UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA - CCT
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA - COPIN

Modelo e Implementação de Objetos Móveis em Banco de Dados Objeto-Relacional

Ricardo Santos de Oliveira

(Mestrando)

Cláudio de Souza Baptista

(Orientador)

Campina Grande – PB
Abril de 2003

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA - CCT
COORDENAÇÃO DE PÓS GRADUAÇÃO EM INFORMÁTICA – COPIN

Modelo e Implementação de Objetos Móveis em Banco de Dados Objeto-Relacional

Ricardo Santos de Oliveira

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande, como parte dos requisitos necessários para obtenção do grau de Mestre em Informática

Orientador: Cláudio de Souza Baptista

Campina Grande – PB
Março de 2003

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

O armazenamento de objetos que se movimentam em um banco de dados traz uma série de desafios a serem resolvidos. Como esses objetos mudam de posição continuamente, e não discretamente, manter informações sobre sua localização atualizadas na base de dados requer constante modificação nos valores lá contidos. Atualizar essa base de dados constantemente é inadequado, pois gera uma sobrecarga de trabalho no banco, bem como um possível excesso de tráfego e consumo de largura de banda. Neste trabalho propomos um modelo para aplicações de objetos móveis em banco de dados espaço-temporal, que se vale de atributos dinâmicos para amenizar o problema de constantes atualizações no banco e possibilitar a resposta a consultas que envolvam tempo passado, presente e futuro. Este modelo será validado através de uma aplicação que deposita e consulta dados em um SGBD Objeto-Relacional.

Capítulo 1. Introdução

A pesquisa em computação móvel pode ser considerada relativamente recente no estudo da informática devido ao fato de que ela depende da evolução de duas outras áreas da computação: o desenvolvimento de dispositivos portáteis e a comunicação através de redes sem fio.

Para que seja possível a mobilidade, devem existir dispositivos mais adequados do que os computadores de mesa, cujo tamanho e formato tornam-no de difícil transporte e manuseio durante o movimento. Além do desenvolvimento de computadores portáteis de mesmo poder computacional que as máquinas de mesa, outros dispositivos como telefones celulares, PDAs (*Personal Digital Assistants*, ou agendas eletrônicas), computadores de bordo de automóveis, etc. também provêm suporte para sistemas de computação móvel.

A grande maioria das aplicações móveis necessita realizar troca de informações, seja com uma unidade central seja com outros dispositivos móveis. Como um dispositivo que se movimenta não pode estar preso a uma estrutura de fios e cabos, é necessário que esta comunicação se realize através de uma rede sem fios.

Entre outras, uma gama de aplicações que podem surgir a partir da computação móvel são as aplicações de bancos de dados de objetos móveis. Com dados distribuídos em dispositivos que se movimentam, surge a possibilidade de fazer com que estes dados reflitam a localização atual do dispositivo. Baseado nessa informação é possível realizar consultas cujos resultados dependem da localização do objeto. Consultas como estas são chamadas de LBR – *Location Based Retrieval* (recuperação baseada na localidade) [DK98].

Uma aplicação de banco de dados de objetos móveis pode ter, além das informações tradicionais acerca de atributos de um objeto, as informações acerca da localidade do objeto armazenadas em sua base de dados. Estes objetos, porém, se movimentam de maneira contínua, fazendo com que as informações sobre sua localidade contidas no banco de dados devam ser relacionadas com determinado instante de tempo. Um objeto móvel é um

dispositivo que se movimenta continuamente e tem suas informações de localização armazenadas em base de dados como um atributo espacial, dependente do instante de tempo ao qual se refere [WXC+98].

O trabalho em bancos de dados de objetos móveis possibilita uma série de aplicações de interesse científico e comercial. Exemplos dessas aplicações seriam sistemas para empresas de transporte, como táxis e caminhões de carga, que poderiam localizar seus veículos a qualquer instante; ou aplicações militares, como a composição de um campo de batalha virtual, ajudando na definição de estratégias de combate.

O restante do capítulo será dividido da seguinte forma: a seção 1.1 informa a motivação de se realizar este trabalho, a seção 1.2 traz os objetivos que se tentou alcançar com o mesmo, a seção 1.3 resume as contribuições do trabalho e a seção 1.4 mostra a estrutura dos capítulos de toda a dissertação.

1.1. Motivação

O trabalho com bancos de dados de objetos móveis ainda traz inerente a si uma série de problemas para os quais a comunidade científica ainda não padronizou soluções. O principal deles é o fato de que objetos móveis movimentam-se de maneira contínua, enquanto bancos de dados atualizam suas informações de maneira discreta, como se a informação passasse de um estado para o outro e tivesse se mantido constante até a mudança. No caso de informações a respeito da localização de objetos no espaço, é como se uma posição fosse mantida constante até que houvesse uma atualização da base. Como o objeto possivelmente manteve o movimento, é grande a chance de que se venha a trabalhar com dados de localização defasados.

Para evitar essa possibilidade, seria necessário atualizar a base de dados num período muito curto de tempo. Mas esta alternativa torna-se inviável, na medida que atualizar a base de dados com muita freqüência gera uma sobrecarga de trabalho no SGBD (Sistema de Gerência de Banco de Dados), além de ocupar muita largura de banda com a transmissão constante dos dados de localização dos objetos para o servidor onde se localiza o banco. Assim, alternativas têm que ser buscadas para que não haja a necessidade de atualizar o banco constantemente e mesmo assim seja possível trabalhar com dados não defasados.

Há uma grande demanda de aplicações móveis, principalmente com o uso de telefones celulares, que ainda não estão desenvolvidas em todo o seu potencial devido a limitações tecnológicas. A padronização de um modelo de objetos móveis eficiente deve dar um grande impulso de qualidade aos serviços oferecidos aos usuários de dispositivos móveis. Esta é a grande motivação deste trabalho.

1.2. Objetivos

Conforme visto na seção anterior, há a necessidade de estudos no sentido de criar aplicações que envolvam armazenamento de dados sobre objetos móveis em SGBDs. Esta dissertação tem por objetivo geral desenvolver um modelo onde dados de objetos móveis são armazenados em uma base de dados sob o paradigma Objeto-Relacional [SBM98].

O paradigma Objeto-Relacional se mostra adequado, pois possui os recursos necessários para trabalhar com objetos que possuam requisitos espaciais e temporais, além de poder implementar as funções de atualização necessárias para descrever a movimentação dos objetos em métodos de classe. Outro beneficio deste paradigma é tornar o modelo facilmente extensível, o que é importante para aplicações com características as quais o modelo puro não consegue atender.

Espera-se também, com o uso do paradigma Objeto-Relacional, possibilitar o desenvolvimento de consultas em linguagem padrão de acesso a dados, no caso a linguagem SQL. Com isso, evita-se o desenvolvimento de uma nova linguagem específica para acesso a dados de objetos móveis e aumenta-se a chance de padronização do modelo.

1.3. Contribuições

A principal contribuição deste trabalho é um modelo genérico para aplicações de objetos móveis sem extensão, ou seja, pontos móveis, onde a mobilidade dos objetos é gerada acrescentando-se o domínio temporal a objetos espaciais. Assim, um objeto móvel nada mais é que um dado que registra diferentes valores no decorrer do tempo.

Este modelo pode ser estendido para abranger outras categorias de aplicações, como por exemplo uma aplicação em que aspectos de mobilidade sejam necessários também em

regiões, como um sistema que mostra regiões atingidas por um incêndio. Para acrescentar novas características ao modelo, utiliza-se herança de classes e interfaces definidas, constituindo-se numa tarefa simples.

Outra contribuição deste trabalho é a aplicação desenvolvida, que valida o modelo mostrando sua real utilidade. Inclui-se nesta aplicação o desenvolvimento de um simulador, que gera movimentações semelhantes às de objetos reais, e as informa à base de dados, dispensando a utilização de objetos reais dotados de dispositivos GPS (*Global Positioning System* – Sistema de Posicionamento Global) e que se comunicariam com o banco, o que necessitaria de grande disponibilidade de recursos.

1.4. Estrutura da Dissertação

A dissertação está organizada da seguinte forma:

- No capítulo 2 é realizada uma revisão bibliográfica acerca dos estudos realizados em computação móvel e bancos de dados, seguido do estudo de sistemas de objetos móveis em si. É definida uma taxonomia para sistemas de objetos móveis. Por fim, são citados trabalhos já desenvolvidos acerca do tema objetos móveis e um comparativo é realizado entre eles.

- No capítulo 3 é mostrado o modelo desenvolvido, o MOMENT (Modelo para Objetos em Movimento), com seu diagrama de classes e descrições acerca da forma com que foram resolvidos os problemas descritos no capítulo 2. É feito também um comparativo entre o modelo desenvolvido e outros modelos propostos em trabalhos relacionados.

- No capítulo 4 mostra-se como foi implementada uma aplicação baseada no MOMENT, para validar o modelo como sendo realmente implementável. Definem-se os requisitos funcionais e não funcionais para a aplicação, bem como para o simulador desenvolvido para gerar posições de objetos que se movimentam. É mostrado também um estudo de caso, onde consultas típicas em sistemas de objetos móveis são submetidas e têm seus cálculos demonstrados para validar as respostas do sistema.

- No capítulo 5 são apresentadas as conclusões acerca do trabalho, ressaltando os objetivos alcançados e o que falta ser construído, deixando sugestões para trabalhos futuros.

Capítulo 2. Computação Móvel e Bancos de Dados

2.1. Introdução

A computação móvel converge de duas tecnologias: o surgimento de computadores portáteis e o desenvolvimento de redes de comunicação sem fio cada vez mais rápidas e confiáveis. Com isso, um grande número de aplicações foi idealizado, aplicações nas quais os usuários se deslocam e continuam a realizar suas atividades.

Há dois aspectos principais nos quais a computação móvel distingue-se da computação fixa tradicional. O primeiro é a mobilidade dos usuários e o segundo são as limitações dos recursos móveis. A mobilidade dos usuários implica que estes podem se conectar à rede de comunicação a partir de vários pontos diferentes, através de conexões sem fio ou não [Tan96], o que significa que eles podem querer continuar a solicitar dados enquanto se movimentam, independentemente da possibilidade de desconexões [JHE99].

As limitações dos recursos móveis referem-se a restrições como largura de banda reduzida em comunicação sem fio, problemas de alimentação devido ao reduzido tempo de duração das baterias; e ainda a limitações técnicas do dispositivo, como falta de memória ou pequeno tamanho da tela. Há também o fato de a comunicação ser assimétrica, pois a velocidade de comunicação (largura de banda) é maior no sentido dos clientes que no sentido dos servidores, ou em alguns casos o cliente é mesmo incapaz de enviar mensagens aos servidores [Bar99]. Essa configuração sugere alternativas ao modelo em que o envio de dados obedece a solicitações explícitas do cliente. O *broadcast* seria então uma alternativa viável, com uma massa de dados sendo enviada a uma população de clientes, podendo ser ignorada ou aproveitada, modelo também chamado de *push-based*.

Esse novo ambiente traz novos desafios técnicos também na área de acesso à informação. Os usuários usam seus dispositivos móveis também para acessar a informação

necessária a partir de qualquer lugar, a qualquer momento, inclusive estando em movimento. Como se tratam de dispositivos móveis, os usuários podem querer acessar dados relacionados com a sua posição geográfica, ou informações relativas aos lugares para onde estão indo, o que também depende de localidades. Esses dados são chamados de Dados Dependentes da Localidade (*Location Dependent Data* – LDD), que são definidos como dados cujos valores são determinados pela localização a que estão relacionados [RD00].

Devido às questões de desempenho (baixa largura de banda) e desconexões, cogita-se trabalhar com consultas baseadas em localização usando *cache*. Se uma consulta é submetida várias vezes para ter sua resposta variando em função da localização, em submissões consecutivas pode haver uma repetição de alguns dos dados da resposta. Assim, ao menos parte da consulta pode ser respondida com dados armazenados localmente em *cache*, evitando tráfego na rede *wireless* e melhorando a performance do sistema [RD00].

A arquitetura em um sistema de usuários móveis também tem que sofrer algumas adaptações. Em um sistema de informação cliente/servidor fixo, um ou mais servidores armazenam os dados a serem acessados pelos clientes, assumindo-se que as localidades de clientes e servidores não mudam, bem como a conexão entre eles. Em computação móvel, a distinção entre clientes e servidores pode ser temporariamente confundida, gerando um modelo cliente/servidor estendido [JHE99]. Outra arquitetura usada para sistemas de bancos de dados móveis é o modelo *Hoard-Reintegrate* (Acumular-Reintegrar). Nela, os clientes replicam a parte da base de dados em que vão trabalhar, fazem as alterações necessárias mesmo estando desconectados e propagam essas alterações para o servidor quando este estiver disponível e houver uma nova conexão [Bad97].

Outra importante área de pesquisa é a garantia de consistência dos dados em bancos de dados móveis. É necessário que se usem recursos extras para manter essa consistência, uma vez que clientes podem ler dados de diversos servidores devido à sua movimentação e estes podem estar inconsistentes entre si. Como exemplo, um cliente pode escrever um valor em determinado servidor e, mais adiante, tentar lê-lo em outro servidor. Se não houver uma sincronização nesse tempo entre as operações o cliente irá operar com dados inconsistentes.

2.2. Localização de Objetos / Consultas Baseadas em Localização

Uma das principais capacidades adicionadas com a computação móvel é a possibilidade de fazer valer-se da localização de determinados objetos para oferecer serviços personalizados. Para isso, os sistemas devem estar cientes da constante mudança de localização dos usuários e prover um mecanismo de atualização eficiente para a manutenção da consistência das informações referentes a essa localização, de forma a não trabalhar com dados defasados.

Quando se trata de objetos que se movimentam e do armazenamento em um SGBD de informações referentes à sua localização, dois tipos de aplicação podem ser desenvolvidos. Pode-se considerar um cliente de uma rede móvel como portador de um dispositivo móvel (telefone celular, PDA's, computadores de bordo de automóveis, etc.) que, equipado com um dispositivo de GPS, fornece a sua posição e realiza algum tipo de consulta que será respondida baseada na localização fornecida. Assim, a localização do objeto é enviada juntamente com a consulta, não havendo necessidade de um armazenamento prévio desta informação. Por exemplo, um usuário de telefone celular realiza uma consulta para saber quais as farmácias mais próximas de sua localidade no momento da consulta. Nesse caso, o objeto que origina a consulta é móvel e fornece a sua localização, mas o objeto alvo da consulta, a farmácia, é imóvel.

Outro tipo de aplicação seria o armazenamento desses objetos em um SGBD, incluindo suas informações de localidade, que deveriam ser constantemente atualizadas. Nesse caso o alvo das consultas pode se movimentar também, as consultas podem envolver relações entre mais de um objeto ou requisitos temporais, além dos requisitos espaciais. Essas aplicações são chamadas de Banco de Dados de Objetos Móveis [SDK01].

Uma alternativa para estas aplicações seria ter um SGBD centralizado, que armazenasse todas as informações dos objetos, inclusive sua localização [PS01]. Exemplos de usuários favorecidos por essas aplicações seriam empresas de transporte (táxis, caminhões), que podem localizar seus veículos e realizar escolhas acerca do veículo mais apropriado para várias situações; ou militares, que poderiam reproduzir objetos móveis em um campo de batalhas, o que apoiaria o desenvolvimento de estratégias de ataque e defesa [SWC+97, PGS02]. Seria possível também envolver todos os objetos em determinada área geográfica em uma consulta, ou destiná-los a receber uma mensagem. Consultas a bancos de dados

centralizados com informações de localização podem partir de usuários estáticos ou dos próprios objetos móveis. O problema dessa alternativa é a constante atualização da base de dados devido ao armazenamento do posicionamento dos objetos, que se altera a cada movimento.

[JHE99] sugere o uso de *cache* para melhorar a performance das consultas baseadas em localização. Consultas respondidas são armazenadas localmente no *cache*; as demais consultas podem ter parte de suas respostas já armazenadas localmente. Assim, novas consultas são divididas em duas partes: uma que pode ser respondida localmente e outra que precisa ser submetida à base de dados. Há diferentes estratégias para dividir essas consultas em partes que podem e não podem ser processadas localmente.

Apesar do uso de dados em cache ser possível mesmo com o cliente estando desconectado, é necessária a manutenção de sua consistência com a base de dados, o que pode ser caro devido às constantes interrupções de comunicação. [JHE99] sugere dois mecanismos de validação de cache: *Varied Granularity of Cache Coherence* (Coerência de Cache de Granularidade Variada) e *Cache Invalidation Reports* (Relatórios de Invalidação de Cache). Na primeira abordagem, clientes e servidores mantêm um selo que identifica a versão dos dados armazenados. Ao utilizar os dados em cache, o cliente verifica a compatibilidade de seus dados com os do servidor quando uma conexão estiver disponível, validando as operações realizadas utilizando o cache. Na segunda abordagem, relatórios de modificação dos dados são enviados aos clientes pelo servidor, através de um canal de *broadcast*, com cada cliente verificando a validade de suas informações em cache. Nessa abordagem, tem-se a vantagem de não ser necessária uma conexão *uplink* (conexão no sentido cliente para servidor) para validar os dados em cache.

[PS01] considera o uso de cache baseado na premissa de que, após resolver uma consulta, esses dados podem ser reaproveitados para consultas subseqüentes originadas na mesma região. O cache fica armazenado em um *Visitor Location Register* (VLR), que é mantido em cada zona. Assim, se a localização do usuário que realiza a consulta possuir uma resposta em cache, sua consulta é respondida com os dados constantes do VLR. Ainda em [PS01] são sugeridas duas estratégias de invalidação de cache. Com *eager caching*, sempre que um usuário se movimenta, suas entradas no cache são imediatamente atualizadas. Essa operação, apesar de manter o cache sempre consistente, aumenta o custo da operação de movimentação. A outra estratégia é o *lazy caching*, onde uma movimentação não dispara

atualizações de cache. Então quando há uma consulta que é encontrada no cache, verifica-se se o usuário encontra-se naquela posição; caso contrário, a entrada é removida do cache.

2.3. Arquitetura para Bancos de Dados Móveis

[BP97] sugere um modelo com um banco de dados centralizado que atende a dois tipos de clientes: clientes comuns e *hoard clients* (clientes acumuladores). *Hoard clients* fazem *download* de parte dos dados do banco, operam em cima desses dados e propagam essas alterações de volta para o servidor, ou seja, reintegram esses dados já atualizados. *Hoard clients* podem operar mesmo estando desconectados da base, pois armazenam os dados necessários, enquanto clientes convencionais precisam estar conectados à base para operar. A figura 2.1 mostra o diagrama de estados para um *hoard client*.

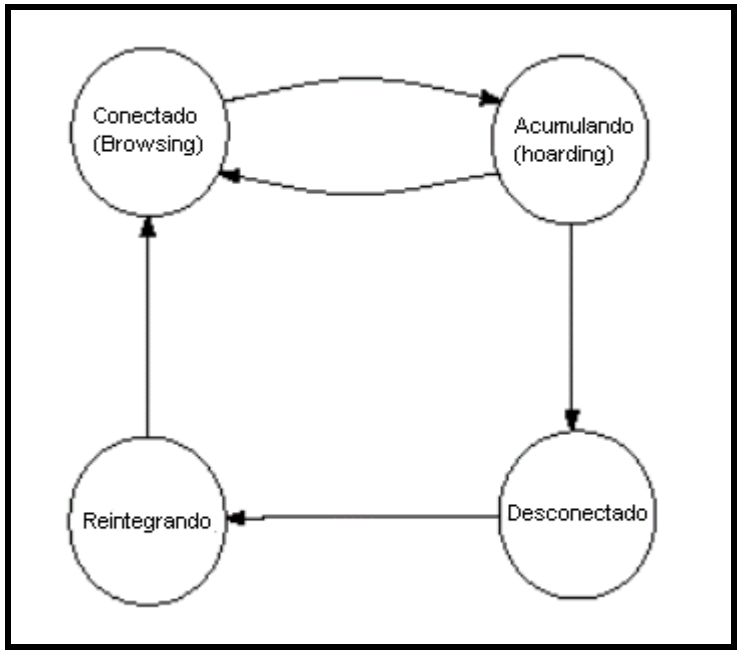

Figura 2.1. Diagrama de estados para um *hoard client*.

Um *hoard client* replica apenas a parte do banco de dados que lhe interessa. As relações são fragmentadas horizontalmente, aumentando a disponibilidade dos dados no modelo, uma vez que os dados do banco que forem replicados em um cliente são indisponibilizados para os demais clientes. Ainda de acordo com o modelo, a reintegração

pode ser feita a partir de qualquer algoritmo de junção de partições (*partition healing algorithms*).

Com a introdução de *hoard clients*, dois tipos de trabalho acontecem no servidor: cargas de trabalho geradas por clientes comuns ou cargas de *hoard clients*. As cargas de *hoard client* são geralmente uma seqüência de leituras, seguida por um período de inatividade, depois uma seqüência de escritas (reintegração). O cliente não precisa necessariamente reintegrar todo o fragmento que foi acumulado, mas apenas a parte que foi alterada.

Os *hoard clients* agem sob o princípio da localidade de acesso [Bad97]. Assim, os dados sobre determinada localidade são posicionados no servidor dentro de um mesmo segmento, de forma que todos sejam passados ao cliente de uma vez, melhorando a performance tanto no *download* dos dados como na reintegração. Muito embora um cliente não precise fazer *download* de todos os dados de um segmento, do ponto de vista do servidor ele o faz. O servidor deve manter um mapeamento entre os clientes e os fragmentos de dados que eles replicaram.

[JHE99] propõe um modelo chamado cliente-servidor estendido. Diferentemente do modelo cliente-servidor clássico, onde as funcionalidades de cada entidade são bem estabelecidas, em computação móvel as funcionalidades de cliente e servidor se confundem. As limitações de recursos dos clientes móveis muitas vezes exigem que estes tenham o comportamento do servidor para algumas situações. Assim, o servidor estático possui funcionalidades de cliente, assim como o host móvel possui algumas funcionalidades de servidor. A figura 2.2 mostra a arquitetura de um sistema cliente-servidor estendido.

