição Ubuntu. Esta máquina virtual é executada reservando 320MB de memória e 20GB de disco rígido em um computador com processador Intel Dual Core T2300 @ 1.66GHz com 1GB de memória RAM DDR2 533 com sistema operacional Microsoft Windows XP Professional Service Pack 3.

4.3 IMPLEMENTAÇÃO

As limitações e capacidades dos tipos de tabela InnoDB e MyISAM são obtidas através da documentação disponível no manual de referência do MySQL de autoria de Dubois (2008) ou através de testes diretos na base de dados.

O OSDB já possui uma implementação bastante completa para testes sobre o MySQL. Portanto, o Quadro 21 demonstra um exemplo parcial do resultado de um teste do OSDB com mil registros, ou seja, para executar os testes de carga, estrutura e consulta, serão necessárias duas iterações (com mil e cem mil registros) para cada tipo de tabela testado.

```
OSDB 0.21

Invoked: osdb-my --mysql=innodb --nomulti --noindexes --restrict MySQL
Compile-time options: none
Restrictions: insert_into rollback subqueries views

                create_tables()    0.07 seconds    return value = 0
                          load()    0.07 seconds    return value = 0

Logical database size 0.4000MB (0.3815MiB, 1000 tuples/relation)

                     join_3_cl()    0.01 seconds    return value = 0
                sel_10pct_ncl()    0.01 seconds    return value = 0
                     join_2_cl()    0.01 seconds    return value = 0
                     join_4_cl()    0.01 seconds    return value = 0
                      proj_100()    0.01 seconds    return value = 966
                    proj_10pct()    0.01 seconds    return value = 100
                     bulk_save()    0.04 seconds    return value = 0
                   bulk_modify()    0.01 seconds    return value = 0
                upd_mod_t_mid()    0.01 seconds    return value = 0
                upd_mod_t_end()    0.02 seconds    return value = 0
                upd_mod_t_cod()    0.01 seconds    return value = 0
                  bulk_append()    0.01 seconds    return value = 0

Single User Test    0.20 seconds    (0:00:00.20)
```

Quadro 21 - Exemplo execução OSDB

No Quadro 22 é feito o mapeamento das funções de teste do OSBD com os resultados definidos na especificação. Contudo, houve a necessidade de criar funções específicas no OSBD para que fosse possível executar os testes necessários para obter os resultados previstos.

	Resultado esperado	Função OSDB
Carga e estrutura	criação das tabelas	`create_tables`
	carga de dados	`load`
	atualização de um registro	`upd_unique`
	atualização de 100% dos registros	`upd_100pct`
	exclusão de um registro	`del_unique`
	exclusão de 100% dos registros	`del_100pct`
Consulta	100% dos registros (projeção total)	`selproj_100_100pct`
	100% dos registros (projeção parcial)	`selproj_100_50pct`
	10% dos registros (projeção total)	`selproj_10_100pct`
	10% dos registros (projeção parcial)	`selproj_10_50pct`
	um registro (projeção total)	`selproj_1_100pct`
	um registro (projeção parcial)	`selproj_1_50pct`

Quadro 22 - Mapeamento dos resultados esperados com as funções do OSDB

4.3.1 Técnicas e ferramentas utilizadas

O OSDB foi utilizado como ferramenta de teste da implementação sobre a mesma plataforma configurada para o ambiente de desenvolvimento do Tipo de Tabela.

