OPS: Um subsistema extensível e configurável de armazenamento de objetos persistentes

Maurício Riguette Mediano
mediano@inf.puc-rio.br

PUC-RioInf.MCC37/97 November, 1997

Abstract: The persistent object storage subsystem OPS proposes an extensible and configurable model through which it is possible to model, implement and combine different database technologies to provide new capabilities or to obtain performance improvements. OPS is based on four abstractions: persistent object, object data area, object manager and media.
Keywords: object-oriented databases, persistence.

Resumo: O subsistema de armazenamento de objetos persistentes OPS propõe um modelo extensível e configurável através do qual torna-se possível modelar, implementar e combinar diversas tecnologias de bancos de dados para prover novas funcionalidades ou obter ganho de desempenho. O OPS é baseado em quatro abstrações: o objeto persistente, a área de dados do objeto, o gerenciador de objetos e e o meio.
Palavras-chave: bancos de dados orientados a objetos, persistência.

1 Introdução

Grande parte das tecnologias disponíveis para otimizar o acesso a bancos de dados são baseadas em estruturas de dados persistentes. Árvores-B [Com79] e árvores-B posicionais [CDKK85, CDRS86] são utilizadas para indexar dados unidimensionais e armazenar objetos longos. Árvores-R [Gut84], árvores-R^+ [SRF87], árvores-R^* [BKSS90] e arquivos *Grid* [NHS84] são usados como filtros para indexar objetos de duas ou mais dimensões. Árvores-V [MCD94b] e árvores-TR [SK91] armazenam a geometria de objetos em duas dimensões. Árvores de sintaxe abstrata armazenam a estrutura sintática de programas e modelos de contextos aninhados [SRC95] armazenam documentos hipermídia com múltiplas representações. Finalmente, a árvore HPS [MGC96] armazena um mapa vetorial com multi-resolução geométrica e topológica. Quando persistentes, tais estruturas nada mais são que objetos complexos persistentes.

Uma modelagem que possibilite implementar persistência de forma eficiente em tais objetos complexos deve permitir: (1) a composição de objetos a partir de tipos básicos (tais como numerais e *strings*); (2) a composição de objetos a partir de objetos complexos; (3) a composição de objetos a partir de referências para outros objetos; (4) o isolamento dos métodos que manipulam os objetos persistentes de detalhes da implementação do meio de acesso, bem como da forma como será feito o gerenciamento da alocação da área de dados do objeto no meio de acesso; (5) o uso de novas tecnologias e a conseqüente obtenção de ganhos de desempenho, sem causar impacto para aplicações e classes de objetos persistentes disponibilizadas antes do advento de tais tecnologias; (6) a composição de tecnologias disponíveis para obter ganhos de desempenho.

A seção 2 descreve o subsistema OPS. A biblioteca básica com classes de persistência pré-definidas é apresentada na seção 3. A discussão sobre arquiteturas em camadas é feita na seção 4. Na seção 5 são feitos testes de desempenho sobre várias arquiteturas diferentes. Finalmente, nas seções 6 e 7 encontram-se as conclusões e os agradecimentos, respectivamente.

2 O subsistema OPS

O termo *objeto persistente* será utilizado para descrever objetos que têm a propriedade de continuar a existir (persistir) entre as execuções das aplicações que fazem uso dos mesmos. O termo *meio de acesso*, ou simplesmente *meio*, será usado durante o texto com dois sentidos: para fazer referência ao meio persistente do qual são recuperados objetos (memória principal, disco, fita etc), ou então para fazer referência a abstrações criadas para esconder ou implementar novas funcionalidades ou para obter ganhos de desempenho, sobre meios de acesso (*buffer*, ata, rede, etc). O termo *gerenciador* será usado para fazer referência à metodologia utilizada para gerenciar a alocação e liberação da área utilizada por objetos persistentes no meio persistente. O termo *memória persistente do objeto*, ou *memória persistente*, faz menção ao conteúdo do objeto armazenado no meio persistente. O termo *arquitetura em camadas*, ou simplesmente *arquitetura*, será usado para descrever uma seqüência de gerenciadores e meios de acesso entre a aplicação (ou bibliotecas que manipulam objetos persistentes) e o meio persistente.

O OPS foi modelado utilizando uma abordagem orientada a objetos. Esta abordagem permite: (1) esconder detalhes da implementação das classes que oferecem serviços de persistência; (2) configurar, em tempo de execução, como será feito o acesso ao meio persistente; (3) configurar, em tempo de execução, que gerenciador será utilizado; (4) em outras palavras, configurar, em tempo de execução, uma arquitetura que melhor se adeque à aplicação, à classe de objetos persistentes e ao meio persistente.