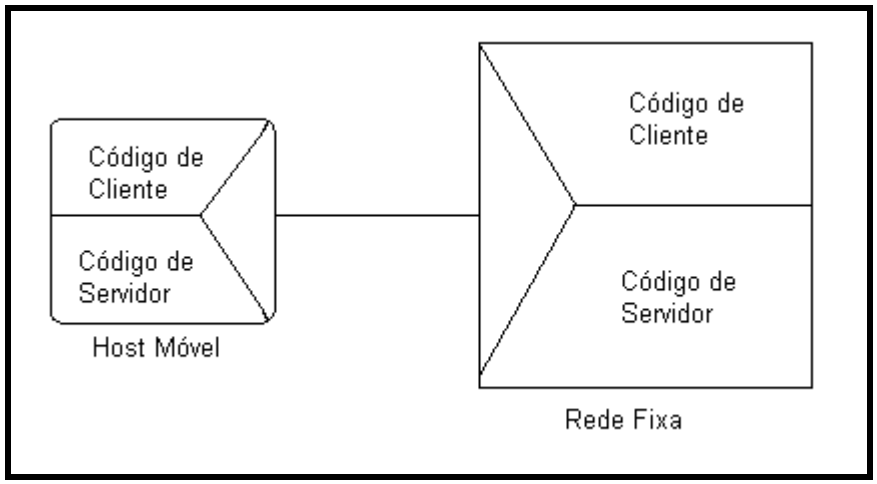

Figura 2.2. Arquitetura do modelo Cliente-Servidor Estendido

Duas variações do modelo são detalhadas: *thin client architecture* (arquitetura de cliente magro) e *full client architecture* (arquitetura de cliente completo). Em *thin client architecture*, a maior parte da lógica e funcionalidade dos clientes passa para servidores estacionários, que sabem que seus clientes são móveis e otimizam os serviços para tal fim. A arquitetura *thin client* é ideal para terminais burros ou pequenas aplicações em PDAs ou celulares.

A arquitetura *full client* é projetada para lidar com características desagradáveis da comunicação sem fio, como interrupções na conexão e baixa largura de banda (situação chamada de baixa conectividade). Assim, os clientes podem ser obrigados a operar em modo desconectado, por indisponibilidade de conexão ou por economia de recursos, como bateria e largura de banda. A forma para que os clientes operem em modo desconectado é transferir algumas das funcionalidades do servidor para a aplicação no cliente.

Há ainda a arquitetura *Flexible client-server*, que combina as arquiteturas *thin client* e *full client*. As funções de cliente e servidor podem ser dinamicamente realocadas e executadas em *hosts* móveis e estáticos.

2.4. Consistência dos Dados

A manutenção da consistência dos dados armazenados é obrigatória para qualquer sistema de banco de dados, não sendo diferente em um sistema de banco de dados móvel. Torna-se, porém, um desafio em sistemas desse tipo, devido a possíveis desconexões e operações com dados replicados da base.

Na arquitetura de *hoard clients* já citada [Bad97], o fato de que clientes fazem *download* de parte dos dados na base e operam nesses dados mesmo estando desconectados do servidor implica na utilização de alguma política de manutenção da consistência dos dados. Isso acontece, pois diferentes clientes podem tentar atualizar os mesmos dados com valores diferentes, no momento da reintegração dos dados ao servidor. São sugeridas no trabalho duas alternativas para lidar com problemas de consistência.

A primeira alternativa, chamada pessimista, é impedir que mais de um cliente faça *hoard* dos mesmos dados, fazendo um bloqueio (*lock*) nos segmentos já utilizados por um usuário, até que este promova a reintegração destes dados. A segunda alternativa, dita

otimista, permite que usuários trabalhem desconectados nos mesmos dados e resolve qualquer tipo de conflito na reconexão. A segunda alternativa, apesar de difícil de implementar, aumenta a disponibilidade dos dados.

Para realizar a alternativa otimista, realiza-se teste de serializabilidade das transações reintegradas. Assim, uma transação pode ser rejeitada se algum dos dados nela envolvidos é atualizado após o cliente fazer *hoard*. Este modelo de controle de consistência é aplicado globalmente, de forma independente do controle de consistência local (transações direto no servidor ou nos dados replicados nos clientes), que pode ser feito apenas fazendo bloqueio dos dados em utilização.

Em [TDP+94] é definido o conceito de garantias de sessão (*session guarantees*) para o problema de fraca consistência em dados replicados, o que pode ser útil para clientes móveis que acessem dados em uma série de servidores distribuídos a depender de sua localização. Esses clientes devem evitar gravar dados em determinado servidor e depois lê-los em outro servidor que ainda não tenha a operação de gravação refletida.

Uma sessão é uma abstração para a seqüência de operações de leitura e gravação realizada durante a execução de uma aplicação. Com a utilização de sessões, o que se busca é uma série de garantias para a manutenção da consistência dos dados operados por um cliente móvel. O resultado das operações executadas em uma sessão deve ser o mesmo da execução dessas operações em um único servidor centralizado.

2.5. Disseminação de Dados

Um outro desafio em um ambiente de clientes móveis e comunicação sem fio é a estratégia de acesso aos dados. Há alguns modos de entrega de dados (*data delivery*) do servidor para os clientes, como *server-push*, *client-pull* ou um modo híbrido [JHE99]. O modo *server-push delivery* é utilizado devido a ser bastante comum que, em redes sem fio, a velocidade de transmissão dos dados (largura de banda) seja muito maior no sentido servidor para cliente (*downstream*) do que no sentido cliente para servidor (*upstream*). Em alguns casos clientes sequer podem se comunicar com os servidores, o que torna impossível a solicitação de dados por parte do cliente, tornando *server-push delivery* uma alternativa atraente. O modo *client-pull delivery* tem as iniciativas tomadas pelo cliente, que faz

solicitações de dados ao servidor, enquanto o modo híbrido usa ambas as definições. O modo híbrido é também chamado de *Interleaved Push and Pull* (IPP).

[Bar99] propõe o uso de transmissão de dados em *broadcast* para sistemas de bancos de dados móveis. O *broadcast* seria utilizado em aplicações com comunicação assimétrica, aplicações onde o tráfego no sentido *downstream* é muito maior do que no sentido *upstream* e as informações têm que chegar a um maior número de clientes.

Quando um servidor envia dados através de *broadcast* para uma comunidade de clientes repetida e continuamente, o canal de *broadcast* torna-se um "disco", de onde os clientes retiram dados [JHE99]. Mais de um disco pode ser formado, com diferentes velocidades e prioridades de leitura, sendo os dados de diferentes discos intercalados por um mesmo canal de *broadcast* Dados de um disco mais veloz (dados de maior prioridade) seriam disponibilizados no canal com maior freqüência. Essa configuração chama-se *broadcast disks*. A figura 2.3 mostra a arquitetura dos três modos de disseminação de dados descritos.

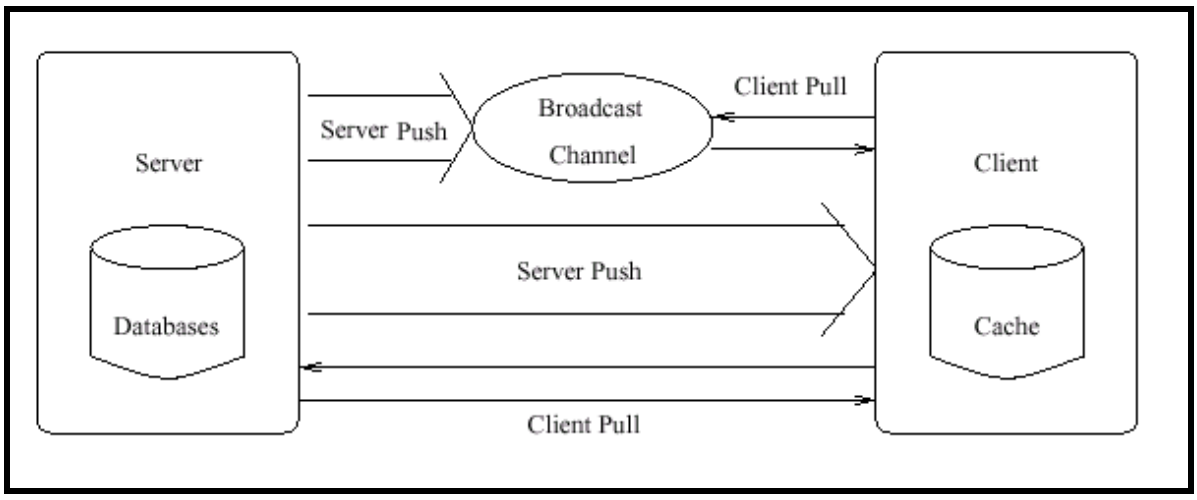

Figura 2.3. Acesso a dados em arquitetura móvel

[JHE99] define também uma técnica chamada *Indexing on Air*. Usando *push-based*, algumas vezes vão a *broadcast* itens muito requisitados, ou *hot spots*. Com *Indexing on Air*, o servidor pode criar dinamicamente um índice para aumentar o envio de *hot spot*, baseado na monitoração periódica da demanda. O índice é ajustado para diminuir dois parâmetros, *query time*, ou o tempo passado entre a realização da consulta por parte do cliente e o recebimento da resposta; e *listening time*, que é o tempo que o cliente passa ouvindo o canal de *broadcast*.

2.6. Objetos Móveis

Uma das vantagens surgidas com a computação móvel e as redes de comunicação sem fio foi a possibilidade de trabalhar com objetos que se movimentam. Essas tecnologias possibilitaram a um objeto que se movimenta estar equipado de um dispositivo computacional que se comunica com uma unidade central ou com outros objetos, fornecendo informações atualizadas sobre sua localização. Essas informações podem ser armazenadas em um banco de dados, possibilitando a realização de consultas cujas respostas variam dependendo da localização do objeto. Usuários ou dispositivos que reportam sua localização para realização de consultas cujos resultados são dela dependentes são chamados de objetos móveis [WXC+98].

Objetos móveis são dispositivos que se movem de maneira contínua, alterando sua posição no espaço com o decorrer do tempo. SGBD's tradicionais não são preparados para manipular dados continuamente modificados [SWC+99], pois os dados são tratados como valores constantes até que sejam explicitamente modificados. Por tratar constantemente com informações envolvendo espaço e tempo, um banco de dados de objetos móveis é uma especialização de bancos de dados espaço-temporais, com informações espaciais e temporais variando de forma contínua e discreta [BP00]. O desenvolvimento de um sistema de banco de dados que trata de variação da informação de forma contínua atende também à variação discreta da informação, que pode ser encontrada em certos tipos de objeto, como fronteiras de estados ou cadastros de propriedade [EGS+99].

Considerando-se um usuário móvel de um banco de dados, um leque de consultas personalizadas baseadas na localização do usuário pode ser realizada. Para tanto, há uma série de estratégias para localização de usuários móveis e identificação da sua posição no espaço. A localização desses objetos é realizada através de um sistema de GPS, que pode ser combinado ao protocolo IP (*Internet Protocol*) para possibilitar a criação de serviços dependentes da localidade [Bar99].

Em um sistema de objetos espaciais podem ser desenvolvidas aplicações que utilizem mobilidade em diferentes pontos da consulta. A seção a seguir faz uma taxonomia dos diferentes tipos de aplicação de objetos móveis que podem ser desenvolvidos.

2.6.1. Taxonomia de Aplicações de Objetos Móveis

Pode-se considerar algumas alternativas para o desenvolvimento de uma aplicação de objetos móveis. A base de dados pode conter ou não informações acerca dos próprios objetos móveis, podendo inclusive manter armazenadas informações atualizadas sobre a localização dos objetos. Além disso, numa consulta, a mobilidade pode estar tanto no objeto que realiza a consulta como no alvo dessa consulta, considerando-se como alvo qualquer objeto sobre o qual a consulta seja realizada. Assim, pode-se estabelecer uma série de combinações entre estas opções; essas combinações são descritas a seguir.

a) Aplicações sem dados sobre localização dos objetos armazenados na base

Neste tipo de aplicação, por não haver na base de dados informações sobre a localidade dos objetos que realizam as consultas, em se tratando de consultas dependentes da localização, esta informação deve também ser submetida. Pode-se dividir em quatro tipos de aplicação, relatados a seguir.

a.1 – Objetos móveis x alvos móveis

Nesse tipo de aplicação a consulta é realizada por um usuário a partir de um dispositivo móvel, como um PDA ou telefone celular, tendo como alvo objetos que se movimentam e, estes sim, têm informações armazenadas na base.

Um exemplo de aplicação seria um sistema de controle de táxis. Qualquer usuário pode solicitar um táxi, que seria o alvo da consulta. É interessante para a operadora descobrir qual táxi está disponível mais próximo do solicitante, sendo, portanto, necessário conhecer a sua localidade. O sistema possui em sua base apenas informações sobre os táxis, sendo a localidade do usuário fornecida no momento da consulta.

a.2 – Objetos móveis x alvos imóveis

Nesse tipo de sistema o usuário também solicita informações a partir de um dispositivo móvel, porém os alvos da consulta não se movimentam. A base de dados continua tendo informações apenas sobre o alvo da consulta.

Um exemplo de aplicação seria um sistema de consulta a estabelecimentos, como farmácias ou restaurantes. O usuário pode realizar consultas sobre estabelecimentos próximos

de sua posição ou em outras localidades, sempre necessitando fornecer a localidade para o sistema basear a resposta à consulta.

a.3 – Objetos imóveis x alvos móveis

Nesse caso os usuários acessam o sistema a partir de um dispositivo que não se movimenta, como um *desktop* ou algum tipo de terminal de consulta. O sistema de táxis, já citado, serve como exemplo para esse tipo de aplicação também; a única diferença fica por conta do dispositivo que realiza a consulta, mas também tem a necessidade de fornecer sua localização.

a.4 – Objetos imóveis x alvos imóveis

Esse tipo de sistema não envolve mobilidade, apenas requisitos espaciais do banco de dados. O exemplo de consulta a estabelecimentos é válido aqui também, considerando-se que o usuário acessa o sistema a partir de um dispositivo fixo.

b) Aplicações com dados sobre os objetos armazenados na base

Em aplicações deste tipo os objetos que irão realizar consultas têm a sua localidade expressada dentro da base de dados. Desta forma, não é necessário fornecer essa informação no momento da consulta, porém devem ser tomados vários cuidados para não utilizar dados defasados contidos no banco, o que geraria respostas inconsistentes às consultas. Quatro tipos de aplicação são descritos a seguir.

b.1 – Objetos móveis x alvos móveis

Tipo de aplicação em que os objetos são dispositivos móveis e suas informações de localidade são armazenadas e atualizadas constantemente na base de dados. Assim, é possível processar consultas não só entre diferentes objetos (como no caso usuário-estabelecimento), mas também entre objetos do mesmo tipo, que se relacionam. As informações de localização do objeto obedecem a uma política de atualização estabelecida na aplicação.

Uma aplicação desse tipo seria útil, por exemplo, para unidades da policia ou corpo de bombeiros. Uma unidade (veículo) em busca de reforços pode localizar outras unidades próximas de locais adequados.

b.2 – Objetos móveis x alvos imóveis

Em aplicações desse tipo a mobilidade fica restrita aos objetos que disparam a consulta. As informações tanto de localização dos objetos como de localização dos alvos devem estar armazenadas na base de dados.

Um exemplo seria o sistema de uma empresa prestadora de serviços de saúde. Uma ambulância transportando um paciente com urgência dispara uma consulta para saber a localização dos hospitais mais próximos.

b.3 – Objetos imóveis x alvos móveis

Neste caso não se considera a mobilidade como requisito para o objeto que dispara a consulta, mas continua-se considerando seus requisitos espaciais. Os objetos continuam tendo suas informações de localidade armazenadas na base de dados, mas por serem imóveis não necessitam de política de atualização. Já o alvo das consultas é tido como móvel, tendo suas informações de localidade armazenadas na base, e necessitando de políticas de atualização.

Como exemplo, pode-se imaginar a administradora de uma empresa de transporte de cargas, realizando consultas sobre a localização de seus caminhões.

b.4 – Objetos imóveis x alvos imóveis

Esse tipo de aplicação é o tipo tradicional de banco de dados, sem requisitos de mobilidade.

Nesse trabalho, ao citar aplicações e bancos de dados de objetos móveis estar-se-á falando de aplicações onde há na base de dados informações sobre a localização de objetos que se movimentam, sejam eles realizadores ou alvo das consultas. Na taxonomia realizada, seriam os tipos de aplicação a.1, a.2, b.1, b.2 e b.3. Assim, estará sendo seguido o padrão observado na literatura produzida pela comunidade de pesquisa em aplicações de bancos de dados móveis.

2.6.2. Características de Objetos Móveis em Bancos de Dados

O armazenamento de objetos móveis em bancos de dados tradicionais exige o tratamento de uma série de questões que vem sendo tema de pesquisas. A primeira das questões é o fato de que um banco de dados de objetos móveis deve armazenar informações atualizadas sobre a localização dos objetos. Essas informações são extremamente inconstantes, uma vez que variam todo o tempo em que o objeto se move. Fazer atualizações constantes no banco é insatisfatório, pois provoca um declínio na performance do SGBD e sobrecarga na rede de comunicação, enquanto atualizações menos constantes poderiam levar a respostas desatualizadas às consultas. Alguns trabalhos [SWC+99, WXC+98, EGS+99, PLM01] propõem que os objetos tenham sua posição atualizada em função do tempo, ao invés de utilizar *updates* explícitos. Para isso são usados os chamados atributos dinâmicos, atributos que mudam continuamente em função do tempo [SWC+99, PLM01].

Como os objetos estão em constante movimentação, cada informação de localização corresponde a um instante de tempo no qual aquela informação foi observada. Assim, objetos móveis são compostos de informações espaciais e temporais. Com bancos de dados espaço-temporais é possível armazenar parâmetros e funções de movimentação, que seriam capazes de determinar a posição do objeto no espaço em determinado instante de tempo. Baseado nisso, é possível fazer previsões de posicionamento em tempo futuro, de acordo com a função e parâmetros de movimentação constantes do banco no momento da consulta [SWC+99].

Uma outra abordagem [BP00] estabelece movimentações não apenas em eixos X e Y do plano cartesiano, mas também alterações em um ângulo, que definem uma movimentação em torno do próprio eixo do objeto. Além disso, são utilizadas duas condições para tentativas de previsão da movimentação do objeto: um perfil que é atribuído a cada classe de objeto e o histórico das movimentações anteriores. Baseado nessas duas condições, tenta-se reduzir o número de observações do posicionamento do objeto, reduzindo com isso o tráfego de rede e carga de atualizações no SGBD.

A seguir, serão detalhados alguns dos problemas que têm sido objeto de estudo na área de objetos móveis, bem como algumas das soluções apresentadas para tentar resolvê-los.

- Modelagem da Localização

Ao armazenar objetos móveis em um banco de dados, esbarra-se no problema de como manter informações atualizadas sobre a localização do objeto no espaço. Bancos de dados tradicionais não estão preparados para manusear dados que se modificam constantemente, pois os dados são considerados constantes até o momento em que são explicitamente atualizados. O objeto pode ser projetado para fornecer essa informação com a freqüência que se julgar necessária, porém atualizar o banco com muita freqüência gera um tráfego excessivo na rede e sobrecarga no SGBD. Por outro lado, atualizar essa informação com pouca freqüência poderia gerar respostas defasadas a consultas.

Algumas soluções alternativas vêm sendo então buscadas. Uma delas é o uso de atributos dinâmicos. Atributos dinâmicos, por definição, são atributos que mudam continuamente em função do tempo, sem a necessidade de um *update* explícito [SWC+97]. Assim, esses atributos armazenam funções matemáticas de movimentação do objeto. A posição do objeto em determinado instante é calculada a partir do tempo em que a consulta é submetida.

Isso cria a possibilidade da realização de consultas prevendo a localização do objeto em um tempo futuro, baseado nos parâmetros da função de localização armazenados no banco de dados no momento da consulta. Mas isso caracteriza uma tentativa de localizar o objeto em um tempo futuro, uma vez que ele pode alterar sua movimentação antes do tempo da consulta, mudando sua trajetória.

Uma outra solução para diminuir a necessidade de atualizações na base de dados, buscando aumentar a chance de acerto na previsão de posicionamento dos objetos é a utilização de perfis de objeto e histórico de movimentação [BP00]. Perfis de movimentação são definidos juntamente com a classe e consideram uma série de restrições de movimentação naturais do objeto. Estas restrições podem ser de eixo de movimentação, como no caso de elevadores, que só podem se mover na vertical; restrições de meio, como esperar que automóveis se movam através de auto-estradas ou navios se movam na água; e há restrições de velocidade máxima para cada tipo de objeto.

Históricos de movimentação são gerados a partir de uma série de observações realizadas sobre a movimentação do objeto, buscando gerar padrões que facilitem a previsão de suas movimentações futuras. Como exemplo, pode se imaginar um animal que repita várias

vezes ao dia um certo trajeto, ou um automóvel que diariamente segue da residência para o trabalho, ou também a velocidade que certo veículo procura estabelecer ao trafegar em auto-estradas. Históricos de movimentação são úteis também para realizar consultas referentes a posições ocupadas pelo objeto no passado.

- Imprecisão

A localização de objetos traz inerente uma certa imprecisão, que será dependente da política de atualização utilizada. Há um *tradeoff* entre a velocidade de atualização dos parâmetros da função de localização e a imprecisão obtida. Atualizações muito freqüentes provocam sobrecarga na rede e queda na performance do SGBD, mas atualizações pouco freqüentes provocam uma defasagem nas respostas às consultas, devido ao aumento da imprecisão na localização. Há pesquisas no sentido de definir qual o equilíbrio ideal entre a velocidade de atualização e a imprecisão aceitável [WSCY99, TWZC02].

O tratamento da imprecisão na localização dos objetos depende da política de atualização utilizada. Independentemente da política de atualização utilizada deve ser estabelecido um intervalo de tempo após o qual o objeto deve enviar novas informações a respeito de sua localização. Em instantes de tempo entre estas atualizações, a posição irá ficar defasada, gerando imprecisão nas respostas às consultas. Essa imprecisão pode ser tratada com o uso dos atributos dinâmicos para gerar uma previsão de novo posicionamento do objeto, fazendo com que a margem de erro seja reduzida, levando a consulta a ser respondida com uma posição mais próxima da real posição do objeto. Mesmo realizando previsões de uma nova posição baseada em movimento uniforme (velocidade constante), reduz-se a necessidade de atualização em até 85% [WSCY99].

Outra política de combate à imprecisão é estabelecer um limite que, ao ser ultrapassado, obriga o objeto a fazer a atualização imediatamente, independentemente de ter alcançado ou não o momento estabelecido para a atualização. O banco de dados é atualizado sempre que a distância entre a localização de um objeto móvel *m* e a sua indicação no banco de dados excede um limite *th*. Dois conceitos são estabelecidos: desvio e incerteza (*deviation* e *uncertainty*). Desvio é a diferença entre posição real e posição no banco de dados em um determinado instante de tempo, enquanto incerteza é o tamanho da área onde o objeto possivelmente deve estar.