4.3.1.1 Código desenvolvido

Para criar um determinado volume de dados deve ser executado o comando `osdb-my --generate-files --size 40mb --datadir /tmp` onde o parâmetro `size` corresponde ao volume de dados criado para os testes. Para criar um volume de cem mil linhas é necessário um tamanho de aproximadamente 40MB. Depois de criado um determinado volume é iniciado a execução do teste a partir do comando `osdb-my --mysql=innodb --nomulti --noindexes --restrict MySQL`. Onde o parâmetro `mysql` identifica em que *storage engine* as tabelas serão criadas, o parâmetro `nomulti` indica que o teste não será executado simulando múltiplos usuários e o parâmetro `noindexes` desativa a criação de índices. O parâmetro `restrict MySQL` não executa os testes que contenham instruções `INSERT INTO`, `ROLLBACK`, `VIEW` ou *subqueries*. Essa parametrização é importante para equilibrar as características dos tipos de tabelas para um teste mais próximo da realidade, diminuindo as diferenças causadas por otimizações que não existam no Tipo de Tabela Vogal.

4.4 RESULTADOS E DISCUSSÃO

O tipo de tabela Vogal possui limitações e capacidades definidas de acordo com este projeto. As limitações dos tipos de tabela são apresentadas no Quadro 23 e as capacidades dos tipos de tabela são apresentadas no Quadro 24.

Limitações \ Tipo de tabela	InnoDB	MyISAM	Vogal
tamanho máximo do banco de dados	64TB	256TB	100MB
tamanho máximo de cada registro	8KB	64KB	1KB
máximo de colunas por tabela	1000	4096	1024
tamanho máximo de campos LOB	4GB		N/A
tamanho máximo de campos de texto	64KB		127B
tamanho máximo de campos numéricos	64 *bits*		32 *bits*
menor data válida	1000		N/A
maior data válida	9999		N/A

Quadro 23 - Limitações dos tipos de tabela

Capacidades \ Tipo de tabela	InnoDB	MyISAM	Vogal
união, diferença	Sim	Sim	Não
junção interna			
junção externa			
consulta interna (*sub-select*)			
LOBs			
domínio			
programa			
ACID			
integridade referencial		Não	
controle transacional			

Quadro 24 - Capacidades dos tipos de tabela

O resultado dos testes de carga, estrutura e consulta com um volume de cem mil registros são destacados na Quadro 25. Estes valores são a média de cinco execuções e estão representados em segundos.

	Testes (segundos) \ Tipos de tabela	InnoDB	MyISAM	Vogal
Carga e estrutura	criação das tabelas	0,06	0,03	40
	carga de dados	19,16	4,77	77000 (~22h)
	atualização de um registro	0,32	0,81	30
	atualização de 100% dos registros	2,69	2,65	65000 (~18h)
	exclusão de um registro	1,44	0,12	40
	exclusão de 100% dos registros	1,75	0,01	2400 (~40m)
Consulta	100% dos registros (projeção total)	2,26	1,91	40
	100% dos registros (projeção parcial)	1,25	1,14	20
	10% dos registros (projeção total)	0,52	0,37	30
	10% dos registros (projeção parcial)	0,36	0,25	20
	um registro (projeção total)	0,34	0,13	30
	um registro (projeção parcial)	0,70	0,14	40

Quadro 25 - Resultados dos testes com cem mil registros

5 CONCLUSÕES

O objetivo geral deste trabalho, implementar um *storage engine*, realizar um *benchmark* e analisar o seu resultado foi alcançado em sua plenitude, assim como os objetivos específicos. Porém, de acordo com o resultado dos testes, os ganhos esperados, principalmente de desempenho, não foram atingidos.

O tipo de tabela Vogal é uma implementação de estrutura de dados que tem como objetivo quebrar o paradigma que os campos de um registro em um banco de dados devem permanecer juntos fisicamente. Dentre os benefícios desta mudança estariam principalmente a diminuição da redundância de dados e o aumento do desempenho.

O MySQL já implementa as operações principais do SQL como junção, união, funções de grupo e filtro e estas funcionalidades também estão disponíveis no tipo de tabela Vogal. Apesar de não estarem contemplados no projeto, ou seja, integridade referencial, filtros complexos, agrupamentos de dados, entre outros recursos disponíveis no SGDB podem ser utilizados nas tabelas de tipo Vogal.