2.1 As classes básicas

O subsistema OPS foi modelado a partir de quatro abstrações: o objeto persistente, o gerenciador, o meio e a memória persistente.

As três primeiras são modeladas através de classes virtuais puras, que são chamadas: *PersistenteVirtual*, *GerenciadorVirtual* e *MeioVirtual*, respectivamente. Para criar uma classe de objetos persistentes, um novo gerenciador ou um meio, é necessário que as classes candidatas satisfaçam ao comportamento e às restrições impostas pelas respectivas classes virtuais, conforme será exposto adiante. A quarta abstração, a memória persistente, chamada *MemoriaPersistente*, é usada para armazenar a parte do objeto que se deseja manter persistente. A classe `MemoriaPersistente` é utilizada como interface entre o objeto persistente e o meio. Seu funcionamento será explicado em maiores detalhes a seguir.

2.1.1 A classe MemoriaPersistente

A classe `MemoriaPersistente` é usada para armazenar a parte persistente de um dado objeto `O` em um meio peristente `M`. Ela possui três métodos:

- `T Tamanho()` - retorna o tamanho `T` da memória persistente do objeto `O`.
- `Le(A)` - lê o atributo `A` do objeto `O`;
- `Escreve(A)` - escreve o atributo A do objeto `O`;

A memória persistente é manipulada entre as camadas da arquitetura até ser armazenada num meio persistente.

2.1.2 A classe PersistenteVirtual

Para tornar persistente uma determinada classe de objetos `C`, é necessário que `C` herde da classe `PersistenteVirtual` e defina três métodos:

- `T Tamanho()` - retorna o tamanho máximo `T` que o objeto pode a ocupar no meio persistente;
- `Le(B)` - lê a memória persistente `B` do objeto;
- `Escreve(B)` - escreve a memória persistente `B` do objeto.

Ao herdar da classe `PersistenteVirtual` os objetos da classe `C` passam a possuir dois novos métodos:

- `Oid Id()` - retorna o identificador `Oid` do objeto;
- `Id(Oid)` - atribui um novo identificador `Oid` ao objeto.

A implementação dos métodos `Tamanho`, `Le` e `Escreve` é uma tarefa bastante simples:

- se os atributos são numerais, *strings* ou referências (*Oids*): (1) quanto ao método `Tamanho`, os métodos para saber o tamanho de numerais, *strings* e referências estão disponíveis na linguagem; (2) os métodos `Le` e `Escreve` usam os métodos `Le` e `Escreve` para numerais, *strings* e referências disponíveis na classe à qual o parâmetro `B` pertence (MemóriaPersistente).
- caso contrário, se os atributos são objetos persistentes (embora isso não seja obrigatório), basta usar os métodos Tamanho, Le e Escreve definidos nas classes dos atributos.

Existem algumas boas razões para exigir que seja definido um tamanho máximo para um objeto persistente: (1) muitas linguagens de programação adotam esta estratégia ainda na memória principal, pois gerenciar o aumento do objeto em tempo de execução gera uma queda no desempenho das aplicações; (2) esse problema é bem mais sério em meios persistentes; (3) na grande maioria dos casos é possível e extremamente aconselhável implementar objetos persistentes de tamanho variável a partir de objetos de tamanho fixo.

2.1.3 A classe MeioVirtual

O meio pode ser visto como a abstração de um vetor de *bytes*, onde pode-se acessar trechos deste vetor a partir de uma posição e da quantidade de *bytes* a serem acessados. Para definir um novo meio é necessário que este defina quatro métodos:

- `Le(Oid,B)` - lê `B.Tamanho()` *bytes* do meio `M`, a partir da posição `Oid`, necessários para preencher `B`;
- `Escreve(Oid,B)` - escreve os `B.Tamanho()` *bytes* contidos em `B` a partir da posição `Oid` em `M`;
- `EscrevePrimeiro(Oid,B)` - escreve `B` pela primeira vez;
- `Mensagem(N,S,Oid)` - se `N` é o nome do meio `M`, `Mensagem` provoca a execução da mensagem `S`; caso contrário, `M` passa a mensagem `S` internamente para um meio `M2`, a partir do qual `M` implementa sua abstração, executando `M2.Mensagem(N,S,Oid)`. Se `M` usa outro meio `M2`, diremos que `M` é *derivado por meio*; caso contrário, diremos que `M` é *primitivo*. Um possível retorno para uma mensagem pode ser feito através do retorno do parâmetro `Oid`, que retorna uma referência para uma posição em `M` onde se encontra a resposta da mensagem.