Com isso, é possível ter um controle sobre a imprecisão, pois se tem uma noção do quão distante estará o objeto da posição prevista. Assim, é estabelecido que a localização do objeto não é um ponto no espaço, mas sim uma região onde ele potencialmente está. Políticas de atualização com essas características são chamadas de *dead-reckoning policies* (políticas de cálculo inoperante), e têm a desvantagem de exigir poder de processamento no cliente. O cliente deve realizar as mesmas operações de cálculo que o servidor realiza para saber onde o servidor imagina que ele esteja, de forma a perceber que o limite de desvio foi ultrapassado e fornecer uma nova localização.

Dois operadores podem ser acrescentados à linguagem de consulta de objetos móveis utilizada, específicos para o tratamento da imprecisão [WSX+99]. São os seguintes:

MAY – Um objeto será retornado se seu intervalo de imprecisão intercepta a região da consulta no intervalo solicitado.

MUST – Um objeto será retornado se seu intervalo de imprecisão está totalmente contido na região da consulta no intervalo solicitado.

Em [WSCY99] são descritas algumas variações da política de *dead-reckoning*: *speed dead-reckoning*, em que o limite *th* é mantido constante durante toda a movimentação do objeto; *adaptative dead-reckoning*, em que são calculadas previsões de movimentação do objeto e fornecidos diferentes limites a cada atualização; *disconnection detecting dead-reckoning*, onde o limite *th* é reduzido com o passar do tempo desde a atualização, numa tentativa de evitar que o objeto exceda o limite e não comunique ao servidor por estar desconectado.

- Processamento Distribuído de Consultas

Em algumas implementações, o banco de dados pode estar distribuído. No caso de dispositivos móveis equipados de poder computacional e capacidade de armazenamento pode-se considerar a possibilidade de distribuir o banco de dados nos próprios objetos móveis. É interessante distribuir os dados de forma que cada objeto fique armazenado em seu próprio

dispositivo real, não sendo replicado em lugar algum [SWC+97]. Dessa forma três tipos de consultas poderiam ser identificadas:

Self-referencing Queries – Apenas atributos do próprio objeto são necessários, podendo a consulta ser realizada localmente, sem a necessidade de comunicação entre máquinas. Por exemplo: "Em que momento alcançarei o ponto (a,b)?".

Object Queries – Consultas onde cada objeto responde por si próprio, enviando suas respostas individuais a uma unidade central, que monta a resposta da consulta. Possível no caso de consultas que não necessitam da interação entre objetos. Exemplo: "Recupere os objetos que alcançarão o ponto (a,b) em 3 minutos". Nessa consulta, cada objeto responde à unidade central se alcançará o ponto em três minutos, ficando a unidade responsável por listar os objetos que responderam positivamente.

Relationship Queries – Consultas que dependem de mais de um objeto. Cada um envia seus dados a uma localidade central, que processa então a consulta. Exemplo: "Recupere os objetos que permanecem a menos de 2 Km um do outro pelos próximos 3 minutos". Nesse caso, a unidade central recebe informações que a possibilitam de montar a trajetória dos objetos e verificar quais os pares que obedecem ao que foi requisitado pela consulta.

- Tipos de consulta para objetos móveis

Uma forma de classificar consultas é utilizar os conceitos de faixa e vizinhança, onde o primeiro retorna objetos que estão em uma determinada faixa de espaço e/ou tempo, enquanto o segundo retorna os objetos que estão relativamente mais próximos do ponto determinado pela consulta. Os conceitos de faixa e vizinhança são aplicados independentemente às condições de espaço e tempo [PLM01]. Isso gera os seguintes tipos de consulta:

Consulta por faixa espacial e temporal – Exemplo:

"Retorne todos os veículos que passaram até 1 Km da localização de um determinado acidente entre 4-5 da tarde".

Neste caso, a faixa espacial é até 1 km do local da consulta e a faixa temporal é entre 4-5 da tarde.

Consulta por vizinhança temporal e faixa espacial – Exemplo:

"Retorne todos os veículos até 100 m de um acidente, ordenando pela diferença de tempo entre a ocorrência do acidente e a passagem do veículo".

Aqui, a faixa espacial corresponde a até 100 m do local do acidente e a vizinhança temporal corresponde à maior proximidade com o momento em que ocorreu o acidente. A ordem dos objetos retornados corresponde aos que mais cedo, a partir do momento do acidente, passaram dentro da faixa determinada.

Consulta por vizinhança espacial e faixa temporal – Exemplo:

"Retorne as 5 ambulâncias que estiveram mais próximas do local do acidente entre 4-5 da tarde".

Nesta consulta, a vizinhança espacial é a proximidade com o local do acidente e a faixa temporal é entre 4-5 da tarde.

Não é possível realizar consultas por vizinhança temporal e espacial, pois perde-se o referencial fixo e duas noções de proximidade passam a influenciar o objeto. Por exemplo na consulta hipotética:

"Retorne as ambulâncias que passaram mais próximas do local do acidente e mais próximas do momento do ocorrido",

Fica difícil definir que ambulância retornar entre uma que passou 1 minuto depois do acidente, mas a 2 km do local, e outra que passou a 100 m do local, mas depois de uma hora do ocorrido. Por isso não se trabalha com as duas entidades (espaço e tempo) consultadas por vizinhança.

- Linguagem de Consulta

Consultas em aplicações MOD (*Moving Objects Database*) requerem a definição de condições espaciais e temporais. Há trabalhos que procuram definir uma linguagem específica para aplicações MOD, integrando ambas as funcionalidades espacial e temporal.

Uma linguagem adequada a aplicações MOD deve responder a consultas do tipo:

"Recupere os objetos que irão passar pelo polígono P nos próximos 3 minutos".

"Retorne os pares de aeronaves inimigas que vão passar a uma distância de 10 Km uma da outra e o tempo em que isso irá ocorrer".

Em [SWC+97], é especificada uma linguagem específica para a realização de consultas em aplicações MOD, a FTL (*Future Temporal Logic*). FTL é uma linguagem de consulta, como SQL ou OQL, desenvolvida para expressar condições espaço-temporais em objetos móveis. Alguns exemplos de consultas em linguagem FTL serão mostrados:

"Retorne os pares de objetos *o* e *n* cuja distância fica no máximo em 5 Km até que ambos entrem no polígono *P*".

RETRIEVE o,n

FROM Moving-Objects

WHERE begin_time(DIST(o,n) ≤ 5) ≤ now

and end_time(DIST(o,n) ≤ 5) ≥

begin_time((INSIDE(o,P) ^ INSIDE(n,P))

Nesta consulta, a primeira linha da cláusula WHERE garante que os objetos já distavam menos de 5 Km no momento inicial da consulta. As duas linhas seguintes garantem que os objetos só irão distar mais de 5 Km depois de terem estado ambos dentro do polígono P.

Outro exemplo: "Recupere os objetos *o* que entram no polígono P nas próximas 3 unidades de tempo e permanecem por outras 2 unidades de tempo".

RETRIEVE o

FROM Moving-Objects

WHERE Eventually_within_3 ((INSIDE(o,P) ^ Allways_for_2 INSIDE(o,P))

No exemplo anterior, a clausula WHERE garante a entrada do objeto no polígono P a qualquer momento nos próximos 3 minutos (operador Eventually_within_3) e a permanência por 2 minutos (operador Allways_for_2).

Outro trabalho [VW01] busca utilizar recursos do modelo Objeto-Relacional e da linguagem SQL-99 para realizar acesso aos dados de objetos móveis. Para tanto, são definidos novos tipos de objeto para representar os objetos móveis e as funções necessárias para manusear os objetos móveis. Um exemplo de consulta vem a seguir:

"Recupere cada objeto em que, a qualquer momento entre starttime e endtime, o objeto esteja a menos de 5 minutos de R"

SELECT id FROM M_O
WHERE id WITHIN 5 mins FROM R ALONG EXISTING PATH
SOMETIME BETWEEN starttime and endtime

2.7. Trabalhos Relacionados

2.7.1. Modelo de Dados MOST

O modelo de dados MOST (*Moving Objects Spatio-Temporal*) foi desenvolvido para lidar com as questões referentes à inclusão de condições espaço-temporais nas consultas, além de lidar com atributos dinâmicos para o armazenamento da posição de Objetos Móveis. O modelo foi apresentado nos trabalhos [SWC+97, WXC+98, SWC+99, WSX+99].

O modelo MOST armazena vetores de movimentação, que são usados para cálculo da posição do objeto no tempo solicitado pela consulta. Ao invés de atualizar constantemente sua posição, os objetos atualizam apenas o seu vetor de movimentação nos momentos em que mudam o movimento que vinham realizando. Baseado nos vetores de deslocamento, é possível realizar consultas de posições previstas para um tempo futuro.

No MOST, o banco de dados é um conjunto de classes, sendo que uma classe especial fornece o tempo a cada instante. Algumas classes são designadas como espaciais; uma classe espacial possui três atributos chamados X.POSITION, Y.POSITION, Z.POSITION, que descrevem a posição do objeto no espaço. Objetos de classes espaciais possuem uma série de métodos que representam relações entre objetos em determinado instante de tempo. Métodos como INSIDE(n,P) e OUTSIDE(n,P), que retornam valores booleanos e indicam se o objeto *n* está dentro ou fora do polígono *P* em determinado instante de tempo. Há também métodos como WITHIN-A-SPHERE(r,o_1,...,o_k), que verifica se os objetos relacionados podem ser englobados por uma esfera de raio *r*, retornando um valor booleano; ou o método DIST(o_1,o_2) que retorna um inteiro que representa a distância entre os objetos. Os dois últimos métodos podem ser considerados métodos de relacionamento, por envolverem mais de um objeto.

O modelo MOST incrementa as capacidades de um SGBD tradicional[1], funcionando como uma camada de software sobre ele. Consultas enviadas ao SGBD são analisadas e possivelmente modificadas pelo MOST, o mesmo ocorrendo com respostas do SGBD.

- Atributos Dinâmicos

Considerando-se um objeto móvel e a determinação de sua posição em duas coordenadas: *x* e *y* cada uma delas torna-se um atributo dinâmico. Um atributo dinâmico *A* é representado por três sub-atributos: *A.value*, *A.updatetime*, *A.function*, onde a função contém o vetor de movimentação, enquanto o valor representa a posição do objeto no tempo indicado em *updatetime*. As posições são calculadas verificando-se a diferença entre o tempo da consulta e o *updatetime*, sendo aplicado à função de deslocamento. Assim, a resposta a uma mesma consulta pode ser diferente em dois momentos, mesmo que o banco de dados não tenha sido atualizado.

- Tipos de Consultas MOST

O MOST diferencia três tipos de consulta: instantânea, contínua e persistente. Os três tipos serão descritos e exemplificados a seguir.

Consulta Instantânea: Uma consulta instantânea refere-se a um instante determinado de tempo. Como a função de movimentação calcula o posicionamento de acordo com o tempo da consulta, esta pode se referir também a tempos futuros, além de presente e passado. Por exemplo:

"Mostre os hotéis a um raio de 500m da minha posição daqui a 5 minutos"

Consulta Contínua: Uma consulta contínua é re-submetida ao banco de dados constantemente, ou até que seja cancelada ou expire. Pode ser considerada uma seqüência de consultas instantâneas, para cada instante de tempo maior que o tempo em que a consulta foi submetida. Por exemplo:

"Liste os hotéis em um raio de 1 Km".

[1] Considera-se, neste trabalho, SGBD tradicional aquele que armazena dados alfanuméricos, sem necessidade de estruturas especiais para armazenamento de tipos de dados complexos.

Essa consulta é re-submetida a cada unidade de tempo, gerando resultados possivelmente diferentes a cada novo posicionamento do objeto, com novos hotéis entrando na resposta e outros saindo.

Consulta Persistente: São consultas que necessitam ao menos de um breve armazenamento histórico. Isso porque os resultados dependem das atualizações submetidas à função de movimentação. Por exemplo:

"Recupere os objetos que dobrarão sua velocidade nos próximos 10 minutos".

Para responder a essa consulta o SGBD deve ter um registro das velocidades ocorridas durante o tempo da consulta.

2.7.2. Projeto CHOROCHRONOS

Os trabalhos [EGS+99, GBE+00, FGN+00], gerados no projeto CHOROCHRONOS propõem a definição de *moving points* e *moving regions*, cujo comportamento seria capturado modelando-os como tipos de dados abstratos, sendo abstrações básicas para a modelagem de um sistema de objetos móveis. *Moving points* definiriam o deslocamento de um objeto qualquer, enquanto *moving regions*, além do deslocamento de regiões inteiras, definiriam crescimento e redução de regiões. Ambas são consideradas entidades tri-dimensionais (espaço bi-dimensional + tempo). Apesar de definidas para o espaço bi-dimensional, ambas as abstrações são válidas também para o espaço tri-dimensional.

[GBE+00] define formalmente os tipos de dados para *moving points* e *moving regions*, bem como uma série de operações nestas entidades. O banco de dados consiste de um conjunto de classes objeto, cada uma associada a um conjunto de objetos. As classes contêm atributos que são definidos como sendo de tipos de dados atômicos. Cada tipo de dados móvel é um mapeamento de um valor do domínio tempo em um valor do domínio espaço. A definição de todos os tipos de dados necessários para trabalhar com os objetos móveis gera um modelo abstrato.

O modelo abstrato é um modelo mais simples, porém impossível de implementar. Isso porque as mudanças nas informações de localização dos objetos são geralmente contínuas. É definido então o modelo discreto, onde para cada tipo de dados abstrato é definido um

correspondente discreto. [FGN+00] define as restrições necessárias para cada representação contínua ser representada por um valor finito correspondente no modelo discreto.

Tipos de dados como *int, real, string, bool* são comuns, normalmente implementados por qualquer SGBD. O tipo *point* representa um ponto no plano cartesiano, *points* representa um conjunto de pontos, *line* uma seqüência de curvas e *region* um subconjunto do plano, que pode ou não conter buracos no seu interior. O tipo *instant* oferece um domínio temporal na mesma forma dos números reais, *range* gera um conjunto de elementos do domínio do argumento passado e *intime* associa um instante de te mpo ao valor do argumento passado.

O construtor *moving* possibilita a qualquer um dos tipos citados possuir uma série de valores contínuos variando em função do tempo para o domínio que for passado como argumento, gerando os tipos de dados temporais. Por exemplo, *moving*(*point*) irá possuir diferentes valores de pontos associados com diferentes intervalos de tempo. Através de uma operação chamada *lifting,* qualquer operação aplicada a tipos estáticos pode ser também aplicada a tipos temporais correspondentes.

Todos os tipos de dados definidos no modelo abstrato têm uma contrapartida no modelo discreto. [FGN+00] define também uma especificação de alto nível de estrutura de dados que seriam utilizadas para mapear tipos de dados do modelo discreto para SGBDs espaço-temporais.

2.7.3. Perfis e Histórico de Movimentações

O trabalho [BP00] não define um modelo completo para aplicações de objetos móveis, mas faz algumas sugestões de adendos para prover um modelo mais eficiente. Suas principais sugestões são os perfis de movimentação e histórico de movimentação.

A definição de perfis de movimentação é uma tentativa de aumentar a precisão das previsões de posições não observadas de um objeto, onde se considera observação uma posição informada pelo objeto à base de dados. Essas previsões podem ser aprimoradas considerando-se apenas os movimentos que o objeto pode realizar. Para tanto, considera-se desde características de comportamento do objeto até restrições do meio físico por onde ele pode se movimentar.

Como comportamento do objeto, têm-se características como a velocidade máxima que o objeto pode alcançar, ou restrições de eixo de movimentação. Exemplos de restrições

do eixo de movimentação são objetos que só podem se movimentar no eixo vertical, como elevadores, ou que só podem se movimentar em um plano, como qualquer objeto incapaz de sair do chão.

O histórico de movimentação também pode ser útil para aumentar a precisão das previsões de posicionamento realizadas. O histórico é gerado através da seqüência de observações realizadas e pode definir uma série de movimentos típicos para determinado objeto, ou seja, movimentos que são realizados com freqüência. Como exemplo, pode-se ter pássaros que migram em determinada estação do ano, ou um motorista que faz diariamente o percurso de casa para o trabalho. Informações desse tipo podem ser apuradas através de técnicas de mineração de dados. De posse dessas informações, ao prever um movimento pode-se optar pelo movimento comumente realizado, aumentando as chances de acerto.

2.7.4. Modelo Proposto por Bei Yi e Claudia Bauzer Medeiros

No modelo do trabalho [YM02], define-se a trajetória dos objetos móveis como objetos espaciais. Assim, a trajetória de um objeto pontual é um conjunto de pontos que forma uma reta, a de um objeto linear é um conjunto de linhas que formam um polígono, e a trajetória de um polígono forma um poliedro, estrutura tridimensional cuja projeção no espaço 2D é também um polígono.

Dessa forma, procura-se trabalhar com os objetos móveis sem envolver o atributo temporal. Assim, busca-se implementar alguns operadores de maneira similar a operadores estritamente espaciais. Um objeto móvel tem associada a si uma trajetória, correspondente a determinado intervalo de tempo, e procura-se responder a consultas de objetos móveis baseado em relações espaciais entre as geometrias que compõem as trajetórias dos diferentes objetos e outras geometrias que formam o mapa por onde os objetos se movimentam.

2.7.5. Comparativo Entre os Modelos Analisados

A tabela 2.1 estabelece um comparativo entre os modelos analisados, de acordo com algumas características de aplicações de objetos móveis.

Modelo / Característica	MOST	CHOROCHRONOS	Beard	Yi
Banco de dados espaço-temporal	√	√	√	√
Histórico de movimentações	√	√	√	√
Atributos dinâmicos	√	X	√	X
Operadores de objetos móveis	X	√	X	X
Aspectos de mobilidade em regiões	X	√	√	√
Aspectos de mobilidade no eixo do objeto	X	X	√	√
Não necessita conhecimento prévio de trajetória	√	√	√	√
Consultas em linguagens padrão de banco de dados	X	√	X	X

Tabela 2.1. Comparativo entre os modelos analisados

2.8. Considerações Finais

Este capítulo tratou da fundamentação teórica acerca do assunto a ser tratado na dissertação. Primeiramente foi abordada a computação móvel de forma geral, especificando-se em seguida para aplicações de bancos de dados que envolvem mobilidade. A seguir, abordou-se os sistemas de informação geográfica, parte importante em qualquer aplicação que envolva armazenamento de dados espaciais. Por fim foram abordadas as questões de objetos móveis em si, sendo mostradas as características de sistemas desse tipo e analisando os trabalhos já existentes na área.

Capítulo 3. MOMENT – Modelo de Objetos em Movimento

3.1. Introdução

No capítulo anterior foram vistos vários conceitos relativos a bancos de dados móveis, especialmente no que se refere a objetos móveis. Foram descritos vários dos desafios de se implementar sistemas de objetos que se movem continuamente, em bancos de dados preparados para receber atualizações discretas. Neste capítulo, será proposto um modelo, o MOMENT: Modelo de Objetos em Movimento, capaz de resolver vários destes problemas. O MOMENT é um modelo facilmente extensível para agregar novas características e resolver problemas de aplicações de objetos móveis nele não resolvidos.

O capítulo está organizado da seguinte forma: a seção 3.2 mostra o diagrama de classes do MOMENT, exibindo cada classe e sua utilidade. A partir do diagrama de classes, serão explicados seus métodos e atributos e como eles são úteis para resolver os problemas citados no capítulo 2. A seção 3.3 estabelecerá um comparativo entre o MOMENT e outros modelos propostos em trabalhos realizados pela comunidade científica. A seção 3.4 traz algumas considerações finais para o capítulo.

3.2. Diagrama de Classes do MOMENT

Nesta seção será apresentado o diagrama de classes do MOMENT. Serão explicados as classes, seus atributos e métodos e de que forma serão usados para dar a funcionalidade de objetos móveis e resolver os problemas inerentes à sua implementação.

No modelo, os objetos móveis têm suas informações armazenadas na base de dados, devendo ser atualizadas através de observações a cada determinado intervalo de tempo. As consultas no MOMENT poderão ser realizadas tendo como alvo objetos estáticos, como pontos de referência, como na consulta "Selecione o objeto mais próximo de determinado local". Também é possível fazer consultas em relação a outros objetos móveis, como no caso da consulta "Em que ponto as trajetórias dos objetos A e B irão se cruzar?", em que é necessário relacionar mais de um objeto do mesmo tipo.

Devido à grande diversidade de aplicações que são possíveis de se conceber utilizando objetos móveis, é bastante desafiador elaborar um modelo totalmente genérico, capaz de atender, sem construções adicionais, a qualquer tipo de aplicação que se deseje. Por isso, o objetivo do MOMENT é atender a um tipo especifico de aplicação, mas de forma que poucas extensões sejam necessárias para adaptar o modelo a outro tipo de aplicação. Nesta fase de concepção, apenas os pontos foram considerados como dotados de mobilidade, ou seja, os objetos tidos como móveis não têm considerada a sua extensão.

Os pontos móveis devem trafegar através de vias e esporadicamente realizar mudanças de trajeto, sendo possível a mudança de via ao atingir cruzamentos. A sua posição sempre será relacionada com o instante de tempo em que foi observada, dando à base de dados características espaço-temporais. Assim, o MOMENT realiza observações da movimentação contínua dos objetos, armazenando-a de forma discreta. No caso de consultas realizadas em instantes de tempo nos quais não tenha sido realizada nenhuma observação da posição do objeto, deve-se realizar uma previsão do local onde o objeto estaria, baseada na movimentação e trajetória que o objeto vinha mantendo. Isso caracteriza o atributo que armazena a posição do objeto móvel como um atributo dinâmico.

Para melhor realizar a previsão citada acima, são criados perfis de movimentação dos objetos, armazenando velocidades que o objeto costuma manter em determinados trechos e caminhos normalmente percorridos, de forma a facilitar a tomada de decisão acerca de qual caminho o objeto teria tomado num momento em que se apresentam várias possibilidades.

O MOMENT foi desenvolvido objetivando aplicações em que os objetos móveis necessitam ser equipados apenas com um dispositivo de GPS para poder fornecer seu posicionamento a cada determinado intervalo de tempo. Assim, qualquer dispositivo móvel que possa ser dotado desta tecnologia é adequado para equipar um objeto móvel do sistema. A figura 3.1 mostra a arquitetura de um sistema baseado no modelo MOMENT.