Na arquitetura foi implementado um mecanismo de campo dinâmico que, através de um *byte* é possível delimitar quaisquer tamanhos de campo de forma ilimitada. Esse tipo de modelagem é importante para atender uma possível expansão da implementação. Porém foram atribuídos limites para que seja possível testá-los e validá-los no tempo disponível para a confecção desta monografia.

As seguintes limitações estão devidamente elencadas nos requisitos do projeto e correspondem aos principais itens identificados. Não são apenas itens que não funcionam na aplicação, são também recursos e limites não plenamente validados:

a) apenas são permitidos campos de tipo texto ou numérico inteiro;
b) os tipos texto serão limitados a 127 caracteres;
c) os tipos numéricos deverão estar entre -2147483648 e 2147483647 (inteiro de 32 *bits*);
d) as limitações de tamanho na definição do campo não são validadas.

Três itens vieram de encontro ao fator desempenho do Tipo de Tabela: falta de um gerenciamento de memória adequado que reduzisse o acesso a disco, a não implementação do mecanismo de índices disponibilizado pela API do MySQL e, como nos blocos de RIDs existem endereços para os dados, qualquer atualização nos blocos dos dados implica na necessidade de reorganização dos deslocamentos a cada atualização da base de dados.

O MySQL tem um gerenciamento de memória, mas este é completamente independente do tipo de tabela. Portanto, ao contrário do que se acreditava no início do projeto, este deve ser implementado por completo no *storage engine*. E assim o foi porém, de forma precária.

O mecanismo que a API do MySQL dispõe para utilização de índices não foi implementado pela razão de o tipo de tabela Vogal não criar índices secundários. Contudo, é necessário que o MySQL entenda que o *storage engine* possui formas de localizar de maneira eficiente a informação que ele necessita, ou seja, o SGDB faz toda preparação do plano de execução do SQL informado pelo usuário (ou aplicação cliente). Este plano é formado através de estatísticas obtidas do tipo da tabela, entre as quais estão informações de quais os melhores índices a serem ou não utilizados, de forma a otimizar esse plano de execução. A não implementação desse recurso no *storage engine* fez com que o MySQL entendesse toda consulta do tipo de tabela Vogal como varredura completa da tabela. Dessa forma, para otimizar as consultas é necessário implementar os métodos da API que são relacionados a índices, informando ao MySQL que o tipo de tabela Vogal possui índices porém, de uma forma diferente.

Os dados do Tipo de Tabela são organizados em duas estruturas: árvore de RIDs e árvore de registros. A primeira estrutura grava o valor do RID e o endereço (bloco e deslocamento) do dado na segunda. E a árvore de registros grava o valor do dado e o valor do RID, ou seja, caso a informação seja localizada pelo dado, é possível localizar o RID na árvore de RIDs através do número encontrado. Caso seja localizada através do RID, é possível obter o dado da coluna através de seu endereço. Porém, os dados mudam de posição dentro de um bloco de acordo com as atualizações impostas sobre eles ou seus irmãos. Isso imprime ao *storage engine* a necessidade de atualizar o endereço dos dados em suas respectivas árvores de RIDs. Assim, caso um bloco da árvore de registros possua cinco registros e seja inserido um novo registro que fique em primeiro no bloco, todos os registros subsequentes terão seu endereço alterado, necessitando um número maior de atualizações. Um gerenciamento de memória adequado poderia minimizar esse problema.

O tipo de tabela Vogal está disponibilizado para a comunidade de código aberto junto ao MySQL, esperando desta forma a contribuir e receber contribuições de quem se mostrar interessado.