Ao herdar da classe `MeioVirtual` os objetos da classe `C` passam a possuir quatro novos métodos:

- `Le(O)` - lê um objeto `O`;

- `Escreve(O)` - escreve um objeto `O`;
- `EscrevePrimeiro(O)` - escreve `O` pela primeira vez;
- `Meio(M2)` - deriva `M` através de `M2`.

O fato do `Oid` representar diretamente a posição em que o objeto está no meio persistente, deve-se ao fato do meio representar a abstração de um vetor. Na classe `GerenciadorVirtual`, o `Oid` eventualmente possui um significado menos restrito, como será visto adiante. A noção de que um trecho do vetor está sendo escrito pela primeira vez é muitas vezes importante para a implementação de alguns tipos de meio. O método `Meio` nos meios `M`, `M2`, `M3` e `M4` pode ser usado para definir, em tempo de execução, uma seqüência de meios que `M` acessa:

```
M.Meio(M2)
M2.Meio(M3)
M3.Meio(M4)
```

2.1.4 A classe GerenciadorVirtual

O gerenciador é o responsável por alocar o espaço de memória necessário para armazenar um determinado objeto no meio persistente. Para definir um novo gerenciador é necessário que o mesmo herde da classe GerenciadorVirtual e defina cinco métodos:

- `Le(Oid,B)` - lê a memória persistente `B` identificada por `Oid`;
- `Escreve(Oid,B)` - escreve a memória persistente `B` identificada por `Oid`;
- `Oid Cria(B)` - aloca espaço no meio persistente referenciado por `Oid` para armazenar a memória persistente `B`;
- `Destroi(Oid,B)` - libera o espaço de `B` no meio persistente referenciado por `Oid`;
- `Mensagem(N,S,Oid)` - se `N` é o nome do gerenciador `G`, o método `Mensagem` provoca a execução da mensagem `S`; caso contrário, a mensagem é passada para um meio `M` a partir do qual `G` aloca memória, ou para um gerenciador `G2` utilizado por `G`. Se `G` usa um meio `M` diremos que `G` é *derivado por meio*; caso contrário, diremos que `G` é *derivado por gerenciador*. O retorno da resposta, se necessário, é feito através do identificador `Oid`, que é uma referência para um objeto que contém a resposta.

Ao herdar da classe `GerenciadorVirtual` os objetos da classe `C` passam a possuir seis novos métodos:

- `Le(O)` - lê um objeto `O`;
- `Escreve(O)` - escreve um objeto `O`;
- `Cria(O)` - cria espaço em meio persistente para o objeto `O`;
- `Destroi(O)` - libera o espaço do objeto `O` em meio persistente;

- `Meio(M)` - o gerenciador passa a administrar a alocação de espaço no meio `M`;
- `Gerenciador(G2)` - o gerenciador passa a alocar memória usando o gerenciador `G2`.

O `Oid` retornado pelo método `Cria` não precisa referenciar a posição onde será armazenada a memória persistente do objeto, pois podem ser criadas indireções. É suficiente que, a partir do `Oid`, os métodos `Le` e `Escreve` do gerenciador possam ler e escrever a memória persistente do objeto, respectivamente. O método `Mensagem` deve ser usado para mandar comandos específicos para o gerenciador `G` ou para o meio `M` derivado ou para o gerenciador `G2` derivado. Os métodos `Meio` e `Gerenciador` podem ser usados para definir, em tempo de execução, qual sequência (arquitetura) de gerenciadores e meios `G` acessa (`G-G2-G3-M-M2-M3`), como no exemplo a seguir.

```
G.Gerenciador(G2)
G2.Gerenciador(G3)
G3.Meio(M)
M.Meio(M2)
M2.Meio(M3)
```

3 A biblioteca básica

A biblioteca básica que acompanha o `OPS` não tem o objetivo de oferecer o estado-da-arte em técnicas de acesso a bancos de dados, mas de oferecer um conjunto inicial de meios e gerenciadores a partir do qual é possível acessar novos meios, construir novas abstrações para implementar novas funcionalidades ou obter ganhos de desempenho no acesso a uma determinada classe de objetos persistentes.