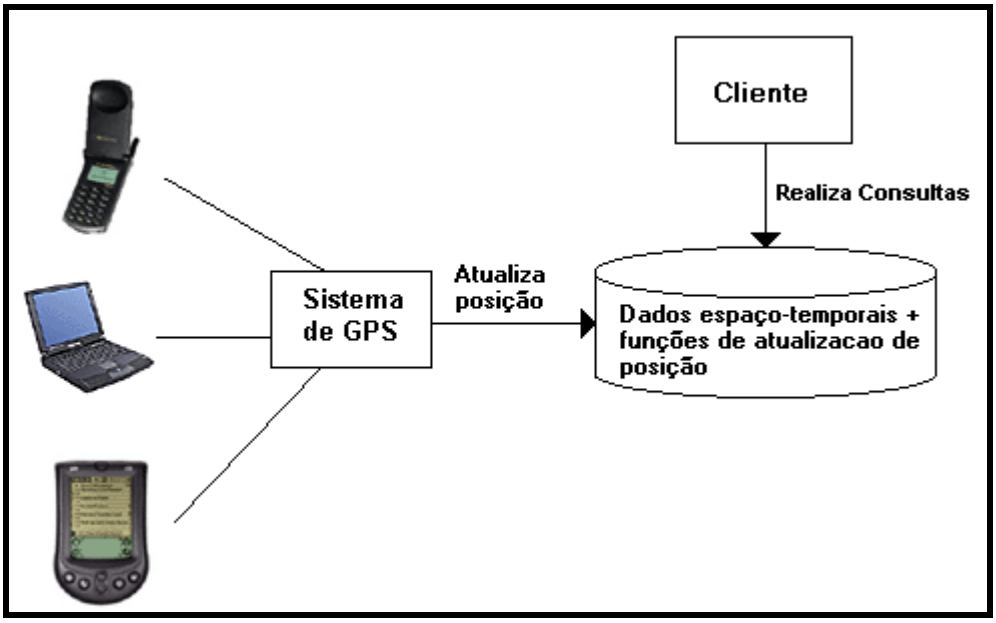

Figura 3.1. Arquitetura de um sistema baseado no MOMENT.

Como alguns dos dispositivos que podem ser usados na aplicação são dotados de baixo poder de processamento, como telefones celulares, handhelds e aparelhos de GPS puro, não é possível utilizar um tratamento de imprecisão em que o objeto móvel dispara uma atualização por conta própria. Nessa política, ao distanciar-se acima de um determinado limite da posição imaginada pelo banco de dados, o objeto antecipa a atualização de sua posição. Mas isso só é possível se o objeto é capaz de realizar o mesmo processamento que o banco realiza para calcular posições, e assim conhecer a posição que o banco considera ser a sua posição. Tendo sido idealizado para aplicações que se utilizem destes dispositivos de baixo poder de processamento, o MOMENT somente diminui a imprecisão na localização dos objetos realizando atualizações de posição baseadas na utilização de um atributo dinâmico.

A figura 3.2 mostra o modelo conceitual do MOMENT. O modelo pode ser dividido em duas partes: uma parte puramente espacial, em que as classes são úteis para representar as diferentes geometrias que podem ser usadas no modelo, e outra parte de objetos móveis, onde as classes estendem a parte espacial, fazendo atender aos requisitos de mobilidade.

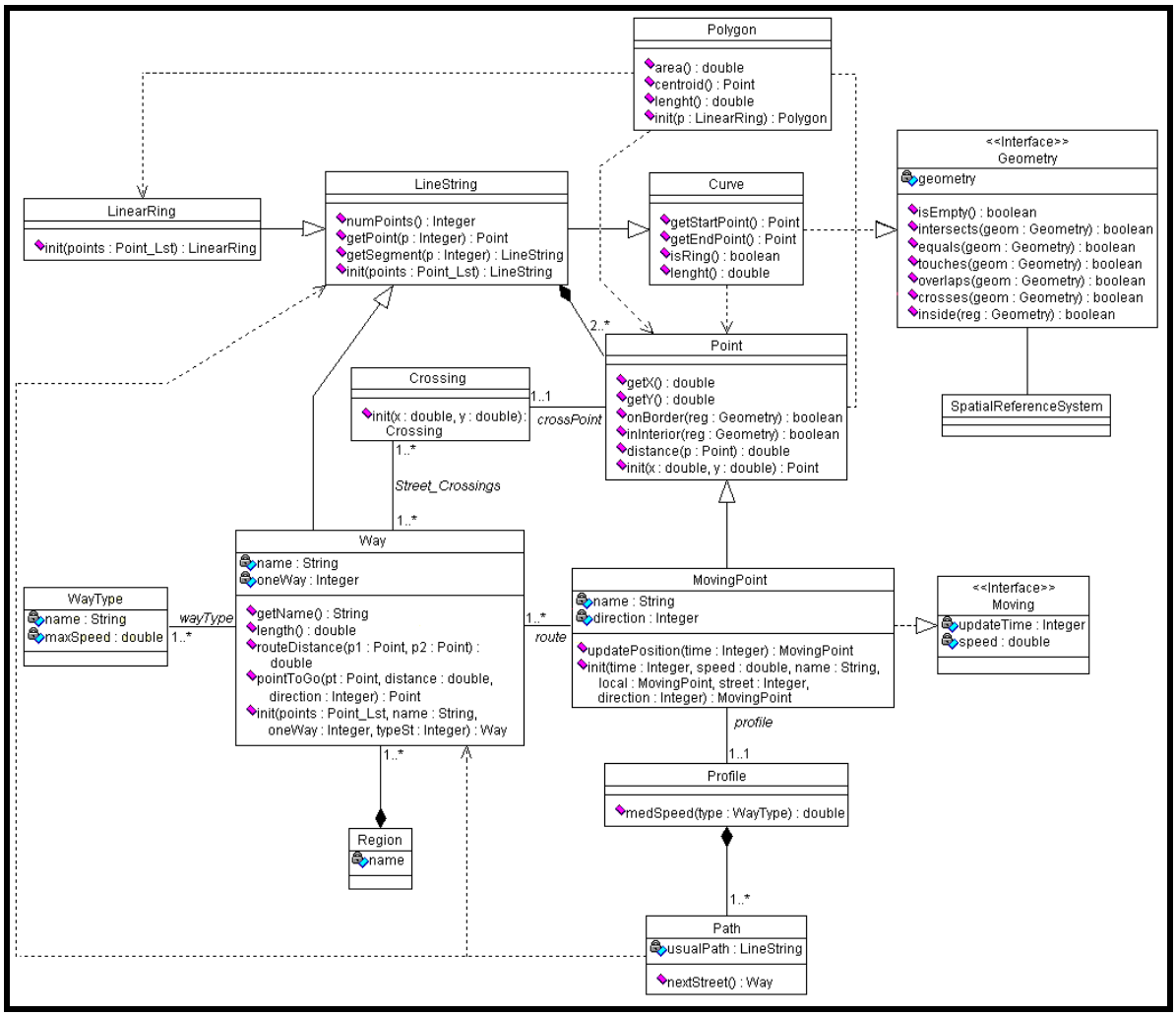

Figura 3.2. Modelo conceitual do MOMENT

A parte espacial do modelo segue o padrão de geometrias especificado pelo *Open GIS Consortium* [OGIS98, CCL01], e é composta das seguintes classes:

- **Geometry** é a base para todas as geometrias possíveis de se representar, onde uma geometria é definida como qualquer objeto espacial representável no modelo. É responsável por fazer uma ligação entre o modelo e as características espaciais da base de dados, seja através de herança de alguma classe espacial pré-existente, ou definindo o atributo *geometry* como sendo de um tipo definido para tal fim. Os métodos nela definidos (*intersects(), equals(), touches(),* etc.) são úteis para qualquer tipo de geometria, podendo ser utilizados para responder consultas estabelecendo relação entre mais de um objeto.

- **Point** é uma classe que define um ponto como sendo um tipo especial de geometria, herdando por isso as características de *Geometry*. Possui apenas dois valores (X e Y) que formam um par de coordenadas, sendo acessíveis através dos métodos *getX()* e *getY()*. Os métodos *onBorder()* e *inInterior()* são úteis para referenciar o posicionamento do ponto em relação a uma região, enquanto o método *distance()* estabelece a distância em linha reta para um outro ponto. O método *init()* é o construtor da classe, que necessita da definição das coordenadas do ponto.

- **Curve** define uma linha através de um ponto inicial e um ponto final, também herdando as definições de *Geometry*. É definido por dois pontos, um inicial e um final, que podem ser obtidos pelos métodos *getStartPoint()* e *getEndPoint()* respectivamente. O método *isRing()* verifica se o ponto inicial e o ponto final são coincidentes, sendo útil para as classes que fizerem herança de *Curve*. O método *length()* verifica o tamanho do segmento.

- **LineString** é um conjunto de segmentos de reta, que herda as definições de *Curve*. É definido por uma seqüência de pontos interligados, agregando no mínimo dois pontos, sem limite de número de pontos. O método *numPoints()* retorna quantos são os pontos formadores do objeto *LineString*, enquanto *getPoint()* retorna um determinado ponto da seqüência, respeitando a sua ordem. O método *getSegment()* pode retornar qualquer um dos segmentos que formam o *LineString*, também respeitando a ordem da seqüência. Essas funções são úteis para implementar alguns métodos das classes específicas de objetos móveis. O construtor da classe (*init()*) recebe uma lista de pontos, que formam os segmentos.

- **LinearRing** define uma seqüência de pontos que se fecha, ou seja, uma seqüência de pontos em que o ponto final é o próprio ponto inicial. Seu construtor recebe uma lista de pontos. É um tipo usado para a definição de um *Polygon*.

- **Polygon** define uma região, herdando também as definições de *Geometry*. É definido por um *LinearRing*, que forma as fronteiras do polígono. Possui os métodos *area(),* que retorna a área da região; *centroid(),* que retorna seu centróide; e *length(),* que retorna o perímetro. Seu construtor recebe um *LinearRing*.

Além da parte espacial, há no modelo a parte de objetos móveis, que é uma extensão da parte espacial feita de forma a acrescentar as funcionalidades desejadas para um modelo de objetos móveis. A seguir serão descritas as classes do diagrama e suas funcionalidades.

- **Moving** é utilizada para manter características que são úteis para qualquer geometria que venha a ser considerada móvel. Uma delas é o tempo de atualização, representado pelo atributo *updateTime*, que indica o instante de tempo em que a posição indicada foi observada, além da velocidade com que o objeto se movimenta, representada pelo atributo *speed*.

- **MovingPoint** representa um ponto móvel. Herda as características de mobilidade de *Moving* e *Point*, trazendo as características espaciais de um ponto para acrescentar a mobilidade. Além de indicar a posição, seu tempo de atualização e velocidade, indica a direção para a qual se movimenta, através do atributo *direction*. O método *updatePosition()* é responsável por calcular a posição em que o objeto deve estar após determinada quantidade de tempo decorrido desde a última atualização, evitando que uma consulta realizada no tempo atual seja respondida baseada em uma posição de uma instante anterior. O método *init()* é o construtor da classe.

  Um ponto móvel se relaciona também com a classe *Profile* (relacionamento *profile*), de forma a associar um perfil a cada ponto móvel, e com a classe *Way* (relacionamento *route*), indicando através de qual rua o objeto está transitando no momento.

  Além desta classe *MovingPoint*, outra extensão que pode ser realizada no modelo é a criação da classe *MovingPolygon*, herdando as definições das classes *Moving* e *Polygon*, concebendo uma região móvel. Porém, conforme foi exposto anteriormente, esta concepção do modelo MOMENT não irá tratar de mobilidade em regiões.

- **Way** corresponde a uma via, ou seja, ao meio por onde transitam os objetos. Herda atributos e métodos de *LineString*, sendo espacialmente também uma seqüência de pontos interligados. Possui o atributo *oneWay*, que indica se é permitido o tráfego nos dois sentidos ou não, e relaciona-se com a classe *WayType*, para indicar o tipo da via. Alguns

dos métodos fundamentais para a localização de um objeto móvel sem a realização de uma observação[2] estão nesta classe. O método *length()* retorna a extensão da via, percorrendo-se cada um dos segmentos que a constituem. O método *routeDistance()* deve retornar a distância entre dois pontos na via, não em linha reta, mas a distância obtida partindo de um ponto para o outro através dos segmentos da via. O método *pointToGo()* estabelece o ponto de destino de um objeto, partindo-se de um ponto conhecido da via e percorrendo determinada distância através dela. O método *init()* é o construtor da classe, recebendo a seqüência de pontos que forma os segmentos da via, além das propriedades da via já citadas.

- **Crossing** define um cruzamento, que é um ponto em determinada via no qual ocorre uma interceptação por uma outra via. Ao passar por um cruzamento, um objeto pode tomar a decisão de prosseguir na mesma via ou tomar a via que cruza. A classe se relaciona com *Way*, definindo as vias que passam pelo cruzamento, e *Point*, pois um cruzamento tem as características espaciais de um ponto. O método *init()* é o construtor, recebendo os valores das coordenadas do ponto.

- **WayType** define um tipo de via. Essa classe pretende armazenar características que atendem a um grupo de vias, evitando replicação de informação. O atributo *maxSpeed* determina a velocidade máxima permitida para cada tipo de via.

- **Region** é um agrupamento de vias, formando o mapa por onde os objetos devem se movimentar.

- **Profile** define o perfil do objeto móvel. É útil para tentar prever com maior precisão uma movimentação ainda não realizada de determinado ponto móvel. Deve armazenar padrões de comportamento dos pontos móveis, tais como a velocidade com que o ponto costuma se movimentar em cada tipo de via, ou caminhos que o ponto costuma realizar, tornando a previsão mais condizente com a realidade. Essas características da movimentação de um objeto móvel são obtidas através do histórico de observações. O método *medSpeed()* fornece a velocidade média com que o objeto costuma transitar naquele tipo de via.

[2] Neste trabalho, chama-se observação a uma posição que foi informada pelo objeto móvel.

- **Path** corresponde a um caminho comumente executado pelo veículo. Define uma seqüência de pontos que forma um caminho pelo qual o ponto móvel costuma passar. O atributo *usualPath* define o caminho como um objeto *LineString*, e o método *nextStreet()* auxilia no processo de decisão de que rua seguir, baseado nos caminhos armazenados. Esse método deve ser utilizado no processo de tomada de decisão ao atingir um cruzamento, no caso da realização de uma previsão de movimentação.

O modelo MOMENT foi concebido para ser uma extensão da funcionalidade espacial provida pelo SGBD sobre o qual será implementado. No caso do SGBD utilizado não dispor de característica espacial, deve-se implementar essa funcionalidade em uma classe a ser herdada pela classe *Geometry*.

3.3. O MOMENT e Outros Modelos Propostos

- MOST

O modelo de dados MOST – *Moving Objects Spatio-Temporal* [SWC+97, SWC+99, WXC+98, WSX+99] tem como principal característica o uso de um atributo dinâmico para representar a posição do objeto móvel. Este atributo calcula a posição atual do objeto no espaço em função do tempo, tornando o modelo espaço-temporal. O cálculo da posição é realizado baseado na função de movimentação do objeto, que representa o movimento que vinha sendo realizado pelo objeto no momento da última atualização da base.

O modelo é capaz de responder às consultas sobre os objetos móveis nos tempos passado, presente e futuro, pois além de armazenar as posições ocupadas no passado é possível realizar previsões baseadas na função de movimentação e imaginar uma posição que virá a ser ocupada no futuro.

O MOST não traz um conjunto completo de tipos de dados para o desenvolvimento de sistemas de objetos móveis, mas preocupa-se com soluções para problemas encontrados para se implementar um sistema desse tipo, como o atributo dinâmico, usado para manter as posições dos objetos móveis atualizadas em momentos nos quais não houve realização de observações. Além disso, desenvolve uma linguagem para acesso a dados espaço-temporais, a

FTL, alegando que linguagens tradicionais de acesso a dados como SQL não são preparadas para consulta a dados de objetos móveis.

- Modelo de Vazirgiannis e Wolfson

O modelo desenvolvido por Vazirgiannis e Wolfson [VW01] é específico para sistemas de veículos transitando pelas ruas de uma cidade. Assim sendo, considera apenas pontos móveis, que formam uma trajetória representada por segmentos de reta. Estes segmentos formam um grafo que compõe o mapa através do qual os objetos se movimentam.

O modelo sugere alguns atributos e funções a serem construídos de forma a dar suporte ao deslocamento dos objetos e realização de previsões de localização para tempos futuros e instantes de tempo não observados. O atributo *Drive Time* sugere um tempo médio que se leva para percorrer determinado segmento do mapa, enquanto *T(O)* representa a trajetória do objeto *O*, entre outras funções. Ao presumir o conhecimento da trajetória do objeto e do tempo que o veículo leva para percorrer cada segmento da mesma o modelo é prejudicado, pois conta com situações irreais e dados que dificilmente estarão disponíveis. Mas é um modelo que apresenta praticabilidade de implementação, por apresentar um conjunto enxuto de tipos de dados bem definidos. Define também uma série de predicados que estendem a linguagem SQL para trabalhar com dados espaço-temporais, ao invés de trabalhar com a linguagem SQL padrão.

- CHOROCHRONOS

O projeto CHOROCHRONOS [EGS+99, GBE+00, FGN+00] define uma série de tipos de dados, espaciais ou não, que podem ser mapeados para o domínio do tempo através de um construtor chamado *moving*. Com isso são gerados os tipos de dados espaço-temporais utilizados para os objetos móveis, representados pelo próprio nome do tipo de dados original iniciado pela letra M, como, por exemplo, o tipo *Mpoint*, que representa um ponto móvel.

Além da mobilidade dos pontos, o projeto considera a mobilidade também em regiões, que além do deslocamento das mesmas abrange o seu crescimento e encolhimento. É definida também uma série de funções para os tipos de dados gerados, de forma a prover o relacionamento entre as diversas entidades móveis.

Uma vantagem do modelo é que seus tipos de dados definem operações que podem ser utilizadas em linguagem comum de acesso a dados, ou seja, a própria linguagem SQL é apropriada para gerar consultas neste modelo. O modelo é, porém, excessivamente abstrato, além de definir um número tão grande de tipos de dados e operações que torna sua implementação de um alto grau de dificuldade.

- Modelo de Beard e Palancioglu

O trabalho de Beard e Palancioglu [BP00] não define um modelo completo de dados para objetos móveis, mas sim preocupa-se com a resolução de alguns dos problemas naturais de objetos móveis e sugestões para melhoria de desempenho. Trabalha também com mobilidade em regiões, considerando além do deslocamento, crescimento e encolhimento de regiões, o giro em torno do seu próprio eixo como movimentação.

É sugerido o armazenamento de histórico de movimentações não só como forma de realizar consultas em tempo passado, mas como forma de definir padrões de movimentação que possam se úteis para previsão de localização em tempo futuro. Além do histórico, sugere a construção de um perfil de movimentação, que aumenta a precisão das previsões de movimentação baseado tanto em restrições de movimentação do objeto como nos padrões detectados em seu histórico de movimentações.

- Bei Yi e Cláudia Bauzer Medeiros

O modelo idealizado por Bei Yi e Cláudia Bauzer Medeiros [YM02] leva em consideração que as trajetórias dos objetos móveis podem ser tomadas como geometrias, e dessa forma, serem tratadas como elementos puramente espaciais. Assim, a trajetória de um ponto móvel pode ser vista como uma linha, a de uma linha móvel como um polígono e a de um polígono móvel como um poliedro, mapeado para um polígono no espaço 2D.

O modelo define uma série de classes para representação dos objetos e outras para representação de suas trajetórias. Define também uma série de operadores para relacionar os diferentes tipos de objeto. Porém, o grande número de tipos de dados e alto grau de abstração tornam o modelo de difícil implementação. O trabalho nada fala sobre linguagem para acesso aos dados espaço-temporais.

A tabela 3.1 estabelece um comparativo entre os modelos estudados e o MOMENT, apresentando as características de cada um.

Modelo / Característica	MOST	Vazirgiannis	CHORO-CHRONOS	Beard	Yi	MOMENT
Banco de dados espaço-temporal	√	√	√	√	√	√
Histórico de movimentações	√	√	√	√	√	√
Conjunto enxuto de tipos de dados	X	√	X	X	X	√
Viabilidade de implementação	X	√	X	X	X	√
Aspectos de mobilidade em regiões	X	X	√	√	√	X
Aspectos de mobilidade no eixo do objeto	X	X	X	√	√	X
Não necessita conhecimento prévio de trajetória	√	X	√	√	√	√
Consultas em linguagens padrão de banco de dados	X	X	√	X	X	√

Tabela 3.1. Comparativo entre os modelos de objetos móveis

3.4. Considerações Finais

Este capítulo tratou do MOMENT, modelo concebido para resolver problemas que surgem ao se desenvolver uma aplicação de objetos móveis. Foi exibido o diagrama de classes do MOMENT, descrevendo cada classe com seus métodos e atributos e como são usadas para resolver cada um dos problemas. Por fim, é realizada uma comparação do MOMENT com outros modelos, mostrando qualidades e deficiências de cada um.

Capítulo 4. Um exemplo de implementação do MOMENT

4.1. Introdução

Neste capítulo serão abordados os aspectos de implementação de um sistema baseado no MOMENT. A implementação deste sistema serve para validar o MOMENT como um modelo funcional para o desenvolvimento de sistemas que tratem de objetos em movimento. O sistema constitui-se de uma série de funções a serem implementadas aproveitando-se das funcionalidades objeto-relacional e espacial de um SGBD, bem como de um simulador destinado a produzir o efeito da movimentação de objetos reais.

A aplicação consiste de um mapa de ruas de uma cidade, pela qual transita uma série de veículos. As ruas correspondem às vias do MOMENT e os veículos representam os objetos móveis, sendo considerados como pontos móveis, o que quer dizer que eles têm a sua dimensão desprezada. Os veículos informam sua posição à base de dados a cada período de tempo pré-determinado, sendo utilizado para isso um simulador de movimentação.

A seção 4.2 irá abordar os requisitos funcionais e não funcionais do sistema, além de explicar a utilidade do simulador. A seção 4.3 aborda as adaptações feitas no MOMENT para o sistema ser implementado, bem como as questões de desenvolvimento e funcionamento do mesmo, enquanto a seção 4.4 aborda os mesmos aspectos para o simulador. A seção 4.5 mostra um estudo de caso realizado sobre o sistema, com respostas a consultas estabelecidas e cálculos para demonstrar a validade das respostas, enquanto a seção 4.6 realiza algumas considerações finais.

4.2. Análise de Requisitos

Nesta seção serão apresentados os requisitos estabelecidos para a implementação de um sistema utilizando o modelo de dados MOMENT. Estes requisitos são classificados como funcionais e não funcionais, e serão divididos entre requisitos de implementação do simulador e do sistema em si.