5.1 EXTENSÕES

São possíveis inúmeras extensões a este projeto, dentre elas:

a) ultrapassar os limites do Tipo de Tabela definidos neste projeto para garantir que as estruturar possibilitem uma faixa de limite muito mais abrangente;
b) permitir os tipos de campo de ponto flutuante, data, LOBs, entre outros;
c) implementar um gerenciamento de memória mais robusto;
d) utilizar métodos da API do MySQL para utilização de índices;
e) desenvolver um gerenciamento de transações (se possível ACID);
f) possibilitar junções (externas, internas, naturais, entre outras);
g) adicionar funções de conjunto como união e interseção;
h) incluir funções de agrupamento: `group by`, `sum`, `avg`, `distinct`, entre outros;
i) validar obrigatoriedade e limites de campos;
j) aproveitar mecanismo de incremento automático no valor das colunas presente no MySQL;
k) implementar seleção por faixa (`LIMIT`);
l) configurar o arquivo de dados, seu tamanho, e adicionar possibilidade de que ele aumente automaticamente;
m) desenvolver mecanismo de rebalanceamento das árvores de índices;
n) implementar mecanismo de coleta de estatísticas dos índices para possibilitar a montagem de um plano de execução mais apurado;
o) compatibilizar com todos os sistemas operacionais em que o MySQL é compatível além do Linux;
p) melhorar o tratamento de erros para utilizar os mesmos padrões do desenvolvimento do MySQL aproveitando sua infra-estrutura de tratamento de exceções;
q) construir mecanismo de visões para possibilitar a consulta das tabelas do metadados;
r) melhorar a implementação da exclusão em árvore B.

5.1.1 Gerenciamento de memória

A classe `vogal_cache` é o módulo que tem como papel principal organizar as otimizações de memória do Tipo de Tabela. Portanto, deve ser nesta classe o desenvolvimento de um gerenciamento de memória mais robusto. Um primeiro passo é manter o metadados em memória. Isto já trará grandes ganhos por se tratar da estrutura primordial onde o acesso, na versão atual, ocorre desnecessária e repetidamente a disco.

5.1.2 Implementando tipos de dados adicionais

Na classe `vogal_utils` existe uma enumeração dos tipos esperados e na classe `vogal_storage` os métodos de escrita e leitura genéricos para os dados atualmente mantidos. Na classe `vogal_manipulation` está o método `comparison` que faz a comparação dos dados para localização na consulta. Portanto, é possível disponibilizar novos tipos de dados, simples ou complexos, utilizando apenas estas classes.

5.1.3 Validando obrigatoriedade e limites de campos

Na classe `ha_vogal` está implementada a interface de comunicação com a API do *storage engine* com o MySQL. Inclusive, nesta classe já existe uma varredura dos campos para mapeamento. Assim, possibilitando também a validação de obrigatoriedade e limites.

5.1.4 Utilizando os métodos sobre índices da API do MySQL

Como os métodos de sobre índices da API não foram implementados, o MySQL interpreta as tabelas do Tipo de Tabela como uma tabela sem índices. Portanto, as consultas se dão através de uma varredura completa de todas as colunas e registros derrubando toda e qualquer tentativa de otimização. Portanto, esta alteração se faz fundamental para se obter a eficiência almejada.

Todos os métodos sobre índices da API estão comentados na classe `ha_vogal`. Estes

métodos devem ser implementados utilizando a classe `vogal_manipulation` através do método `findNearest` sobre a coluna que se deseja consultar. Porém, isso não é o suficiente. É necessário também criar estruturas para coletar estatísticas importantes sobre as colunas como cardinalidade, densidade, entre outras. Portanto, o MySQL deve entender que o Tipo de Tabela é uma tabela com um índice por coluna, mesmo sendo essa uma só informação.

5.1.5 Compatibilizando o Tipo de Tabela com outros sistemas operacionais

Primordialmente é necessário definir um tipo de dados padrão, por exemplo, tipos inteiros devem ter sempre o mesmo formato no arquivo de dados em todos os sistemas operacionais, independentemente do processador ou arquitetura da placa. Em seguida é necessário levantar as principais diferenças de codificação em C++ inclusive elencando otimizações e limitações para acesso ao sistema de arquivos. O MySQL já é multi-plataforma. Portanto, a principal tarefa é compatibilizar os códigos fonte e a arquitetura do arquivo de dados.