3.1 Meios

O OPS possui cinco meios pré-definidos. São eles:

- `MeioArquivo` - este meio constrói uma abstração em torno do arquivo do sistema operacional, logo é um meio primitivo. Mensagens disponíveis:
  - `"cria nomearq"` - cria um arquivo de nome `nomearq`;
  - `"abre nomearq"` - abre o arquivo de nome `nomearq`;
  - `"fecha"` - fecha o arquivo.
- `MeioAta` - este meio de anota e desfaz alterações feitas no meio derivado. Mensagens disponiveis:
  - `"cria nomearq"` - cria a ata de nome `nomearq`;
  - `"abre nomearq"` - abre a ata de nome `nomearq`;
  - `"fecha"` - fecha a ata;
  - `"marca"` - escreve uma marca na ata;

- "limpa" - esvazia a ata;
- "desfaz" - desfaz todas as alterações executadas no meio derivado até a última marca feita.

- `MeioBuffer` - retém uma quantidade pré-determinada de memórias persistentes por ocasião da execução de seus métodos `Le` e `Escreve`, ao invés de repassar imediatamente para o meio derivado. Mensagens disponíveis:
  - "numero num" - passa a reter um número `num` de memórias persistentes;
  - "atualiza" - executa o método `Escreve` do meio derivado para todas as memórias persistentes retidas pelo método `Escreve`;
  - "limpa" - destrói todas as memórias persistentes retidas.

- `MeioBufferBloco` - por ocasião da execução de seus métodos `Le` e `Escreve`, retém uma quantidade pré-determinada de blocos do meio derivado de mesmo tamanho ao invés de repassar os blocos imediatamente. As memórias persistentes são inseridas e extraídas dos blocos. Mensagens disponíveis:
  - "numblocos num" - passa a reter um número `num` de blocos do meio;
  - "tamanhobloco num" - passa a usar blocos de `num` *bytes*;
  - "atualiza" - executa o método `Escreve` do meio derivado para todos os blocos retidos pelo seu método `Escreve`;
  - "limpa" - destrói todos os blocos retidos.

- `MeioRede` - faz a comunicação com um meio remoto através da rede. É um derivado por meio. Mensagens disponíveis:
  - "abre C" - abre uma conexão dada a string de conexão `C`;
  - "fecha" - fecha a conexão;
  - "usemeio N" - passa a usar o meio de nome `N`. Se `N` for derivado por meio, o fará a partir do último meio usado na última mensagem "usemeio". Chamadas sucessivas de "usemeio" definem uma arquitetura de servidor de páginas [DMFV90].

São cinco os gerenciadores pré-definidos no OPS:

3.2 Gerenciadores

- `Gerenciador1` - aloca, em um meio derivado, memórias persistentes de um único tamanho a ser definido. Mensagens disponíveis:
  - "tamanho num" - define como `num` o número máximo de *bytes* que será alocado para cada objeto;
  - "cria" - cria o cabeçalho do gerenciador;
  - "abre" - lê o cabeçalho;

  - `"fecha"` - escreve o cabeçalho.

- `GerenciadorN` - aloca, em um meio derivado, memórias persistentes de um número máximo de tamanhos diferentes a ser definido. Mensagens disponíveis:
  - `"tamanhomaximo tam"` - define como `tam` o tamanho da maior memória persistente a ser alocada;
  - `"blocosdiferentes num"` - define como `num` o número de blocos de tamanhos diferentes a ser alocado;
  - `"cria"` - cria o cabeçalho do gerenciador;
  - `"abre"` - lê o cabeçalho;
  - `"fecha"` - escreve o cabeçalho.

- `GerenciadorGrupo` - tenta agrupar memórias persistentes alocadas em repositórios no meio derivado. Mensagens disponíveis:
  - `"criarep N"` - Cria um repositório de nome `N`;
  - `"usarep N"` - Passa a usar o repositório de nome `N`;
  - `"cria"` - cria o cabeçalho do gerenciador;
  - `"abre"` - lê o cabeçalho;
  - `"fecha"` - escreve o cabeçalho.

- `GerenciadorVer` - este gerenciador é derivado por outro gerenciador, `G2`, para que as estruturas internas que gerenciam o controle de versões e as próprias versões de cada objeto possam ser alocadas. Permite criar versões múltiplas de objetos alocados pelo gerenciador derivado. Mensagens disponíveis:
  - `"criaver N"` - Cria uma versão de nome `N` a partir da versão corrente;
  - `"usaver N"` - torna a versão de nome `N` corrente;
  - `"cria"` - cria o cabeçalho do gerenciador;
  - `"abre"` - lê o cabeçalho;
  - `"fecha"` - escreve o cabeçalho.