4.2.1. Requisitos do Sistema

Requisitos Funcionais

Dispor de dados que representem veículos movendo-se por um mapa de ruas

Como se trata de um sistema de objetos móveis que considera apenas pontos como geometrias capazes de se movimentar, representar veículos transitando pelas ruas de uma cidade é um bom exemplo. Deve ser utilizado um pequeno mapa, com uma série de ruas que se cruzam, sendo possível definir geometrias como linhas e regiões e possibilitar aos veículos a movimentação por sobre este mapa. A base de dados deve armazenar informações que possibilitem à aplicação conhecer as ruas por onde os veículos se movimentam e o posicionamento de cada uma das geometrias.

A definição dessas geometrias serve para a realização de consultas espaciais envolvendo os veículos e suas relações com estes objetos geográficos, como por exemplo, a presença de um veículo em determinada rua ou região.

Permitir atualização do posicionamento dos veículos

Como o sistema representa veículos em constante movimentação, deve ser permitido que entidades externas atualizem as informações de localidade dos veículos, de forma a manter a base de dados coerente com essa movimentação. Devido à impossibilidade de dispor de equipamentos de GPS para equipar veículos reais, que transmitiriam suas posições em tempo real para a base de dados, além da dificuldade de realizar testes no sistema operando

desta forma, é necessária a construção de um simulador. Ficará por sua conta a atualização das posições dos objetos dispostos no mapa constituído a partir da base de dados.

Responder a consultas dependentes de espaço e tempo

Esta é a principal funcionalidade desejada em um sistema de objetos móveis. A realização de consultas espaço-temporais é o interesse primário de um usuário de sistemas desse tipo. Com o objeto se movimentando, deseja-se consultar seu posicionamento em determinados momentos, além de relacionar seu posicionamento e tempo com o de outros objetos referenciados.

Armazenar posições anteriores dos veículos para responder a consultas em tempo passado

Além da realização de consultas espaço-temporais em tempo presente, deseja-se também para o sistema a possibilidade de consultar posições anteriores dos veículos. Desta forma, a trajetória do veículo deve ser mantida. A cada atualização de posição de determinado veículo, sua posição anterior deve ser armazenada sem sobrescrever as demais posições já armazenadas. Para conhecer detalhes sobre as passagens anteriores de um veículo, deve-se consultar a sua posição no instante de tempo apropriado.

Fazer previsões de movimentação para consultas em tempo futuro

Outra funcionalidade desejada é a realização de previsões das futuras posições dos veículos para realização de consultas em tempo futuro. Para isso, cálculos devem ser feitos baseados no padrão de movimentação do veículo, para que a estimativa tenha um nível aceitável de precisão.

Não se pode exigir exatidão de uma consulta de posicionamento em tempo futuro, uma vez que veículos reais podem mudar de direção e padrão de movimentação a qualquer momento, sem que haja indicativos para se prever essa mudança.

Armazenar perfis de movimentações de usuário

O sistema deve armazenar perfis de movimentação, que são úteis para dar uma maior exatidão nas previsões de posicionamento realizadas em consultas de tempo futuro. Um perfil deve conter informações como velocidade que o veículo costuma desenvolver para cada tipo de rua e alguns roteiros de caminhos que o veículo executa com maior freqüência. Assim, as previsões realizadas para movimentação futura podem levar em conta essas informações para poder prever mudanças de direção e de velocidade, o que eleva a taxa de acerto em consultas de tempo futuro.

Requisitos Não Funcionais

Ter toda a funcionalidade do sistema dentro da base de dados

Todas as funções necessárias para que o sistema seja capaz de localizar objetos através do tempo, atualizar posições e prever novos posicionamentos devem ser construídas dentro da base de dados, utilizando se da capacidade do modelo Objeto-Relacional. Isso irá possibilitar que clientes leves, não construídos especialmente para acesso ao sistema possam realizar consultas, pois todo o processamento ficará a cargo do SGBD.

Possibilitar construção de clientes de diversos tipos

Com o sistema todo dentro da base de dados, qualquer cliente capaz de formular uma consulta compreendida pelo SGBD, seja SQL padrão ou uma linguagem própria do SGBD, deve ser capaz de realizar consultas no sistema. Assim, tanto pode haver clientes desenvolvidos exclusivamente para acessar a aplicação, como clientes já existentes ou mesmo browsers, capazes de submeter consultas na linguagem necessária.

4.2.2. O Simulador

A funcionalidade deste simulador é basicamente consultar na base de dados os valores referentes à última posição do veículo registrada e o tempo ao qual aquela posição correspondia. Estes valores serão atualizados pelo simulador, considerando a velocidade do veículo, o tempo decorrido e a possibilidade de realizar desvios aleatórios na trajetória, entrando em outras ruas. Desta forma será buscada uma movimentação não uniforme dos veículos, semelhante à movimentação de veículos reais.

Como o simulador opera sobre os objetos instanciados no sistema, tanto a construção do mapa quanto o posicionamento das geometrias deve ser baseado em informações colhidas na base de dados. O simulador deve construir internamente o mapa, de forma a simular a movimentação dos veículos por sobre estas ruas.

As posições dos veículos produzidas pelo simulador devem ter como destino a base de dados. Conforme descrito no capítulo anterior, para se descrever uma movimentação contínua utiliza-se uma seqüência de atualizações discretas [EGS+99]. Assim, as posições devem ser geradas pelo simulador a cada intervalo de tempo determinado, sendo sempre transmitidas para a base de dados, onde possibilitarão que as consultas sejam feitas sobre os dados mais atualizados. Quanto menor o intervalo de tempo entre as atualizações, mais precisos serão os resultados das consultas.

4.3. Implementação do Sistema

Esta seção visa explicar aspectos de implementação de um protótipo baseado no modelo MOMENT. O sistema foi desenvolvido usando o SGBD Oracle9i, escolhido por ser objeto relacional, o que mantém as características do modelo, e por possuir uma excelente funcionalidade espacial, que foi bastante aproveitada no desenvolvimento do sistema. Na base de dados ficam armazenados, além dos dados, as classes desenvolvidas, na forma de tipos de objeto. Assim, toda a funcionalidade do sistema fica dentro da base de dados, possibilitando que os clientes sejam leves e sejam dotados apenas da capacidade de realizar consultas especialmente elaboradas para o acesso aos dados de objetos móveis. O sistema foi

desenvolvido em PL-SQL, linguagem para desenvolvimento procedimentos armazenados (*stored procedures*) do Oracle9i.

O sistema foi definido para trabalhar com um tipo específico de aplicação de objetos móveis: veículos transitando pelas ruas de uma cidade. Assim, os veículos fazem o papel dos objetos móveis, todos eles sendo pontos móveis, como é estabelecido pelo modelo, e as ruas serão as vias por onde eles trafegam.

A seguir, serão descritos os tipos de dados gerados na implementação do sistema.

- **Geometry_Typ** – equivale à interface *Geometry*, do MOMENT.

  As características espaciais do sistema são implementadas utilizando um tipo de dados pré-definido do Oracle9i: o *MDSYS.SDO_GEOMETRY*[3]. Este tipo de dados é utilizado para representar todos os tipos de geometria que o Oracle9i dá suporte. Por isso, o atributo *geometry* de *Geometry_Typ* é estabelecido como sendo do tipo *MDSYS.SDO_GEOMETRY*, sendo utilizado para representar qualquer tipo de geometria, uma vez que todas as geometrias são definidas através de herança do tipo *Geometry_Typ*. Este tipo é definido como "*not instantiable*", pois este tipo de dados não deve ser instanciado, apenas serve para ser herdado pelas classes de geometrias específicas, e "*not final*", pois outras classes herdarão suas características.

  As funções implementadas em *Geometry_Typ* são aplicáveis a qualquer tipo de geometria, e são implementadas utilizando recursos disponíveis no tipo de dados *MDSYS.SDO_GEOMETRY* para este fim. Funções como *intersects()*, *touches()* e *overlaps()* são calculadas através de recursos deste tipo de dados, e somente são mantidas como funções à parte para manter as definições do MOMENT.

- **Point_Typ** – equivale à classe *Point*, no MOMENT. As coordenadas necessárias para a definição de um ponto são estabelecidas no atributo *geometry*, herdado de *Geometry_Typ*. As funções *getX()* e *getY()* podem ser utilizadas para acessar cada coordenada individualmente, e são implementadas através de recursos de *MDSYS.SDO_GEOMETRY*, que pode acessar cada uma das ordenadas que compõem uma geometria individualmente. As funções *onBorder()* e *inInterior()* são definidas para utilização em consultas que

[3] O funcionamento do tipo de dados *MDSYS.SDO_GEOMETRY* é descrito no anexo I deste trabalho.

envolvam relação espacial entre um ponto e uma região no mapa, e são implementadas também usando recursos de *MDSYS.SDO_GEOMETRY*.

A função *init()* corresponde ao construtor da classe, não tendo sido este implementado com o próprio nome da classe, como é padrão no desenvolvimento de aplicações orientadas a objeto, devido ao Oracle9i em sua versão 9.0.1 não permitir o desenvolvimento de construtores por parte do usuário.

- **Point_Lst** – é um array de elementos do tipo *Point_Typ*, usado para definir outros tipos de dados.

- **Curve_Typ** – corresponde à classe *Curve*, no MOMENT. Como define um segmento de reta, armazena quatro coordenadas: duas para o ponto inicial e duas para o ponto final. As coordenadas que formam estes pontos podem ser acessadas através dos métodos *getStartPoint()* e *getEndPoint()*, que retornam objetos da classe *Point_Typ* formados com estas coordenadas.

  O método *isRing()* verifica se o primeiro ponto coincide com o último, formando um polígono fechado, ou um anel. É definido para ser usado por classes que herdam as definições de *Curve_Typ*. O método *length()* calcula o tamanho do segmento, e é definido através de um recurso de *MDSYS.SDO_GEOMETRY*. A classe é definida também como "*not instantiable*" e "*not final*".

- **LineString_Typ** – equivale à classe *LineString* no MOMENT. Como representa uma série de segmentos de reta, um objeto desta classe é definido por uma série de pontos que, ligados, formam os segmentos de reta. Assim, o atributo *geometry* armazena uma série de coordenadas que constituem os seus pontos.

O método *numPoints()* conta o número de coordenadas armazenadas de forma a determinar quantos pontos compõem o objeto *LineString_Typ*. O método *getPoint(p)* determina um dos pontos que compõem a geometria, a partir de um índice *p* que determina a posição do ponto, enquanto *getSegment(p)* retorna um dos segmentos que a compõem, também a partir de um índice *p* para determinar a posição do segmento em relação aos demais.

O construtor *init()* deve receber a lista de pontos que formam o objeto; essa lista é passada como um objeto *Point_Lst*. O tipo *LineString_Typ* é definido como "*not final*"

- **LinearRing_Typ** – corresponde à classe *LinearRing* no MOMENT. Representa um polígono fechado, que é passado ao construtor do tipo *Region_Typ*, definindo as fronteiras de uma região. Seu único método é o construtor, *init()*, que recebe também uma lista de pontos *Point_Lst*, mas verifica se os segmentos de reta se fecham.

- **Region_Typ** – equivale à classe *Polygon* do MOMENT. Representa um polígono, que pode delimitar uma região no mapa, sendo definido por um *LinearRing_Typ*. Os métodos *area()*, *centroid()* e *length()* utilizam recursos de *MDSYS.SDO_GEOMETRY* para determinar a área, centróide e perímetro da região, respectivamente. Seu construtor *init()* deve receber um objeto *LinearRing_Typ* que define as fronteiras da região.

- **Moving_Typ** – equivale à interface *Moving* do MOMENT. Uma geometria que herda as características deste tipo de dados passa de estática a móvel. Seus atributos *updateTime* e *speed* são típicos para qualquer geometria que venha a ser considerada móvel, e correspondem ao instante de tempo em que o objeto teve suas características espaciais atualizadas e a velocidade que ele desenvolvia no mesmo instante. É um tipo de dados também declarado como "*not instantiable*" e "*not final*".

- **MovingPoint_Typ** – corresponde à classe *MovingPoint* no MOMENT. Representa os veículos que transitam pelas ruas, ou seja, são os objetos móveis do sistema. Possui o atributo *updateValue*, definido como sendo do tipo *Point_Typ*, que representa a localização do veículo no espaço, e herda as características de *Moving_Typ*, fazendo com que a classe *MovingPoint_Typ* possua as características espaço-temporais necessárias para objetos móveis. O atributo *updateValue* refere-se à localização do veículo no instante de tempo registrado em *updateTime*, herdado de *Moving_Typ*. O atributo *name* designa um nome ao veículo, enquanto *route* referencia a rua pela qual o veículo se move, e *direction* indica a direção na qual o veículo se movimenta pela rua. O valor 0 indica movimentação do fim para o início da rua, enquanto 1 indica movimentação do início para o fim da rua.

O atributo simTime é utilizado apenas para manter a compatibilidade de tempo com o simulador (será explicado na seção 4.4).

O método *updatePosition()* é usado sempre que houver uma defasagem do momento em que a posição do veículo foi atualizada para o momento em que a consulta é realizada. Ou seja, se já decorreram alguns segundos desde a última atualização da posição, a movimentação contínua do objeto fará com que esta posição esteja defasada no momento de realização da consulta. Assim, *updatePosition()* deve simular a movimentação ocorrida durante aqueles segundos e propor a nova posição do veículo, o que fará com que a imprecisão no resultado da consulta seja reduzida, na maioria dos casos. É usado também para realizar previsão de posicionamento em tempos futuros.

Este método verifica quanto espaço deve ser percorrido durante aquele tempo, mantendo-se a velocidade registrada anteriormente pelo veículo, e calcula o posicionamento que deve ser alcançado ao se percorrer aquele espaço pela rua. Esse cálculo é feito considerando-se a distância entre os pontos extremos de cada segmento da rua, e destes pontos para o ponto onde se localiza o veículo. Somando-se cada um dos trechos compõe-se o espaço a ser percorrido. No caso da rua acabar e ainda haver espaço a ser percorrido, continua-se a movimentação pela rua adjacente, virando-se o veículo arbitrariamente em alguma direção, uma vez que não há subsídios para determinar que direção o veículo tomará.

O construtor da classe, *init()*, recebe os valores de todos os seus atributos, incluindo o ponto de sua localização inicial, que é um objeto do tipo *Point_Typ*.

- **Street_Typ** – corresponde à classe *Way* no MOMENT. Representa as ruas por onde transitam os veículos. Espacialmente, é constituído de um *LineString_Typ*, que perfaz o desenho da rua, sendo então uma herança deste tipo de dados. Seus atributos são *name*, que atribui um nome à rua, *oneWay*, que indica se a rua possui ou não mão dupla (0 indica mão única, e 1 indica mão dupla), e s*treetType*, que referencia o tipo da rua.

O método *getName()* retorna o nome da rua, enquanto *length()* retorna a extensão da rua, somando a extensão de cada um dos seus segmentos. O método *routeDistance()* retorna a distância existente entre dois pontos da rua caso se percorra o caminho entre eles através da rua. Para realizar este cálculo, soma-se a distância entre o ponto de origem e o próximo ponto que determina um segmento, o comprimento de cada segmento entre os pontos, e a distância entre o fim do último segmento entre os pontos e o ponto de destino.

O método *pointToGo()* determina o ponto da rua em que deve parar um veículo, considerando-se os seguintes dados: o ponto em que o veículo se encontra, a quantidade de metros a serem percorridos e a direção em que o veículo se move. Para realizar o cálculo de onde o veículo irá parar, subtrai-se da quantidade de metros a ser percorrida a extensão de cada segmento já percorrido. Quando o tamanho do próximo segmento for maior do que a quantidade restante de espaço a percorrer, calcula-se o ponto interno do segmento onde o veículo deve parar. Caso a rua acabe antes de se percorrer todo o espaço especificado, o método retorna a quantidade de espaço remanescente, além do ponto limite da rua como posição alcançada. O método *simplePointToGo()* tem funcionamento semelhante, mas não retorna o espaço remanescente caso acabe a rua, simplesmente assume o ponto limite da rua como destino final.

O construtor da classe, *init()*, recebe os pontos que formam a rua, na forma de um objeto *Point_Lst*, além dos valores dos demais atributos.

- **Crossing_Typ** – equivale à classe *Crossing* no MOMENT. Representa os pontos em que ocorrem intersecções entre ruas, possibilitando ao veículo mudar a pista em que transita. Possui o atributo *crossPoint*, que localiza o ponto onde ocorre o cruzamento, e relaciona-se com o tipo *Street_Typ*.

- **StreetType_Typ** – equivale à classe *WayType* no MOMENT. Sua utilidade é possibilitar a alguns atributos estarem associados a um grupo inteiro de ruas, ao invés de cada rua ter seu valor replicado. No caso, utiliza-se o atributo *name*, para dar nome ao tipo de rua, e *maxSpeed*, para definir a velocidade máxima para aquele tipo de rua.

Nesta versão do sistema não foi implementada a funcionalidade do perfil de movimentação. Por isso, não há equivalentes para as classes *Profile* e *Path*.

Além dos tipos de dados, uma série de tabelas foi construída, de forma a armazenar todas as informações acerca dos objetos instanciados. As tabelas são as seguintes:

- **City** – armazena as informações das ruas que constituem o mapa por onde transitam os veículos.

- **Crossings** – armazena os pontos onde ocorrem cruzamentos.

- **Street_Crossings** – associa os cruzamentos à rua a que pertencem e a rua transversal.

- **StreetType** – armazena os tipos de rua e seus atributos.

- **MovingPoints** – armazena as informações acerca da última observação do posicionamento dos objetos móveis.

- **MovingPointsHist** – armazena todo o histórico de movimentação dos objetos móveis, ou seja, as observações anteriores à última realizada.

4.4. Implementação do Simulador

O simulador é utilizado para realizar atualizações das posições dos objetos móveis na base de dados. Antes de o simulador ser construído, foi testado um outro simulador, o *City Simulator V 2.0*, da IBM [KMJ01]. Este, porém, não foi adequado para os propósitos do trabalho, uma vez que simula movimentos desordenados e sem padrão, como seriam os de pessoas se movimentando. Para este trabalho era desejado um simulador que gerasse movimentações obedecendo a um certo padrão de movimentação, como seria a movimentação de um veículo real. Nessas movimentações, era preciso que o veículo se movimentasse por sobre a rua, e não era interessante, por exemplo, que o veículo mudasse de direção em uma mesma rua. Decidiu-se então pela construção de um simulador.

O simulador foi implementado em linguagem Java, com acesso aos dados no Oracle9i via JDBC, que é um conjunto de classes e interfaces que constituem uma API própria para acessar dados em tabelas constantes de SGBD's existentes no mercado.

O papel do simulador é ler as informações sobre os veículos e sobre o mapa das ruas armazenadas nas tabelas citadas na seção sobre o sistema, fazer cálculos de atualização das posições dos objetos utilizando os próprios métodos dos objetos armazenados dentro do Oracle9i e devolvê-los atualizados para a base de dados, juntamente com os instantes de atualização. Além disso, deve preservar as informações anteriores na tabela de histórico das movimentações dos objetos.

Os instantes de tempo no simulador são contados em segundos desde a meia-noite da data 1 de janeiro de 1970. por isso, para facilitar o tratamento do tempo dentro do simulador, além do instante de atualização real da classe *MovingPoint_Typ*, foi mantido armazenado na base também o atributo *simTime*, que armazena o tempo de atualização no mesmo formato que o simulador.

A figura 4.1 mostra o diagrama de classes do simulador.

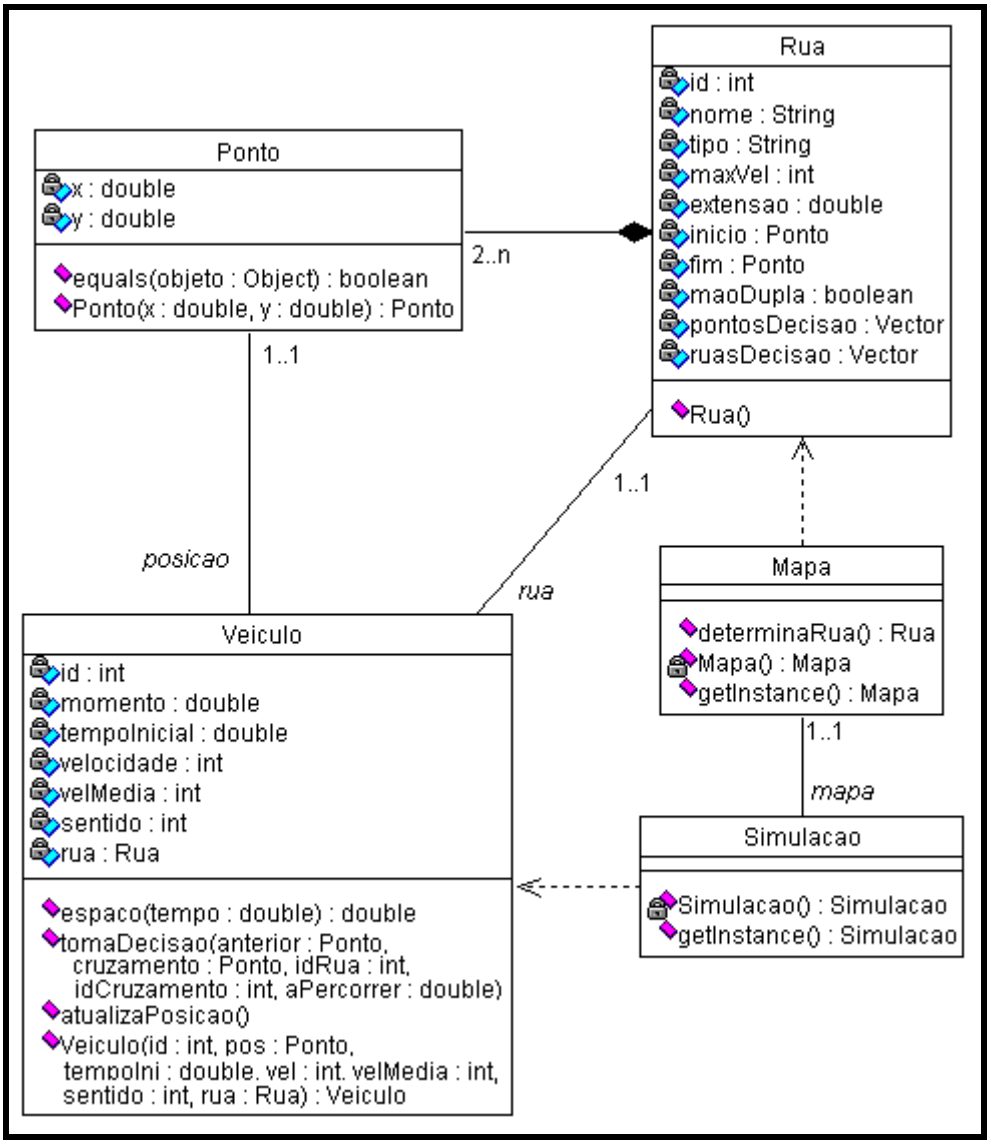

Figura 4.1. Diagrama de classes do simulador

As classes que constituem o simulador serão descritas a seguir:

- **Ponto** – classe que armazena as coordenadas *x* e *y* de um ponto como atributos. Possui também um método para verificar a igualdade de dois pontos.