5.1.6 Configurando o arquivo de dados

Ao final da classe `ha_vogal` existe uma estrutura comentada que se refere a obtenção de parâmetros dos arquivos de configuração do MySQL. A utilização destas estruturas se faz necessária para permitir configurar opções que estão fixadas em constantes, como o tamanho do arquivo de dados ou a possibilidade deste se redimensionar automaticamente. Algumas destas constantes estão definidas nas classes `vogal_utils` e `vogal_storage`.

REFERÊNCIAS BIBLIOGRÁFICAS

CROOK, Neal. **Brief introduction to alpha systems and processors**. [S.l.]: Digital Equipment, 1997. Disponível em: <http://www.alphalinux.org/docs/alpha-howto.html>. Acesso em: 23 out. 2008.

CYRAN, Michele et al. **Oracle® database**: concepts - 10g release 2. [S.l.], 2005. Disponível em: <http://download.oracle.com/docs/cd/B19306_01/server.102/b14220/toc.htm>. Acesso em: 20 out. 2008.

DUBOIS, Paul et al. **MySQL 5.1 reference manual**. [S.l.], 2008. Disponível em: <http://dev.mysql.com/doc/refman/5.1/en>. Acesso em: 20 out. 2008.

ELMASRI, Rasmez; NAVATHE, Shamkant B**. Sistemas de bancos de dados**. 4 ed. Revisor Técnico Luis Ricardo de Figueiredo. São Paulo: Pearson Addison Wesley, 2005.

GARCIA-MOLINA, Hector; ULLMAN, Jeffrey D.; WIDOM, Jennifer: **Implementação de sistemas de bancos de dados**. Tradução Vanderberg D. de Souza. Rio de Janeiro: Campus, 2001.

HEINZLE, Roberto. **Estrutura de dados**: implementações com C e Pascal. Blumenau: Diretiva, 2006.

MYSQL AB. **Why MySQL?** [S.l.], 2008. Disponível em: <http://www.mysql.com/why-mysql/>. Acesso em: 22 out. 2008.

MYSQL COMMUNITY. **MySQL internals**. [S.l.], 2008. Disponível em: <http://dev.mysql.com/doc/internals/en>. Acesso em: 20 out. 2008.

NG, Martin et al. **Comparison of relational database management systems**. [S.l.], 2009. Disponível em: <http://www.wikipedia.org>. Acesso em: 01 jun. 2009.

PIRES, Carlos E. D.; NASCIMENTO, Rilson O.; SALGADO, Ana C. Comparativo de desempenho entre bancos de dados de código aberto. In: ESCOLA REGIONAL DE BANCO DE DADOS. 2., 2006, Passo Fundo. **Anais**... Porto Alegre: Sociedade Brasileira de Computação, 2006. p. 21-26. Disponível em: <http://www.upf.br/erbd/download/15997.pdf> Acesso em: 30 out. 2008.

RAMAKRISHNAN, Raghu. **Database management systems**. Singapura: WCB McGraw-Hill, 1998.

RIEBS, Andy. **OSDB readme**. [S.l.], 2004. Disponível em: <http://osdb.cvs.sourceforge.net/viewvc/osdb/osdb/README?revision=1.8&view=markup>. Acesso em: 30 maio 2009.

TOKUTEK. **Performance brief**: TokuDB for MySQL. [S.l.], 2009. Disponível em: <http://www.tokutek.com/mysql-performance-brief.pdf>. Acesso em: 30 maio 2009.

WUESTEFELD, Klaus. **Prevayler**: the open source prevalence layer. [S.l]. 2009. Disponível em: <www.prevayler.org>. Acesso: 17 jun. 2009.

APÊNDICE A – Diagrama de classe completo

O diagrama de classes completo (com atributos e métodos) do Tipo de Tabela é destacado na Figura 22.