- `GerenciadorRede` - faz a comunicação com um gerenciador derivado remoto através da rede. Mensagens disponíveis:
  - `"abre C"` - abre uma conexão dada a string de conexão `C`;
  - `"fecha"` - fecha a conexão;
  - `"usemeio N"` - passa a usar o meio de nome `N`, se `N` for derivado, o fará a partir do último meio usado na última mensagem `"usemeio"`. Chamadas sucessivas definem a parte dos meios de uma arquitetura de servidor de objetos [DMFV90];

– "`usegerenciador N`" - passa a usar o gerenciador de nome `N`; se `N` for derivado por meio, o fará a partir do último meio usado na última mensagem "`usemeio`", senão `N` será derivado pelo último gerenciador usado na última mensagem "`usegerenciador`". Chamadas sucessivas definem a parte dos gerenciadores de uma arquitetura de servidor de objetos.

4 Arquiteturas

Ao combinar as várias camadas de meios e gerenciadores derivados, torna-se possível definir uma arquitetura em camadas, através da qual uma dada aplicação ou biblioteca de objetos persistentes irá acessar objetos em um meio primitivo. Uma infinidade de arquiteturas pode ser gerada com base nos meios e gerenciadores contidos somente na biblioteca básica discutida anteriormente.

A arquitetura genérica proposta abaixo restringe as possibilidades disponíveis a um conjunto de arquiteturas comumente utilizadas:

```
(a)                           (1) GerenciadorVer
(b) (1) Gerenciador1 (2) GerenciadorN (3) GerenciadorGrupo (4) GerenciadorRede
(c)                              (1) MeioAta
(d)                    (1) MeioBuffer (2) MeioBufferBloco
(e)                              (1) MeioAta
(f)                     (1) MeioRede (2) MeioArquivo
```

O gerenciador de versões, a camada (a), é opcional. Na camada (b) um único gerenciador dentre (1), (2), (3) e (4) tem que ser escolhido. Se o gerenciador de rede (4) é escolhido, nenhuma das camadas (c) (d), (e) e (f) é usada localmente na aplicação. As camadas (c) e (e) são opcionais, e apenas uma delas deve ser usada. A camada (d) é opcional, apenas um meio dentre (1) e (2) deve ser usado. Na camada (b) um único meio dentre (1) e (2) deve ser escolhido.

Quando o meio de rede ou o gerenciador de rede são escolhidos, pode-se ver a arquitetura como a sobreposição das camadas da arquitetura local e as camadas da arquitetura remota. Se, a partir de um gerenciador, define-se um novo gerenciador no qual são acrescentadas novas mensagens que conhecem a semântica de uma determinada classe de objetos, pode-se implementar um servidor de serviços desta classe de objetos. Por exemplo, se um servidor conhece a semântica de catálogos, relações e tuplas, ele estará apto a responder mensagens direcionadas para estes objetos, tornando possível implententar um servidor SQL usando esta arquitetura. Um outro exemplo seria um servidor WWW.

5 Testes

Para testar a flexibilidade e os ganhos de desempenho que podem ser obtidos usando o OPS, dois conjutos de testes foram feitos usando diferentes arquiteturas baseadas na arquitetura genérica apresentada. Cada teste foi feito através da interpretação de um arquivo-texto que possui três tipos de comandos: configurar a arquitetura, enviar mensagens (para os gerenciadores e os meios) e executar os testes. Foram usadas 16.000 memórias persistentes no meio *buffer* e e 1000 blocos de 4 *kbytes* no meio buffer de bloco.

O primeiro conjunto de testes foi baseado em um subconjunto de um sistema relacional que implementa catálogos, relações, árvores-B e tuplas. O segundo conjunto de testes foi feito sobre duas estruturas, a árvore-R e a árvore-V, que, combinadas, armazenam conjuntos de linhas poligonais de mapas de bancos de dados geográficos. A árvore-V [MCD94a] permite usar o conceito de multiresolução geométrica para gerar representações de linhas poligonais com menos detalhe (em escalas maiores), o que possibilita ganho de desempenho, como será visto adiante.

Os testes foram feitos em uma máquina `PC-Pentium 133 Mhz` com o sistema operacional `Linux`.

5.1 Testes usando o modelo relacional

A primeira parte do teste feito sobre o modelo relacional consiste em criar um catálogo e uma relação `R` com duas colunas: uma cadeia `fc` de dez caracteres e um número inteiro `fi`. A seguir é criada e mantida uma ordenação sobre `fi`. São inseridas cinqüenta mil tuplas na relação `R`; em seguida, são feitas cinco mil consultas, que recuperam as dez primeiras tuplas retornadas que satifazem a condição $fi > X$ (com um valor diferente de `X` para cada consulta).