- **Rua** – armazena as definições de uma rua como atributos.

- **Mapa** – classe *singleton*, ou seja, que pode ser instanciada uma única vez, que lê as informações sobre as ruas na base de dados e constitui dentro do simulador o mapa por onde os veículos podem transitar. Possui o método *determinaRua()*, que dada a chave da rua retorna as informações referentes à rua solicitada.

- **Veículo** – classe que contém as definições dos objetos móveis. Possui os métodos *espaco()*, que calcula quanto espaço deve ser percorrido baseado na velocidade e no tempo; *atualizaPosicao()*, que é disparado sempre que o contador de tempo atinge o intervalo entre atualizações e calcula a nova posição do objeto; e *tomaDecisao()*, que é disparado dentro do método *atualizaPosicao()* toda vez que um veículo alcança um cruzamento. Neste método, o simulador escolhe randomicamente se o veículo deve permanecer em seu curso na rua que já vinha seguindo ou se deve tomar a outra rua, e em que sentido seguirá na outra rua.

- **Simulacao** – outra classe *singleton*, que executa a simulação, instanciando todas as classes necessárias e disparando seus métodos. Possui um único método, *simula()*, que conta o tempo, realiza as leituras na base de dados, dispara as atualizações e escreve as informações atualizadas na base de dados.

- **Simula** – a classe que contém o método *main()*, o principal método de um programa Java, que dispara o funcionamento de todo o programa.

Todas estas classes estão contidas dentro de um *package*, um pacote que contém classes do Java, chamado Simulador. Além deste, foi utilizado um outro *package*, BancodeDados, que continha classes responsáveis pela conexão da aplicação com o banco de dados utilizando JDBC.

4.5. Estudo de Caso

Para demonstrar a funcionalidade do sistema MOMENT, foi desenvolvido o estudo de caso que será detalhado a seguir. Foi criado um mapa fictício, inserindo-se uma série de informações acerca de ruas imaginárias na tabela *City*. As ruas formariam o mapa mostrado na figura 4.2. Além da tabela *City*, para constituição do mapa foram inseridos dados também na tabela *StreetType*, detalhando os tipos de rua aos quais pertencem as ruas criadas; na tabela *Crossings*, indicando os pontos no mapa em que ocorrem os cruzamentos de ruas; e na tabela *Street_Crossings*, que estabelece o relacionamento entre os cruzamentos e as ruas que por eles passam.

Além das ruas, foram criados também alguns objetos móveis (veículos), armazenados na tabela *MovingPoints*. O posicionamento inicial destes veículos é mostrado também na figura 4.2. Além dos dados, também foi definido para o estudo de caso um tempo de 15 segundos para que o simulador forneça uma atualização de posicionamento dos objetos.

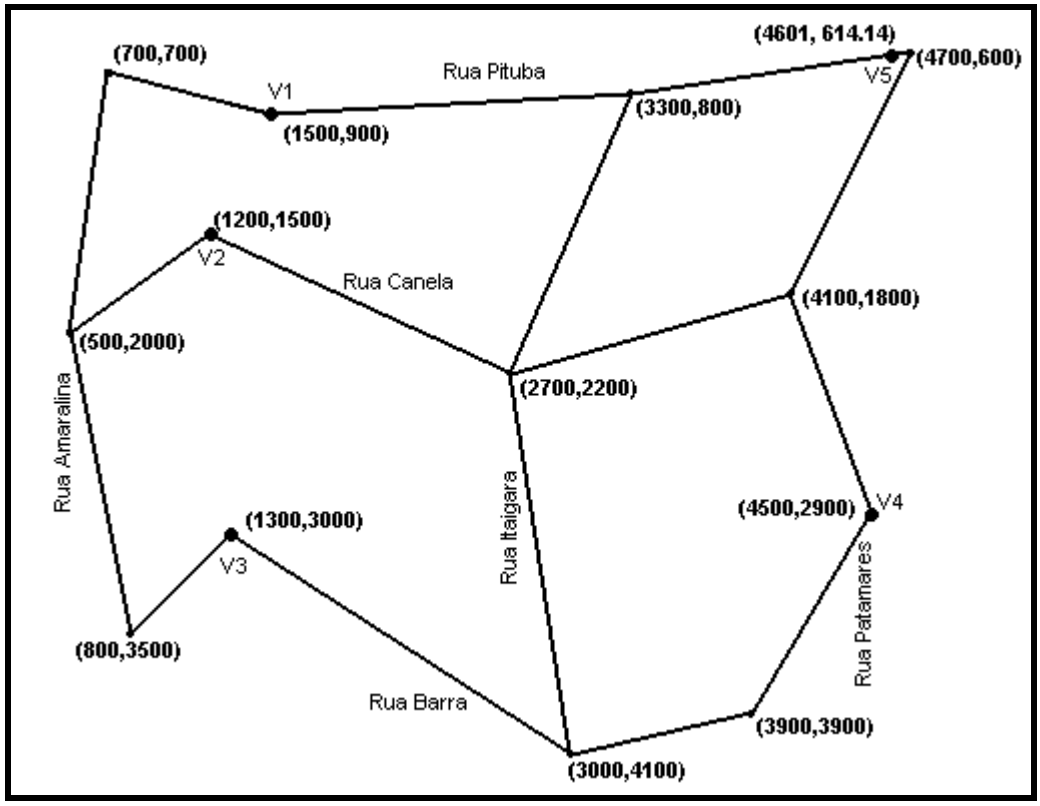

Figura 4.2. Posicionamento inicial dos veículos no mapa

Foram criadas algumas consultas de comum interesse em um sistema de objetos móveis, para serem executadas em diferentes instantes de tempo e terem seus resultados detalhados nesta seção. Estas consultas são:

1. Em quanto tempo o veículo 1 chegará ao ponto (3300,800)?
2. Onde estava o veículo 1 às 14:02:20?
3. Onde estará o veículo 1 às 14:05:00?
4. Qual veículo está mais próximo do ponto (1300,850)?
5. Em que momento ocorrerá a interseção das trajetórias dos veículos 1 e 2?
6. Em que ponto ocorrerá a interseção das trajetórias dos veículos 1 e 2?

Os cálculos necessários para demonstrar as respostas às consultas serão aqui realizados, como forma de validar os resultados alcançados. Porém, será utilizada uma precisão de duas casas decimais, enquanto o sistema utiliza-se de uma precisão maior, fazendo com que os resultados encontrados sejam aproximados em relação aos resultados retornados pelo sistema.

Consulta 1: Em quanto tempo o veículo 1 chegará ao ponto (3300,800)?

A consulta 1 busca saber em quanto tempo um objeto deve atingir determinado posicionamento. Para tanto, considera-se que o objeto vem percorrendo a rua onde o ponto desejado se localiza e faz uma trajetória que vai e direção ao ponto. No exemplo, o objeto V1 parte do ponto P0 (1500;900) em direção ao ponto (3300;800), alvo da consulta. Supondo-se que a consulta tenha sido realizada após 10 segundos da última atualização do objeto, sua posição para efeito da consulta precisa ser atualizada para refletir este tempo decorrido.

```
Procedimento PL/SQL concluído com sucesso.
     TEMPO
----------
51,1665383
```

Figura 4.3. Resultado da consulta 1

Para comprovar o resultado da consulta, serão exibidos os cálculos realizados e o resultado será comparado. Primeiramente, deve-se atualizar a posição do objeto, levando em consideração que o objeto já se moveu após a última observação de posicionamento. Para saber quanto tempo o veículo se moveu, utiliza-se:

```
select ((sysdate - mp.object.updateTime) * 86400) into tDecorrido
from MovingPoints mp where mp.id = 1;
```

Essa consulta verifica a diferença, em dias, entre o momento atual e aquele em que o veículo em questão teve sua posição atualizada pela última vez. Depois, o valor é multiplicado por 86400, a quantidade de segundos em um dia, para que o valor passe a ser considerado em segundos, resultando neste caso em 10 segundos. Assim, utiliza-se o método *MovingPoint_Typ.updatePosition*(), passando como parâmetro o tempo averiguado.

Em 10 segundos, a 60 km/h, o automóvel percorreria aproximadamente 166,6 metros, pois espaço = velocidade x tempo, no caso: espaço = 16,66 m/s (60 km/h) x 10 s ≅ 166,6 m. A última observação foi realizada no ponto PI (2282,12 ; 856,54), e corresponde à posição que deve ser atualizada em 10 segundos, ou seja, o objeto deve percorrer 166,6 m a partir dela. Assim, por semelhança de triângulos, verifica-se que o novo posicionamento do veículo é o ponto (2448,52 ; 847,30), conforme demonstrado na figura 4.4.

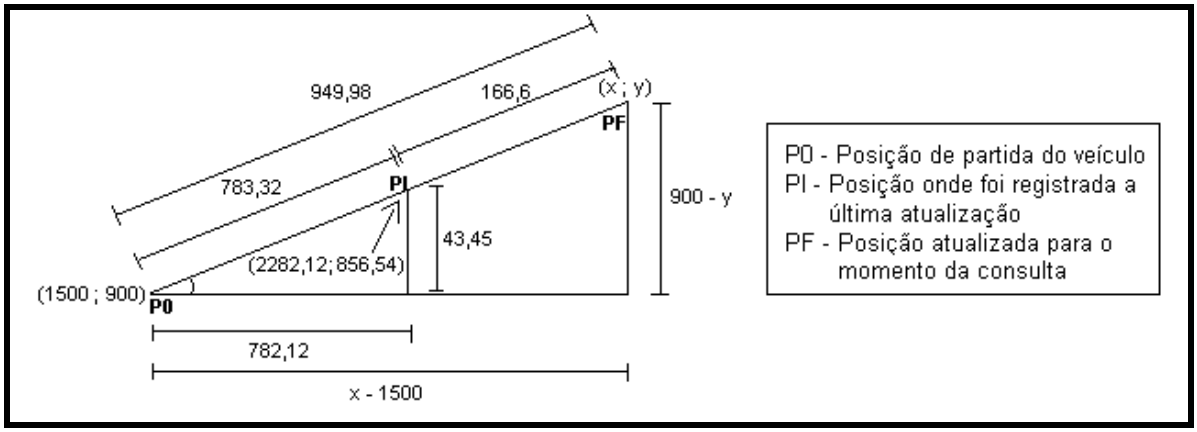

Figura 4.4. Demonstração do cálculo do posicionamento da consulta 1

Por fim, para saber o tempo que o veículo levará para chegar até o ponto desejado, verificamos a sua distância até o ponto e, de posse da sua velocidade, determinamos o tempo. A distância entre os pontos (2448,52 ; 847,30) e (3300,800) é de 852,79 metros, calculada

utilizando-se o teorema de Pitágoras. Assim, tempo = 852,79 m ÷ 16,66 m/s = 51,18 segundos, que é aproximadamente o tempo retornado pela consulta, exibido na figura 4.3, considerando-se que não foi utilizado o mesmo número de casas decimais no cálculo.

Consulta 2: Onde estava o veículo 1 às 15:39:00?

A consulta 2 procura a posição de determinado objeto dado um instante de tempo passado em relação ao momento de submissão. Neste estudo de caso, a consulta foi realizada no instante de tempo 15:42:35, e foi solicitada a posição do objeto no instante 15:39:00. Como o armazenamento de instantes de tempo no Oracle9i é feito juntamente com a data, considera-se tanto instante de submissão da consulta quanto o instante procurado como sendo ocorridos na mesma data. O resultado retornado é mostrado na figura 4.5.

```
Procedimento PL/SQL concluído com sucesso.
         X
----------
2931,24692
         Y
----------
1660,42384
```

Figura 4.5. Resultado da consulta 2

Para realizar a consulta, deve-se buscar no histórico de observações a última atualização anterior ao instante de tempo em que se deseja saber a posição. Assim, encontra-se que no instante de tempo 15:38:52, o veículo encontrava-se na posição PI (3001,28 ; 1497,02).

Após isso, deve-se atualizar esta posição em 8 segundos, que é a diferença entre o instante de tempo consultado e o instante em que houve a atualização. Com o veículo a 80 km/h (22,22 m/s), devem ser percorridos 177,76 m nestes 8 segundos. Por semelhança de triângulos, demonstrada na figura 4.6, verifica-se que a nova posição do objeto é PF (2931,25; 1660,41), que é aproximadamente o valor retornado na consulta.

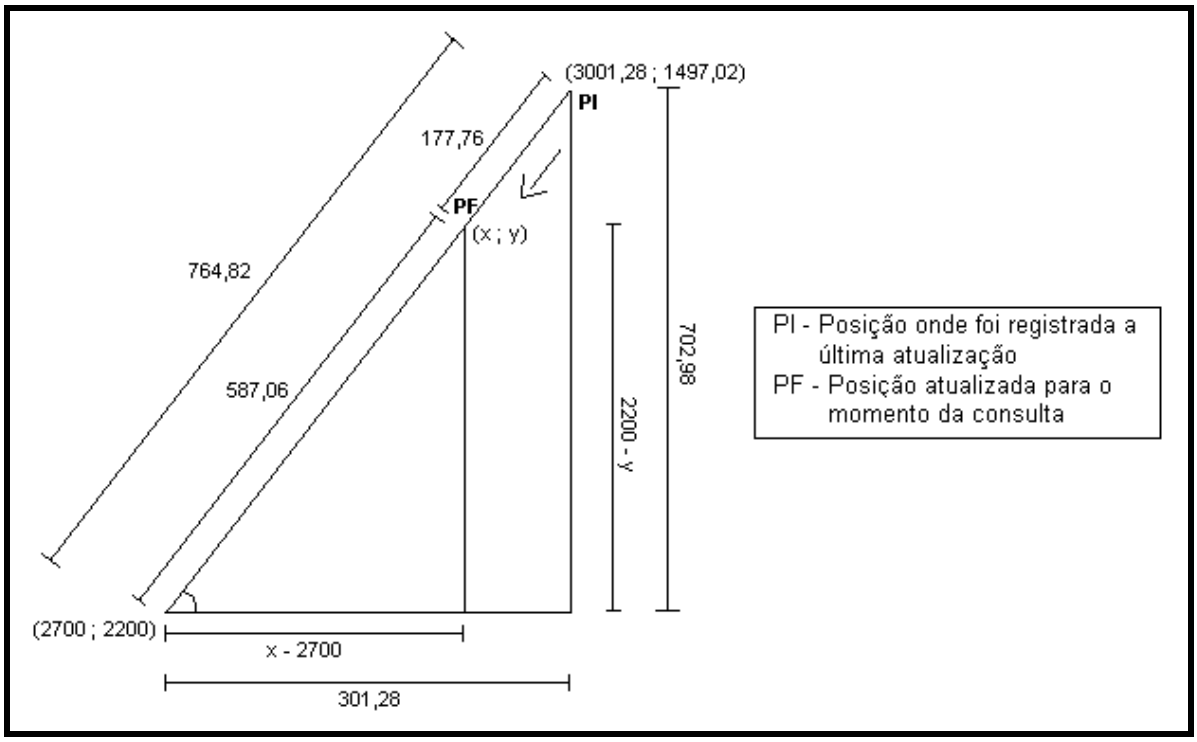

Figura 4.6. Demonstração do cálculo do posicionamento da consulta 2

Consulta 3: Onde estará o veículo 1 às 07:33:00?

Esta consulta é semelhante à consulta 2, porém ao invés de se referir a um tempo passado, consulta uma posição que o objeto deverá ocupar no futuro. Assim, ao invés de se basear em uma amostra armazenada no histórico de movimentações do veículo, a consulta deve se basear na última posição estabelecida do veículo. Esta posição deve ser atualizada baseando-se em quanto tempo existe entre a última atualização e o tempo desejado pela consulta. A figura 4.7 mostra o resultado da consulta.

```
Procedimento PL/SQL concluído com sucesso.
         X
----------
3495,24216
         Y
----------
772,108263
```

Figura 4.7. Resultado da consulta 3

A submissão desta consulta ocorreu no instante 07:32:16, tendo a última atualização de posição do veículo 1 ocorrido no instante 07:32:15, reportando a posição PI (2748,07 ; 830,66). Assim, a última atualização ocorreu 45 segundos antes da posição que se deseja consultar, sendo necessário atualizar a posição do veículo neste tempo. Em 45 segundos, a 60 km/h (16,67 m/s) percorre-se a distância de 750,15 metros. A figura 4.8 demonstra a atualização da posição, com a posição PF (x ; y) sendo a posição de destino, que a consulta deve retornar.

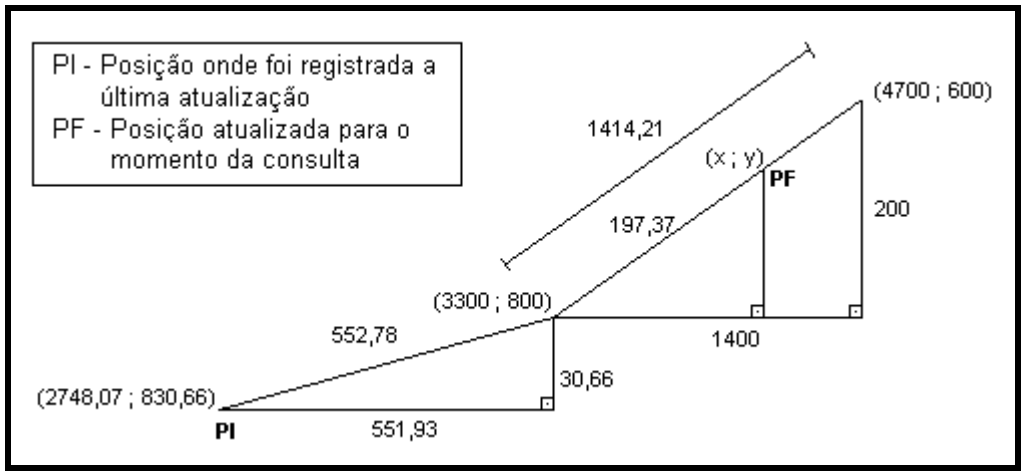

Figura 4.8. Demonstração da consulta 3.

Como no primeiro segmento da rua já podem ser percorridos 552,78 metros, faltam 197,37 metros para completar os 750,15 metros a serem percorridos até o instante de tempo consultado. Por semelhança de triângulos, a posição de destino encontrada é (3495,39 ; 772,09), que é aproximadamente igual à posição retornada pela consulta.

Consulta 4: Qual veículo está mais próximo do ponto (1300;850)?

A consulta 4 busca o veículo que se encontra mais próximo de um ponto indicado no momento em que é submetida. Como as ruas não são representadas por um modelo de grafo, objetos em ruas diferentes daquela na qual se encontra o ponto de referência têm sua distância medida de forma direta, ou seja, em linha reta. Para que essa medida leve em consideração o traçado que deve ser tomado através das ruas, deve-se acrescentar ao modelo a capacidade de roteamento, o que inclui um modelo de grafo para representar as ruas.

A consulta foi então realizada considerando o ponto de coordenadas (1300;850) como referência, e consulta veículos que trafegam no momento pela mesma rua da qual faz parte, a rua 1. Como a consulta foi submetida no instante de tempo 10:30:42, retornou o resultado mostrado na figura 4.9.

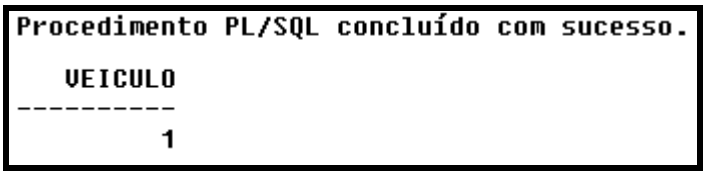

Figura 4.9. Resultado da consulta 4 no instante 10:30:42.

No referido instante de tempo, a posição dos veículos estava defasada em 14 segundos, sendo atualizada para (3047,61 ; 814,02) e (3193,54 ; 805,91), veículos 1 e 2 respectivamente. A atualização de posições já foi demonstrada nas consultas anteriores. Nessas posições, suas distâncias para o ponto (1300 ; 850) eram de 1756,15 metros para o veículo 1 e 1902,31 metros para o veículo 2, , estando o veículo 1 mais próximo do referido ponto, conforme a resposta da consulta.

Consulta 5: Em quanto tempo ocorrerá a interseção das trajetórias dos veículos 1 e 2?

A consulta 5 considera dois veículos que transitam em uma mesma rua, e desenvolvem uma trajetória na qual tendem a se cruzar. Assim, deseja-se saber em quanto tempo estes veículos irão se cruzar. No exemplo são considerados os veículos 1 e 2, que partiram das posições (1500 ; 900) e (4601 ; 614,14), respectivamente. A figura 4.10 mostra o resultado da consulta.

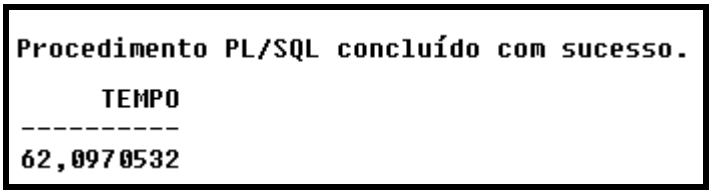

Figura 4.10. Resultado da consulta 5

Primeiro, deve-se atualizar as posições dos veículos para a posição estimada no momento em que foi realizada a consulta. Assim, verifica-se que a última atualização dos veículos reportava as posições (1999,23 ; 872,26) para o veículo 1 e (4132,15 ; 681,12) para o veículo 2. Considerando-se que já foram decorridos 5 segundos da atualização dos veículos, suas posições são ajustadas para (2082,43 ; 867,64) e (4056,53 ; 691,92), respectivamente, como já foi demonstrado nas consultas anteriores.

A partir das posições consideradas, deve-se calcular a distância entre os veículos percorrendo-se a rua em que eles transitam. Esse valor pode ser calculado aplicando-se o teorema de Pitágoras em cada um dos segmentos que formam a rua, conforme demonstrado na figura 4.11. Assim, conclui-se que a distância entre os objetos é de 1219,45 + 764,21 = 1983,66 m.