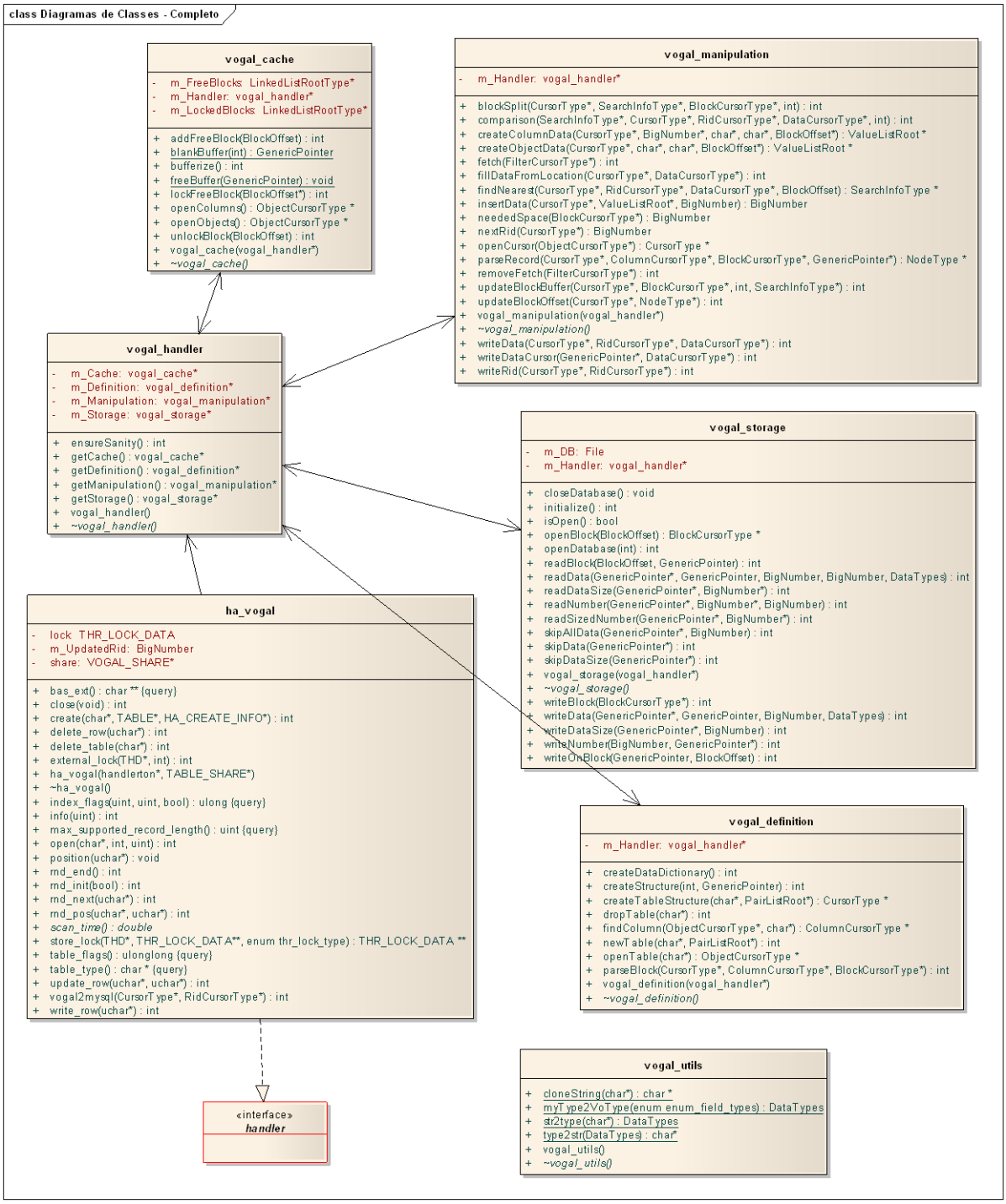


Figura 22 - Diagrama de classes completo do Tipo de Tabela