A segunda parte do teste consiste em usar o gerenciador de versões para criar diferentes versões do catálogo e da relação criada na primeira parte do teste com apenas mil tuplas. A primeira versão é criada com as mesmas mil tuplas. Em seguida, são criadas sete versões, onde dez por cento das linhas da relação `R` são atualizados em cada versão. Mais uma vez são criadas sete versões para cada uma das sete versões criadas anteriormente, e novamente dez por cento das linhas da relação `R` são atualizados a cada versão. São feitas mil consultas. No total são geradas cinqüenta e sete versões $(1 + 7 + 49)$.

A terceira parte do teste consiste em refazer a segunda parte, atualizando cinqüenta por cento das tuplas da relação `R` em cada versão, ao invés de atualizar apenas dez por cento.

A primeira conclusão a que pode-se chegar ao ver os resultados da Tabela 1 é que se a arquitetura não é dinâmica, ou seja, ela não permite ser configurada, há uma grande perda de desempenho pelo uso recursos que não são necessários. O uso do gerenciador de versões gera um arquivo três vezes maior, oito vezes o número de acessos de leitura e 50% mais acessos de escrita. O uso de ata piora o desempenho do sistema em todos os casos. A alternativa de usar a ata abaixo da camada de buffer diminui a perda ocasionada pelo uso da ata. Para usar esta estratégia, é necessário escrever num meio persistente as memórias persistentes que estão na camada de buffer (por ocasião de uma chamada do método de escrita) antes de esvaziar a ata (para levar o banco de dados a um estado consistente). Em vários casos não houve acessos de leitura, o que significa que cada objeto permaneceu no buffer enquanto era acessado pelo teste. A maior diferença entre os desempenhos foi de seis vezes (NBF e VNLF), o que mostra a importância da camada de buffer, principalmente de bloco. O bom desempenho da camada de buffer manteve-se durante os testes de consulta (veja a Tabela 2). Nos resultados das consultas praticamente não houve variação quando a ata foi usada, como era esperado, pois neste caso a ata simplesmente repassa o acesso de leitura para a camada inferior.

Pode-se ver nas Tabelas 3 e 5 que o gerenciador de versões é eficiente ao armazenar múltiplas versões com poucas diferenças entre si (10% de atualizações). O tamanho do arquivo é de três a quatro vezes maior quando o número de atualizações sobe de 10% para 50%, embora ambos possuam o mesmo número de tuplas. O aumento no número de atualizações praticamente não influenciou no resultado das consultas (veja as Tabelas 4 e 6).

5.2 Testes com mapas

O teste com dados vetoriais consistiu em armazenar um conjunto de linhas poligonais de um mapa de vegetação da amazônia de 18 *megabytes* e gerar uma versão deste mapa com representações menos refinadas das linhas originais, como proposto em [MCG96, MGC96]. A tolerância usada considerará como 5 kilômetos a distância máxima que a nova versão de uma linha do mapa poderá se deslocar de sua posição original. Devido à dimensão do mapa (a Amazônia toda), esta variação (tolerância de 5km) é praticamente imperceptível quando a nova versão do mapa é usada para gerar imagens *raster* de 512x512.

Na Tabela 7 pode-se notar a variação no desempenho do algoritmo de criação das estruturas que armazenam o mapa em função da arquitetura utilizada. A arquitetura que obteve o melhor desempenho a que utilizou o buffer de bloco, pois o número de acessos de leitura foi próximo de zero e o número de acessos de escrita diminuiu cem vezes. Foram feitos 19.372 acessos de escrita de páginas de 4096 *bytes*, totalizando aproximadamente 80 *megabytes*, o dobro do tamanho do arquivo, o que significa que na média cada página foi escrita apenas duas vezes. Quando o efeito da camada de buffer não é eficiente o bastante para compensar o aumento da complexidade do sistema ao usá-la, a camada de buffer acaba por piorar o desempenho do sistema (NLF e NLbF). A maior diferença entre os tempos de processamento gastos para a criação do mapa é de quatro vezes (NBF e NLbF).

Em relação aos resultados da consulta ao mapa (veja a Tabela 8), a melhor arquitetura é a que acessa diretamente o meio persistente (NF), pois cada objeto é tocado uma única vez para gerar o mapa. O ganho obtido no tempo de processamento (Tabela 9) pelo algoritmo que faz uso de multi-resolução é de aproximadamente 20% em todos os casos. O uso de técnicas de agrupamento possivelmente traria ganhos maiores, pois a camada de buffer de bloco fez 9.747 acessos de leitura ao meio persistente, o que equivale a ler praticamente todo o arquivo, apesar das técnicas de multi-resolucão terem diminuído 15 vezes o número objetos lidos (de 609.498 para 44.221).

6 Conclusão

Os resultados mostram que a arquitetura usada para acessar bancos de dados deve ser dinâmica para que se possam obter ganhos de desempenho. Não existe uma arquitetura que seja boa em todos os casos.