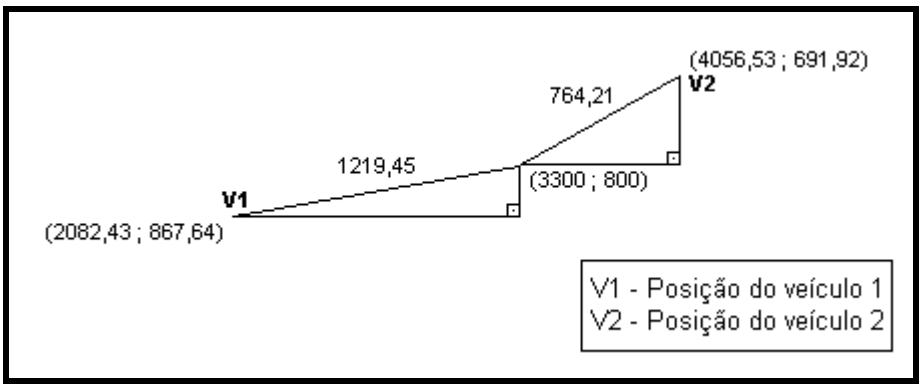

Figura 4.11. Demonstração da consulta 5.

Com o veículo 1 a 60 km/h e o veículo 2 a 55 km/h, tem-se uma velocidade relativa de 115 km/h (31,94 m/s). Assim, o tempo necessário para que os veículos se cruzem é de 1983,66 ÷ 31,94 = 62,10 segundos, que é aproximadamente o tempo retornado pela consulta.

Consulta 6: Em que ponto ocorrerá a interseção das trajetórias dos veículos 1 e 2?

Essa consulta é semelhante à consulta anterior, porém busca conhecer o ponto aonde as trajetórias dos veículos irão se encontrar. Realiza os mesmos cálculos da consulta anterior, acrescentando um cálculo ao final. Uma vez que se sabe em quantos segundos o encontro se

dará, é só verificar onde estará qualquer um dos objetos passados tais segundos. A figura 4.12 mostra o resultado da consulta.

```
Procedimento PL/SQL concluído com sucesso.
         X
----------
3115,79263
         Y
----------
810,233743
```

Figura 4.12. Resultado da consulta 6.

Utilizando-se os mesmos dados da consulta 5, basta saber onde estará o objeto 1 após 62,10 segundos. A uma velocidade de 60 km/h (16,67 m/s), ele deve percorrer 16,67 x 62,10 = 1035,21 metros. A figura 4.13 mostra o cálculo necessário para se conhecer o posicionamento do objeto.

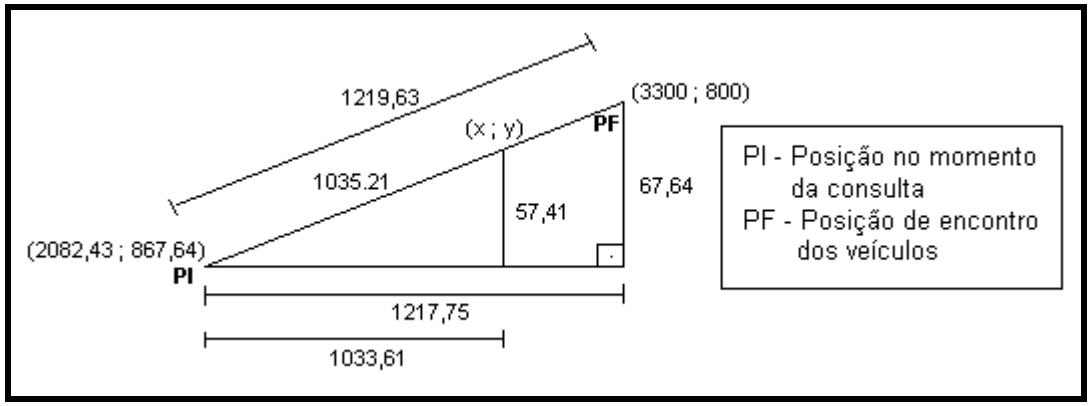

Figura 4.13. Demonstração da consulta 6.

Assim, a posição do veículo seria PF (3116,04 ; 810,23), que é aproximadamente o valor encontrado pela consulta.

4.6. Considerações Finais

Este capítulo discorreu sobre os aspectos de implementação de uma aplicação baseada no MOMENT. Foram descritos os requisitos da aplicação e colocados detalhes de como foi implementado o sistema na base de dados e o simulador. Por fim, um estudo de caso mostra o funcionamento do sistema, com dados fictícios e algumas consultas realizadas sobre esses dados.

Capítulo 5. Conclusão

O desenvolvimento da pesquisa em objetos móveis torna possível o surgimento de uma nova gama de aplicações de grande utilidade, principalmente para os setores de transportes e militar. Contudo, aplicações desse tipo trazem inerente uma série de desafios para que se realize sua implementação. O surgimento de um padrão para o desenvolvimento desse tipo de aplicação daria impulso para que uma série de sistemas desse tipo fosse implementada e possibilitaria até mesmo que pacotes para tal fim fossem fornecidos juntamente com os SGBD's disponíveis no mercado.

Nesta dissertação foi desenvolvido um modelo, o MOMENT, que procura resolver os principais problemas enfrentados para se desenvolver sistemas para monitoração de objetos móveis. Foi também desenvolvida uma aplicação usando o MOMENT, como forma de demonstrar a viabilidade de implementação do modelo.

5.1. Principais Contribuições

A principal contribuição da dissertação é desenvolvimento do MOMENT, um modelo orientado a objetos que não é específico para um tipo de aplicação, mas serve para trabalhar com objetos móveis em geral. Sendo utilizado um SGBD Objeto-Relacional, além de transportar toda a funcionalidade para dentro da base de dados, o modelo é mais facilmente extensível. Assim, torna-se mais fácil adaptar o MOMENT a qualquer tipo de aplicação de objetos móveis, sendo necessário apenas o desenvolvimento de novas classes através de herança das classes do modelo.

Sendo o modelo bastante genérico, é possível torná-lo base para uma padronização em se tratando de bancos de dados de objetos móveis. Assim como já acontece com aplicações de

bancos de dados espaciais, os fabricantes de SGBD podem se interessar por fornecer um pacote destinado a aplicações de bancos de dados móveis juntamente com o seu produto.

Outra importante contribuição do modelo é o fato de que não é exigido um alto poder de processamento da parte do dispositivo que equipa o objeto móvel. Isso é fundamental, visto que grande parte do interesse de se desenvolver aplicações móveis está voltada para trabalhar com dispositivos de baixo poder de processamento, como telefones celulares, PDA e computadores de bordo de automóveis. Assim, o MOMENT não é limitado ao uso para aplicações que disponham de dispositivos poderosos, como *notebooks*.

Outra contribuição do trabalho é a implementação de uma aplicação baseada no MOMENT. O desenvolvimento da aplicação valida o modelo, mostrando que ele é passível de implementação, já que outros modelos sofrem críticas quanto à inviabilidade de serem implementados.

Há ainda o simulador, desenvolvido para fornecer ao banco de dados posições atualizadas de objetos móveis imaginários. Durante o desenvolvimento da dissertação, foi testado um simulador já existente, o City Simulator, da IBM. Este, no entanto não se mostrou adequado, por gerar uma movimentação desordenada e sem padrão, que fugia do comportamento de objetos reais que se desejava simular. O simulador desenvolvido lê posições na base de dados e as atualiza usando as próprias funções de atualização de posição já implementadas na base de dados.

5.2. Trabalhos Futuros

As pesquisas sobre objetos móveis e as funcionalidades que podem ser implementadas dão margem a uma série de trabalhos. A partir dos resultados obtidos com o MOMENT, pode-se identificar algumas questões importantes para o aperfeiçoamento e extensão deste trabalho. Seguem algumas sugestões de funcionalidades ainda não implementadas e que podem ser realizadas em trabalhos futuros:

- Roteamento: utilizando um grafo para representar as ruas da cidade é possível utilizar algoritmos de menor caminho entre pontos, e assim verificar a distância entre os objetos e um ponto de referência através da via, e não a distância aérea entre eles. Isso porque há casos em que objetos que estão mais próximos de um ponto pela distância direta não

seriam os mais próximos através da via, devido à sinuosidade da mesma. Além disso, usando grafos pode-se definir a melhor rota entre dois pontos.

- Implementação de perfis de movimentação: uma das funcionalidades desenvolvidas no MOMENT, a definição de perfis de movimentação, não foi implementada no sistema. Pode-se adaptar o sistema para realizar previsões que se baseiem no perfil de movimentação e assim aumentar a taxa de acerto das previsões. Para tanto, ainda é necessário definir de que forma serão observados os padrões de movimentação que alimentarão os perfis.

- Mobilidade para regiões: Nesta versão do MOMENT foram considerados apenas objetos sem extensão como portadores de mobilidade, ou seja, o modelo trata apenas de pontos móveis. Pode-se modificar o modelo para que ele passe a aceitar também regiões móveis, provendo suporte a outras categorias de sistemas como, por exemplo, sistemas meteorológicos ou monitores de queimadas e incêndios.

- Desempenho: Pode-se verificar maneiras de melhorar o desempenho do acesso do simulador aos dados ou das gravações realizadas, bem como das consultas realizadas. No desenvolvimento deste trabalho não foram levadas em conta questões de otimização de consultas.

6. Referências Bibliográficas

[AG97] ADAM, Nabil R. e GANGOPADHYAY, Aryya. Database Issues in Geographic Information Systems. Boston: Kluwer Academic Publishers, 1997, 136 p.

[Bar99] BARBARA, Daniel. Mobile Computing and Databases - A Survey. IEEE Transactions on Knowledge and Data Engineering. Páginas 108 – 117, 1999.

[BP97] BADRINATH, B.R. e PHATAK, Shirish Hemant. An Architecture for Mobile Databases. Department of Computer Science Technical Report DCS-TR-324, 1997.

[BP00] BEARD, Kate e PALANCIOGLU, Mustafa. Estimating Positions and Paths of Moving Objects. Em Workshop Proceedings IEEE Press. Páginas 155-162, 2000.

[CCL01] COX S.; CUTHBERT A.; LAKE R.; Et al. Geography Markup Language (GML) 2.0, OpenGIS Implementation Specification, OGC Document Number: 01-029, 2001. Disponível em http://www.opengis.net/gml/01-029/GML2.html em 14 de dezembro de 2001.

[CF00] CLEMENTINI, Eliseo e FELICE, Paolino Di. Spatial Operators. Em SIGMOD Record 29(3). Páginas 31-38, 2000.

[DK98] DUNHAM, Margaret H. e KUMAR, Vijay. Location Dependent Data and its Management in Mobile Databases. Em Proceedings of the Ninth International Workshop on Database and Expert Systems Applications. Páginas 414 – 419, 1998.

[EGS+99] ERWIG, M.; GUTING, R.; SCHNEIDER, M.;Et al. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. GeoInformatica, 1999.

[FGN+00] FORLIZZI, Luca; GÜTING, Ralf Hartmut; NARDELLI, Enrico; Et al. A Data Model and Data Structures for Moving Objects Databases. Em Proc. ACM SIGMOD Conf. (Dallas, Texas), Páginas 319 – 330, 2000.

[GBE+00] GÜTING, Ralf Hartmut; BÖHLEN, Michael H.; ERWIG, Martin; Et al. A Foundation for Representing and Querying Moving Objects. Em ACM Transactions on Database Systems, Vol. 25, No. 1, Páginas 1 – 42, 2000.

[JHE99] JING, Jin; HELAL, Abdelsalam e ELMAGARMID, Ahmed. Client-Server Computing in Mobile Environments. Em ACM Computing Surveys, Vol. 31, No.2, 1999.

[KMJ01] KAUFMAN, James H.; MYLLYMAKI, Jussi; e JACKSON, Jared. City Simulator. Disponível em http://www.alphaworks.ibm.com/tech/citysimulator, em novembro de 2001.

[MP94] MEDEIROS, Claudia Bauzer e PIRES, Fatima. Databases for GIS. Em ACM SIGMOD Record 23 (1), Páginas 107 – 115, 1994.

[OGIS98] The OpenGIS Consortium. OpenGIS simple features specification for SQL. Technical Report, disponível em http://www.opengis.org/techno/specs.htm em abril de 2002.

[OM96] OLIVEIRA, Juliano Lopes e MEDEIROS, Claudia Bauzer. User Interface Issues in Geographic Information Systems. Relatório Técnico IC-06-96, 1996.

[PGS02] PIERRE, Gustavo Moreira; GATASS, Marcelo; SEIXAS, Roberto de Beauclair. Ambiente Integrado para Posicionamento em Operações Militares. Em IV Simpósio Brasileiro de Geoinformática, 2002.

[PLM01] PORKAEW, Kriengkrai; LAZARIDIS, Iosif e MEHROTRA Sharad. Querying Mobile Objects in Spatio-Temporal Databases. SSTD. Páginas 59 – 78, 2001.

[PS01] PITOURA, Evangelia e SAMARAS, George. Locating Objects in Mobile Computing. IEEE Trans. Know. 2001.

[RD00] REN, Qun e DUNHAM, Margaret H. Using Semantic Caching to Manage Location Dependent Data in Mobile Computing. Em Sixth Annual International Conference on Mobile Computing and Networking (MobiCom'00), 2000.

[SBM98] STONEBRAKER, Michael; BROWN, Paul e MOORE, Dorothy. Object-Relational DBMSs: Tracking the Next Great Wave. Morgan Kaufmann, 1998. 350 p.

[SDK01] SEYDIM, Ayse Yasemin; DUNHAM, Margaret H. e KUMAR, Vijay. Location Dependent Query Processing:Overview of a Framework. Submetido ao Mobility in Databases and Distributed Systems, 2001.

[SWC+97] SISTLA, A. P.; WOLFSON, O.; CHAMBERLAIN, S.; Et al. Modeling and Querying Moving Objects. Em Proceedings of the Thirteenth International Conference on Data Engineering (ICDE'97), Birmingham U.K., IEEE Computer Society. Páginas 422 – 432, 1997.

[SWC+99] SISTLA, A. P.; WOLFSON, O.; CHAMBERLAIN, S.; Et al. Updating and Querying Databases that Track Mobile Units. Em Distributed and Parallel Databases Journal on Mobile Data Management and Applications, Kluwer Academic Publishers. Páginas 257 – 287, 1999.

[Tan96] TANENBAUM, Andrew S. Redes de Computadores. Rio de Janeiro: Campus, 1997, 993 p.

[TDP+94] TERRY, Douglas B.; DEMERS, Alan J.; PETERSEN, Karin; Et al. Session Guarantees for Weakly Consistent Replicated Data. Em Proc. of the Intl. Conference on Parallel and Distributed Information Systems (PDIS), Páginas 140 – 149, 1994.

[TWZC02] TRAJCEVSKI, Goge; WOLFSON, Ouri; ZHANG, Fengli; Et al. The Geometry of Uncertainty in Moving Objects Databases. Em Proceedings of the International Conference on Extending Database Technology (EDBT02), Páginas 233 – 250, 2002.

[VW01] VAZIRGIANNIS, Michalis e WOLFSON, Ouri. A Spatiotemporal Model and Language for Moving Objects on Road Networks. Em Symposium on Spatial and Temporal Databases (SSTD), 2001.

[WSCY99] WOLFSON, Ouri; SISTLA, A. Prasad; CHAMBERLAIN, Sam; Et al. Updating and Querying Databases that Track Mobile Units. Em Distributed and Parallel Databases Journal on Mobile Data Management and Applications, 7(3), Páginas 257 – 287, 1999.

[WSX+99] WOLFSON, Ouri; SISTLA, A. Prasad; XU, Bo; Et al. DOMINO: Databases fOr MovINg Objects tracking. Em SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania,USA. ACM Press. Páginas 547 – 549, 1999.

[WXC+98] WOLFSON, Ouri; XU, Bo; CHAMBERLAIN, Sam; Et al. Moving objects databases: Issues and solutions. Em 10th Conf. on Scientific and Statistical Database Management. Páginas 111 – 122, 1998.

[YM02] YI, Bei e MEDEIROS, Claudia Bauzer. Um Modelo de Dados para Objetos Móveis. Em IV Simpósio Brasileiro de Geoinformática, 2002.

Apêndice A. Oracle Spatial – Características Oracle para Dados Espaciais

A.1. Introdução

O Oracle Spatial é um módulo do Oracle8i composto por um conjunto de funções e procedimentos, destinado a armazenar, acessar e analisar dados espaciais. Estes são dados que representam a localização de objetos no espaço real ou conceitual.

O Oracle Spatial é composto de um esquema SQL e funções que facilitam o armazenamento, recuperação, atualização e consulta a coleções de dados espaciais no banco de dados Oracle. Consiste dos seguintes componentes: um esquema para o armazenamento, sintaxe e semântica dos tipos de dados geométricos, um mecanismo de indexação espacial, um conjunto de operadores e funções para realização de consultas e junções espaciais, além de utilidades administrativas.

O atributo espacial de um objeto é a representação geométrica de seu formato em algumas coordenadas do espaço. Essas coordenadas são referidas como sendo a geometria do objeto. O Oracle Spatial permite o armazenamento de geometrias em dois modelos: Objeto-Relacional e Relacional. Apenas o modelo Objeto-Relacional será discutido nesse documento.

A.1.1. Modelo Objeto-Relacional

O modelo Objeto-Relacional usa para o armazenamento dos dados espaciais uma tabela comum contendo uma coluna do tipo MDSYS.SDO_GEOMETRY, que contém uma linha por instância (objeto representado). O este modelo possui as seguintes vantagens:

- Tipos adicionais de geometrias são acrescentados em relação ao modelo Relacional. São eles arcos, círculos, polígonos compostos, seqüências de linhas compostas, etc.

- Facilidade de uso com a utilização de índices nos campos espaciais.

- Geometrias são modeladas em uma única linha, em uma única coluna.

- Melhora no desempenho.

A.2. Dados Espaciais

Um exemplo comum de dados espaciais seria um mapa rodoviário. Ele seria um objeto bidimensional contendo pontos, linhas e polígonos que podem representar cidades, estradas e fronteiras estaduais ou nacionais. Essas representações devem preservar distâncias e posições relativas.

Um atributo espacial é uma seqüência ordenada de vértices conectados por linhas retas ou arcos. A semântica do atributo é determinada por seu tipo, que pode ser um ponto, seqüências de linha ou polígono, os considerados tipos primitivos. Pontos são compostos de duas ordenadas, X e Y, que correspondem a latitude e longitude, no caso de representação do espaço terrestre. Seqüências de linha são compostas por um ou mais pares de pontos que definem segmentos de reta. Polígonos são formados por seqüências de linhas que formam um anel fechado, caracterizando seu interior e exterior. A figura A.1 mostra os tipos geométricos primitivos.

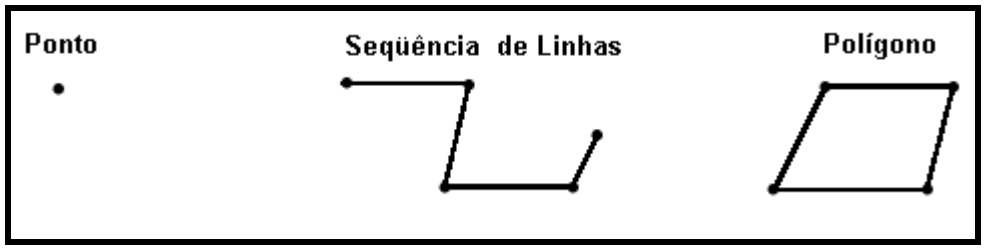

Figura A.1. Tipos Geométricos Primitivos

Um polígono não permite linhas cruzadas, pois dessa forma não é possível identificar o interior do polígono. Uma seqüência de linhas admite o cruzamento, mas não se torna um polígono.

O modelo Objeto-Relacional pode representar também os seguintes tipos de dados: arcos, polígonos de arcos, polígonos compostos, seqüência de linhas composta, círculos e retângulos, conforme a figura A.2.

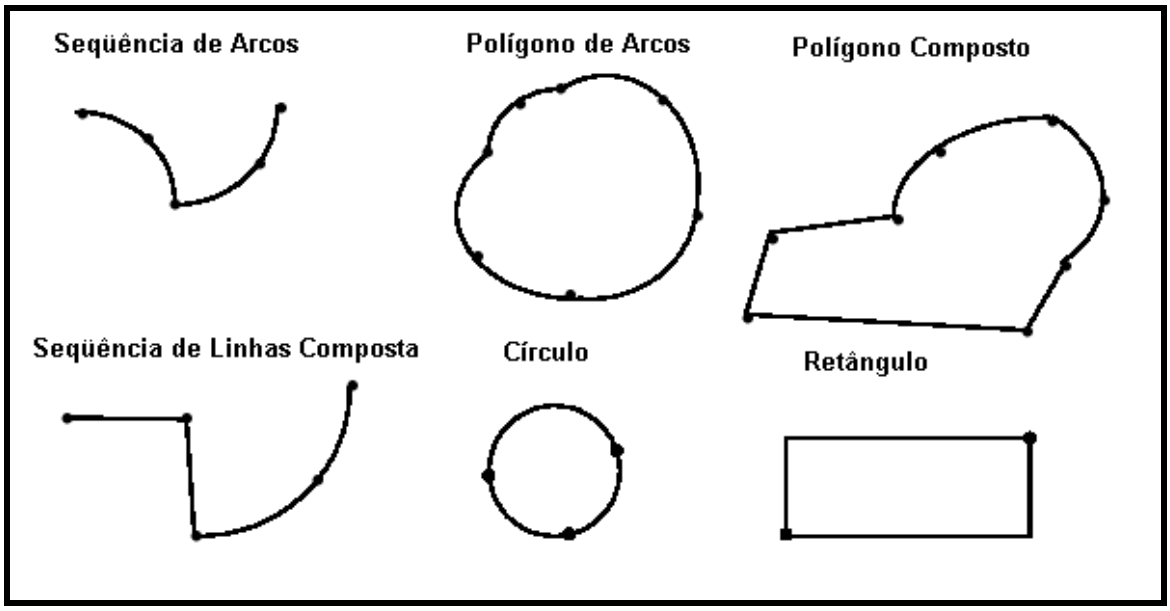

Figura A.2. Tipos Geométricos adicionais usando o modelo Objeto-Relacional

A.2.2. Representação dos Dados

O modelo de dados espacial é uma estrutura hierárquica, formada por elementos, geometrias e camadas. Camadas são compostas por geometrias, que são compostas por elementos.