O subsistema OPS permite configurar e estender diferentes arquiteturas de acesso a objetos persistentes, em tempo de execução e sem causar impacto às aplicações. Através da flexibilidade e extensibilidade do OPS é possível aplicar novas tecnologias de bancos de dados a aplicações existentes, combinando-as em novas arquiteturas para obter ganhos de desempenho.

Arq	Tempo	Tam	ALB	AEB	ALL	AEL	ALA	AEA
N F	3:10.48	2831					453103	636423
N LF	6:23.02	2831			453103	636423	1028417	636423
N B F	1:41.73	2834	453103	636423			0	1384
N BLF	1:45.60	2834	453103	636423	0	1384	692	1384
N b F	2:06.71	2831	453103	636423			0	122218
N bLF	2:26.34	2831	453103	636423	0	122218	61109	122218
NLB F	4:44.87	2834	1028417	636423	453103	636423	0	1384
NLb F	5:32.93	2831	1028417	636423	453103	636423	0	122218
VN F	7:39.33	8943					3477236	1064186
VN LF	13:04.98	8943			3477236	1064186	4358095	1064186
VN B F	7:49.27	8945	3477236	1064186			0	4368
VN BLF	6:22.96	8945	3477236	1064186	0	4368	2184	4368
VN b F	6:14.36	8943	3477236	1064186			0	366654
VN bLF	6:59.34	8943	3477236	1064186	0	366654	183327	366654
VNLB F	9:37.44	8945	4358095	1064186	3477236	1064186	0	4368
VNLb F	12:08.56	8943	4358095	1064186	3477236	1064186	0	366654

Arq - Arquitetura usada.
V - GerenciadorVer.
N - GerenciadorN.
B - MeioBufferBloco.
b - MeioBuffer.
L - MeioAta.
F - MeioArquivo.
Tempo - Tempo total de processamento.
Tam - Tamanho em k̂bytes do arquivo gerado por F.
ALB e AEB - acessos de leitura e escrita em buffer (B ou b).
ALL e AEL - acessos de leitura e escrita em ata (L).
ALA e AEA - acessos de leitura e escrita em arquivo (F).

Tabela 1: Inserção de 50.000 tuplas em R.

Arq	Tempo	ALB	ALL	ALA
N F	0:08.95			80551
N B F	0:07.41	80551		692
N b F	0:10.83	80551		55556
VN F	0:24.21			241660
VN B F	0:44.49	241660		2184
VN b F	0:50.51	241660		166668

Tabela 2: Resultados de 5.000 consultas sobre R.

Arq	Tempo	Tam	ALB	AEB	ALL	AEL	ALA	AEA
`VN   F`	3:52.55	1876					1778934	123923
`VN  LF`	4:27.75	1876			1778934	123923	1877926	123923
`VN B F`	1:26.13	1880	1778934	123923			0	918
`VN BLF`	1:28.25	1880	1778934	123923	0	918	459	918
`VN b F`	1:37.39	1876	1778934	123923			0	49862
`VN bLF`	1:42.43	1876	1778934	123923	0	49862	24931	49862
`VNLB F`	2:03.60	1880	1877926	123923	1778934	123923	0	918
`VNLb F`	2:24.44	1876	1877926	123923	1778934	123923	0	49862

Tabela 3: Criação de múltiplas versões de R com 10% de atualizações.

Arq	Tempo	ALB	ALL	ALA
`VN   F`	7:46.44			3951816
`VN B F`	4:21.55	3951816		106
`VN b F`	3:58.25	3951816		340

Tabela 4: Resultados de 1000 consultas a cada versão de R com 10% de atualizações.

Arq	Tempo	Tam	ALB	AEB	ALL	AEL	ALA	AEA
`VN   F`	18:31.59	6184					8630330	540203
`VN  LF`	22:32.30	6184			8630330	540203	9062094	540203
`VN B F`	6:58.54	6184	8630330	540203			0	3020
`VN BLF`	7:41.50	6184	8630330	540203	0	3020	1510	3020
`VN b F`	8:26.87	6184	8630330	540203			284	216888
`VN bLF`	9:06.46	6184	8630330	540203	284	216888	108733	216888
`VNLB F`	9:47.46	6184	9062094	540203	8630330	540203	0	3020
`VNLb F`	11:23.07	6184	9062094	540203	8630330	540203	284	216888

Tabela 5: Criação de múltiplas versões de R com 50% de atualizações.

Arq	Tempo	ALB	ALL	ALA
`VN   F`	7:29.91			3951816
`VN B F`	4:18.12	3951816		104
`VN b F`	3:58.33	3951816		340

Tabela 6: Resultados de 1000 consultas a cada versão de R com 50% de atualizações.

Arq	Tempo	Tam	ALB	AEB	ALL	AEL	ALA	AEA
N F	5:14.42	39590					78188	1886722
N LF	13:24.79	39590			78188	1886722	1355412	1886722
N B F	3:28.46	39591	78188	1886722			40	19372
N BLF	7:26.51	39591	78188	1886722	40	19372	9746	19372
N b F	7:47.44	39590	78188	1886722			83	1219079
N bLF	12:17.67	39590	78188	1886722	83	1219079	609664	1219079
NLB F	10:40.26	39591	1355412	1886722	78188	1886722	40	19372
NLb F	14:47.10	39590	1355412	1886722	78188	1886722	83	1219079

Tabela 7: Inserção das linhas de vegetação da amazônia.

Arq	Tempo	ALB	ALL	ALA
N F	1:28.48			609498
N B F	1:42.18	609498		11485
N b F	2:27.60	609498		609498

Tabela 8: Consulta ao mapa sem tolerância.

Arq	Tempo	ALB	ALL	ALA
N F	1:09.91			44221
N B F	1:13.02	44221		9747
N b F	1:15.45	44221		44221

Tabela 9: Consulta ao mapa com tolerância de 5 kilômetros.

7 Agradecimentos

Ao meu orientador Marco Antônio Casanova pelas discussões que muito colaboraram para a elaboração deste trabalho. À equipe do Laboratório de Engenharia de Software da PUC-Rio, que participou da análise dos requisitos necessários para o uso do OPS na implementação de árvores de sintaxe abstrata persistentes. À equipe do Laboratório de Multimídia da PUC-Rio que participou da análise dos requisitos necessários para o uso do OPS na implementação de persistência em modelos de contextos aninhados. Este trabalho foi parcialmente sustentado pelos projetos: PROTEM/GEOTEC, financiado pelo CNPQ; e TeCGraf/PUC-Rio, financiado pelo SEPROC/CENPES/PETROBRAS. O autor gostaria de agradecer também a CAPES pelo financiamento de sua bolsa de estudos.

Referências

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Retangles. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 322–332, May 1990.

[CDKK85] H-T. Chou, David J. DeWitt, Randy H. Katz, and Anthony C. Klung. Design and Implementation of the Wisconsin Storage System. *Software-Pratice and Experience*, 15(10):943–962, October 1985.

[CDRS86] M. J. Carey, David J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th VLDB Conference*, pages 91–100, August 1986.

[Com79] D. Comer. The Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–131, June 1979.

[DMFV90] David J. DeWitt, David Maier, Philippe Futtersack, and Fernando Velez. A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems. In *Proceedings of the 16th VLDB Conference*, 1990.

[Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD Conference on Data Engineering*, pages 47–56, June 1984.

[MCD94a] Maurício Riguette Mediano, Marco Antônio Casanova, and Marcelo Dreux. A Family of Storage Methods for Geographic Data. In *9º Simpósio Brasileiro de Bancos de Dados*, Setembro 1994.

[MCD94b] Maurício Riguette Mediano, Marco Antônio Casanova, and Marcelo Dreux. V-tree - A Storage Method for Long Vector Data. In *Proceedings of the 20th VLDB Conference*, September 1994.

[MCG96] Maurício Riguette Mediano, Marco Antônio Casanova, and Marcelo Gattass. Map-tree: Um método de acesso para mapas longos. In *11º Simpósio Brasileiro de Bancos de Dados*, Setembro 1996.

[MGC96] Maurício Riguette Mediano, Marcelo Gattass, and Marco Antônio Casanova. HPS-tree: Um método de acesso para armazenar mapas longos com multiresolução geométrica e topológica. In *IX Simpósio Brasileiro de Computação Gráfica e Processamento de Imangens*, pages 219–226, Outubro 1996.

[NHS84] J. Nievergelt, H. Hinterberguer, and K. C. Sevick. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[SK91] Ralf Schneider and Hans-Peter Kriegel. The TR*-tree: A New Representation of Polygonal Objects Supporting Spatial Queries and Operations. In *Proceedings of 7th Workshop on Computational Geometry*, pages 507–518, 1991.

[SRC95] L.F.G. Soares, N.L.R. Rodriguez, and M.A. Casanova. Nested Composite Nodes and Version Control in an Open Hypermedia System. *Information Systems*, 20(6):501–519, 1995.

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th VLDB Conference*, pages 507–518, September 1987.