O elemento é a base da construção das geometrias. Os tipos de elementos suportados são os pontos, seqüências de linhas e os polígonos. Nos elementos as coordenadas são armazenadas como um par X, Y. Um ponto é representado por uma única coordenada, uma linha por um par de coordenadas que forma um segmento e um polígono consiste de pares de coordenadas e um par de vértices para cada segmento do polígono. Os anéis interior e exterior de um polígono com buracos são considerados dois elementos distintos que formam um polígono complexo.

Uma geometria é modelada como um conjunto de elementos primitivos, podendo ser constituída por um único elemento ou por coleções homogêneas e heterogêneas de elementos. Um exemplo seria vários polígonos para representar um conjunto de ilhas, sendo um conjunto homogêneo. Se diferentes tipos primitivos forem usados, então o conjunto é heterogêneo.

Uma camada é um conjunto heterogêneo de geometrias com o mesmo conjunto de atributos. Como exemplo, uma camada poderia representar características topológicas, com

outra representando densidade demográfica, uma terceira representando a malha de rodovias do local. As camadas são armazenadas no banco de dados, assim como seus índices, em tabelas convencionais.

Muitas funções espaciais aceitam um parâmetro de tolerância. Se a distância entre dois pontos for menor ou igual à tolerância, então os dois pontos são considerados como sendo um só. A tolerância utilizada é reflexo da precisão exigida da representação espacial dos dados no banco. Por exemplo, uma consulta pede todas as farmácias que distam até 5 Km de determinado ponto, com tolerância de 0,1 Km. Assim, uma farmácia que dista 5,1 Km do ponto solicitado será também retornada como resultado da consulta, o que não aconteceria com uma tolerância de 0,05 Km.

A.2.3. Modelo de Consultas

O Oracle Spatial utiliza um modelo de duas camadas para realizar consultas e junções, ou seja, são necessárias duas operações para realizá-los. A saída das duas operações constitui o conjunto de resultados. As duas operações são chamadas de filtro primário e secundário, e são definidas a seguir:

O filtro primário realiza uma rápida seleção de registros candidatos para passar ao filtro secundário. Pode ser considerado uma operação de baixo custo computacional, pois realiza apenas comparações de aproximações geométricas para que o filtro secundário seja aplicado a um conjunto menor de dados. O conjunto de resposta gerado pelo filtro primário contém o conjunto de respostas final.

O filtro secundário é aplicado sobre o resultado gerado pelo filtro primário, gerando uma resposta exata a uma consulta espacial. É uma operação de custo computacional mais elevado, mas não é aplicado a todo o conjunto de dados.

A função SDO_GEOM.RELATE é usada como filtro secundário, verificando se duas geometrias dadas se tocam, cobrem a área uma da outra ou outros tipos de interação. O Oracle Spatial permite que se realize consultas apenas usando o filtro primário, evitando uma operação custosa em casos em que não há necessidade.

A.3. Indexação de Dados Espaciais

Um índice espacial, como qualquer outro, provê um mecanismo para reduzir as buscas na base de dados, no caso, as buscas baseadas em critérios espaciais. Um índice espacial é usado para achar objetos em um espaço de dados indexado que interceptem um determinado ponto ou área de interesse (janela de consulta), ou para achar pares de objetos que interajam espacialmente entre si (junção espacial).

O Oracle Spatial oferece dois mecanismos de indexação: *R-tree* (o default) e *quadtree*, sendo cada um deles apropriado para situações diferentes. Ambos os índices podem ser mantidos para uma mesma coluna espacial. Índices são criados e mantidos como índices comuns, através dos comandos SQL CREATE INDEX e ALTER INDEX. Os mecanismos de indexação devem ser estudados com mais profundidade durante o desenvolvimento do modelo de objetos móveis, previsto no cronograma desta proposta.

A.4. Relações Espaciais e Filtragem

O Oracle Spatial usa métodos de filtragem para relações espaciais entre entidades no banco de dados. As relações espaciais são baseadas na localização, sendo as mais comuns baseadas na topologia e na distância. Por exemplo, as fronteiras são as linhas que separam o objeto do resto do espaço e o interior é a parte o objeto que não pertence às fronteiras. Assim, dois objetos são ditos adjacentes se eles compartilhas parte da fronteira e nenhuma parte de seu interior.

Para determinar relações espaciais o Oracle Spatial tem alguns métodos de filtragem secundária:

- SDO_RELATE – verifica relacionamentos entre objetos por critérios topológicos. Alguns desses relacionamentos são citados a seguir:
  - DISJOINT – as fronteiras e interiores não se interceptam
  - TOUCH – as fronteiras se interceptam, mas os interiores não
  - EQUAL – os objetos têm os mesmos interiores e fronteiras

- CONTAINS – o interior e fronteira de um objeto estão completamente contidos no interior do outro objeto
- INSIDE – o oposto de CONTAINS
- ANYINTERACT – os objetos são não-disjuntos

- SDO_WITHIN_DISTANCE – verifica se dois objetos estão em uma distância Euclidiana um do outro

- SDO_NN – retorna um número especificado de objetos que estão mais próximos de determinada geometria

A.5. Estrutura de Metadados

Os metadados de geometria descrevem as dimensões, limites superior e inferior do espaço e a tolerância. Esses metadados ficam armazenados em uma tabela global que não deve ser diretamente atualizada. Para acesso aos metadados, foram definidas três visões para os usuários:

- USER_SDO_GEOM_METADATA – contém os metadados de todas as tabelas criadas pelo usuário. Única visão que pode ser atualizada, é onde os usuários devem inserir metadados relacionados a suas tabelas com dados espaciais.

- ALL_SDO_GEOM_METADATA – contém metadados de todas as tabelas com dados espaciais às quais o usuário tem acesso de leitura. Essa visão é *read only*.

- DBA_SDO_GEOM_METADATA – contém metadados de todas as tabelas com dados espaciais às quais o usuário tem acesso de leitura quando tiver permissões de DBA. Também é uma visão *read only*.

Cada coluna espacial deve ter seus metadados atualizados na visão USER_SDO_GEOM_METADATA. O Oracle Spatial cuida para que as outras visões tenham os reflexos dessas atualizações.

A visão USER_SDO_GEOM_METADATA tem a seguinte definição:

```
(
TABLE_NAME VARCHAR2(32),
COLUMN_NAME VARCHAR2(32),
DIMINFO MDSYS.SDO_DIM_ARRAY,
SRID NUMBER
);
```

As outras visões acrescentam a coluna OWNER, que identifica o usuário que possui a tabela. Essas informações serão descritas a seguir.

A.5.1. TABLE_NAME e COLUMN_NAME

TABLE_NAME contém o nome da tabela que possui colunas espaciais, enquanto COLUMN_NAME é o nome de uma dessas colunas, gerando uma associação entre tabela e coluna que representa uma camada.

A.5.2. DIMINFO

A coluna DIMINFO é um vetor de tamanho variável, onde cada entrada representa uma dimensão. Assim, com um espaço tridimensional o vetor deve ter três elementos. A definição de DIMINFO é a seguinte:

```
Create Type SDO_DIM_ARRAY as VARRAY(4) of SDO_DIM_ELEMENT;
        Create Type SDO_DIM_ELEMENT as OBJECT (
        SDO_DIMNAME VARCHAR2(64),
        SDO_LB NUMBER, -- limite inferior da dimensão
        SDO_UB NUMBER, -- limite superior da dimensão
        SDO_TOLERANCE NUMBER); -- tolerância da dimensão
```

Cada SDO_DIM_ELEMENT representa uma dimensão, e deve ter todos os seus valores não-nulos.

A.5.3. SRID

Contém informações a respeito das coordenadas (plano cartesiano, latitude e longitude, etc.).

A.6. Esquema Objeto-Relacional

A implementação Objeto-Relacional do Oracle Spatial consiste de um conjunto de objetos, um tipo de índice e operações nesses tipos. Uma geometria é armazenada em um objeto do tipo de dados SDO_GEOMETRY. Índices espaciais são criados e mantidos usando comandos SQL tradicionais, como CREATE, ALTER, UPDATE, etc.

A.6.1. Exemplos

Os exemplos a seguir irão obedecer à seguinte ordem de operações:

- Criação de uma tabela para armazenar dados espaciais
- Inserção de objetos
- Atualização da visão USER_SDO_GEOM_METADATA para refletir as dimensões dos objetos
- Criação de um índice espacial
- Execução de algumas consultas

O exemplo a seguir descreve a criação de uma tabela de exemplo, com uma coluna "area" que armazena geometrias.

```
CREATE TABLE exemplo (
id NUMBER PRIMARY KEY,
nome VARCHAR2(32),
area MDSYS.SDO_GEOMETRY);
```

Os exemplos a seguir inserem na tabela exemplo dois objetos como áreas, um retângulo e um polígono, ambos bidimensionais, chamados de area1 e area2. Nos exemplos abaixo, os seguintes parâmetros são passados no objeto SDO_GEOMETRY: SDO_GTYPE, que contém um código que indica o tipo de geometria e o número de dimensões; SDO_SRID, que associa um sistema de coordenadas à geometria (passado como NULL); SDO_POINT, usado apenas para pontos, para seu uso os dois parâmetros seguintes deveriam ser nulos (aqui este parâmetro é também passado como NULL); SDO_ELEM_INFO, que é um vetor para interpretação das ordenadas; e SDO_ORDINATES, que define os pontos das ordenadas.

O vetor SDO_ELEM_INFO passa uma tripla de informações para cada elemento que compõe a geometria. Isso é útil para polígonos com anéis internos. Cada tripla é composta do seguinte: SDO_STARTING_OFFSET, que indica a posição no vetor de ordenadas onde se inicia a definição do elemento correspondente; SDO_ETYPE, que indica o tipo do elemento; e SDO_INTERPRETATION, que indica a quantidade de elementos que compõe a geometria, no caso de serem mais de um, ou define como a seqüência de ordenadas deve ser interpretada caso seja apenas um elemento. Seguem os códigos dos exemplos:

```
INSERT INTO exemplo VALUES(
1,
'area1',
MDSYS.SDO_GEOMETRY(
2003, -- define um polígono bidimensional
NULL,
NULL,
MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,3), -- um retângulo (1003 =
exterior)
MDSYS.SDO_ORDINATE_ARRAY(1,1, 5,7) -- apenas dois pontos são
necessários para definir um retângulo
)
);

INSERT INTO exemplo VALUES(
2,
'area2',
```

```
MDSYS.SDO_GEOMETRY(
2003, -- define um polígono bidimensional
NULL,
NULL,
MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,1), -- um polígono (anel
exterior do polígono)
MDSYS.SDO_ORDINATE_ARRAY(5,1, 8,1, 8,6, 5,7, 5,1)
)
);
```

O próximo exemplo mostra a atualização da visão USER_SDO_GEOM_ METADATA, necessário antes da criação de índices espaciais. É necessário para cada camada (cada camada é uma associação entre uma tabela e uma coluna espacial).

```
INSERT INTO USER_SDO_GEOM_METADATA
VALUES (
'exemplo',
'area',
MDSYS.SDO_DIM_ARRAY( -- grid 20X20, aproximadamente zero de
tolerância
MDSYS.SDO_DIM_ELEMENT('X', 0, 20, 0.005),
MDSYS.SDO_DIM_ELEMENT('Y', 0, 20, 0.005)
),
NULL -- SRID
);
```

A seguir alguns exemplos de consultas espaciais.

- Retorne a interseção topológica de duas geometrias

```
SELECT SDO_GEOM.SDO_INTERSECTION(a.area, m.diminfo, b.area,
m.diminfo)
FROM exemplo a, exemplo b, user_sdo_geom_metadata m
WHERE m.table_name = 'exemplo' AND m.column_name = 'area'
AND a.nome = 'area1' AND b.nome = 'area2';
```

- Verifique se as duas geometrias têm alguma relação espacial

```
SELECT SDO_GEOM.RELATE(a.area, m.diminfo, 'anyinteract',
b.area, m.diminfo)
FROM exemplo a, exemplo b, user_sdo_geom_metadata m
WHERE m.table_name = 'exemplo' AND m.column_name = 'area'
AND a.nome = 'area1' AND b.nome = 'area2';
```

A.7. Considerações Finais

O Oracle apresenta, através do Oracle Spatial, uma série de características bastante úteis para o armazenamento de dados espaciais. Com isso, mostra-se uma boa opção de SGBD Objeto-Relacional para a implementação de um sistema envolvendo objetos móveis que devem ser armazenados em uma base de dados, sistema esse que deve se aproveitar bastante dessas características espaciais, bem como de características temporais do SGBD.

Bibliografia

[Ora01] Oracle Spatial User's Guide and Reference Version 8.1.7. Disponível em http://otn.oracle.com/docs/products/oracle8i/doc_library/817_doc/inter.817/a85337.pdf, em 13 de dezembro de 2001.

Apêndice B. Implementação – Scripts PL/SQL

Geometry_Typ.sql

```
--tipo geometria
create or replace type Geometry_Typ as object(
  geometry MDSYS.SDO_GEOMETRY,
  member function isEmpty return boolean,
  member function intersects(geom in Geometry_Typ) return boolean,
  member function equals(geom in Geometry_Typ) return boolean,
  member function touches(geom in Geometry_Typ) return boolean,
  member function overlap(geom in Geometry_Typ) return boolean,
  member function crosses(geom in Geometry_Typ) return boolean,
  member function inside(reg in Geometry_Typ) return boolean
  )
  not instantiable not final;
/

--funcoes da classe Geometry_Typ
create or replace type body Geometry_Typ as

  member function isEmpty return boolean is
  begin
    return (self.geometry is null);
  end isEmpty;

  member function intersects(geom in Geometry_Typ) return boolean is
  begin
    return
(sdo_geom.relate(self.geometry,'ANYINTERACT',geom.geometry,0.005) =
'TRUE');
  end intersects;

  member function equals(geom in Geometry_Typ) return boolean is
    begin
      return (sdo_geom.relate(self.geometry,'EQUALS',geom.geometry,0.005) =
'EQUALS');
    end equals;

  member function touches(geom in Geometry_Typ) return boolean is
  begin
    return (sdo_geom.relate(self.geometry,'TOUCH',geom.geometry,0.005) =
'TOUCH');
  end touches;

  member function overlap(geom in Geometry_Typ) return boolean is
  begin
    return
(sdo_geom.relate(self.geometry,'OVERLAPBDYINTERSECT',geom.geometry,0.005) =
'OVERLAPBDYINTERSECT');
  end overlap;

  member function crosses(geom in Geometry_Typ) return boolean is
  begin
```

```
    return
(sdo_geom.relate(self.geometry,'OVERLAPBDYDISJOINT',geom.geometry,0.005) =
'OVERLAPBDYDISJOINT');
  end crosses;

  member function inside(reg in Geometry_Typ) return boolean is
  begin
    return (sdo_geom.relate(self.geometry,'INSIDE',reg.geometry,0.005) =
'INSIDE');
  end inside;

end;
/
```

Point_Typ.sql

```
--tipo ponto
create or replace type Point_Typ under Geometry_Typ(
  member function getX return number,
  member function getY return number,
  member function onBorder(reg in Geometry_Typ) return boolean,
  member function inInterior(reg in Geometry_Typ) return boolean,
  member function distance(p in Point_Typ) return number,
  static function init(x in number, y in number) return Point_Typ
  )
  not final;
/

create or replace type body Point_Typ as

  member function getX return number is
  begin
    return (self.geometry.sdo_ordinates(1));
  end getX;

  member function getY return number is
  begin
    return (self.geometry.sdo_ordinates(2));
  end getY;

  member function onBorder(reg in Geometry_Typ) return boolean is
  begin
    return   (sdo_geom.relate(self.geometry,'TOUCH',reg.geometry,0.005)   =
'TOUCH');
  end onBorder;

  member function inInterior(reg in Geometry_Typ) return boolean is
  begin
    return   (sdo_geom.relate(self.geometry,'INSIDE',reg.geometry,0.005)   =
'INSIDE');
  end inInterior;

  member function distance(p in Point_Typ) return number is
  begin
    return sdo_geom.sdo_distance(self.geometry,p.geometry,0.005);
  end distance;
```

```
  static function init(x in number, y in number) return Point_Typ is
  begin
    return Point_Typ(MDSYS.SDO_GEOMETRY(2001,
                                        null,
                                null,
                                MDSYS.SDO_ELEM_INFO_ARRAY(1,1,1),
                                MDSYS.SDO_ORDINATE_ARRAY(x,y)));
  end init;

end;
/

create or replace type Point_Lst as varray(100) of Point_Typ;
/
```

Curve_Typ.sql

```
--tipo curva
create or replace type Curve_Typ under Geometry_Typ(
  member function getStartPoint return Point_Typ,
  member function getEndPoint return Point_Typ,
  member function isRing return boolean,
  member function length return number
)
  not instantiable not final;
/

create or replace type body Curve_Typ as

  member function getStartPoint return Point_Typ is
  begin
    return
Point_Typ.init(self.geometry.sdo_ordinates(1),self.geometry.sdo_ordinates(2
));
  end getStartPoint;

  member function getEndPoint return Point_Typ is
  ext number;
  begin
    ext := self.geometry.sdo_ordinates.count;
    return                    Point_Typ.init(self.geometry.sdo_ordinates(ext-
1),self.geometry.sdo_ordinates(ext));
  end getEndPoint;

  member function isRing return boolean is
  sx number;
  sy number;
  ex number;
  ey number;
  ext number;
  begin
    ext := self.geometry.sdo_ordinates.count;
    sx := self.geometry.sdo_ordinates(1);
    sy := self.geometry.sdo_ordinates(2);
    ex := self.geometry.sdo_ordinates(ext-1);
    ey := self.geometry.sdo_ordinates(ext);
    if (sx = ex) and (sy = ey) then
      return true;
```

```
    end if;
    return false;
  end isRing;

  member function length return number is
  begin
    return sdo_geom.sdo_length(self.geometry,0,005);
  end length;

end;
/
```

LineString_Typ.sql

```
--tipo linha
create or replace type LineString_Typ under Curve_Typ(
  member function numPoints return number,
  member function getPoint(p number) return Point_Typ,
  member function getSegment(p number) return LineString_Typ,
  static function init(points in Point_Lst) return LineString_Typ)
  not final;
/

create or replace type body LineString_Typ as
  member function numPoints return number is
  begin
    return (self.geometry.sdo_ordinates.count/2);
  end numPoints;

  member function getPoint(p number) return Point_Typ is
  x number;
  y number;
  begin
    if (p <= self.numPoints) then
      x := self.geometry.sdo_ordinates(p*2-1);
      y := self.geometry.sdo_ordinates(p*2);
      return Point_Typ.init(x,y);
    else
      return NULL;
    end if;

  end getPoint;

  member function getSegment(p number) return LineString_Typ is
  points Point_Lst;
  begin
    if (p >= 1) and (p <= self.numPoints() - 1) then
      points := Point_Lst(self.getPoint(p), self.getPoint(p+1));
      return (LineString_Typ.init(Points));
    end if;
  end getSegment;

  static function init(points in Point_Lst) return LineString_Typ is
  ordinates mdsys.sdo_ordinate_array;
  begin
    ordinates := mdsys.sdo_ordinate_array();
    ordinates.extend(points.count * 2);
    for i in 1 .. points.count
```

```
    loop
      ordinates(i*2-1) := points(i).getX;
      ordinates(i*2) := points(i).getY;
    end loop;
    return LineString_Typ(mdsys.sdo_geometry(2002,
                                  NULL,
                                  NULL,
                                  mdsys.sdo_elem_info_array(1,2,1),
                                  ordinates));
  end init;
end;
/
```

LinearRing_Typ.sql

```
--tipo anel
create or replace type LinearRing_Typ under LineString_Typ(
  static function init(points in Point_Lst) return LinearRing_Typ
)
  not final;
/

create or replace type body LinearRing_Typ as
  static function init(points in Point_Lst) return LinearRing_Typ is
  ordinates mdsys.sdo_ordinate_array;
  temp mdsys.sdo_geometry;
  begin
    ordinates := mdsys.sdo_ordinate_array();
    ordinates.extend(points.count * 2);
    for i in 1 .. points.count
    loop
      ordinates(i*2-1) := points(i).getX;
      ordinates(i*2) := points(i).getY;
    end loop;
    temp := mdsys.sdo_geometry(2003,
                        NULL,
                        NULL,
                        mdsys.sdo_elem_info_array(1,1003,1),
                        ordinates);
    if (sdo_geom.validate_geometry(temp,0.005) = 'TRUE') then
      return LinearRing_Typ(mdsys.sdo_geometry(2002,
                                    NULL,
                                    NULL,
                                    mdsys.sdo_elem_info_array(1,2,1),
                                    ordinates));
    end if;
    return NULL;
  end init;
end;
/
```

Region_Typ.sql

```
--tipo poligono - regiao
create or replace type Region_Typ under Geometry_Typ(
  member function area return number,
  member function centroid return Point_Typ,
  member function length return number,
  static function init(p in LinearRing_Typ) return Region_Typ);
/

--regioes apenas consideram poligonos externos, buracos nao serao
considerados nesta aplicacao


create or replace type body Region_Typ as

  member function area return number is
  begin
    return (sdo_geom.sdo_area(self.geometry,0.005));
  end area;

  member function centroid return Point_Typ is
  g mdsys.sdo_geometry;
  x number;
  y number;
  begin
    g := sdo_geom.sdo_centroid(self.geometry,0.005);
    x := g.sdo_ordinates(1);
    y := g.sdo_ordinates(2);
    return (Point_Typ.init(x,y));
  end centroid;

  member function length return number is
  begin
    return (sdo_geom.sdo_length(self.geometry,0.005));
  end length;

  static function init(p in LinearRing_Typ) return Region_Typ is
  ordinates mdsys.sdo_ordinate_array;
  begin
    ordinates := p.geometry.sdo_ordinates;
    return Region_Typ(mdsys.sdo_geometry(2003,
                                NULL,
                                NULL,
                                mdsys.sdo_elem_info_array(1,1003,1),
                                ordinates));
  end init;

end;
/
```

Moving_Typ.sql

```
create or replace type Moving_Typ;
/
create or replace type MovingPoint_Typ;
/
create or replace type Crossing_Typ;
/
create or replace type Street_Typ;
/
create or replace type Street_Type_Typ;
/
create or replace type Profile_Typ;
/

--tipo interface movel
create or replace type Moving_Typ as object(
  updateTime date,
  speed number)
  not instantiable not final;
/
```

MovingPoint_Typ.sql

```
--tipo ponto movel
create or replace type MovingPoint_Typ under Moving_Typ(
  name varchar2(30),
  updateValue Point_Typ,
  route number,
  direction number, -- 0 para sentido normal e 1 para sentido contrario
  simTime number,   -- usado pelo simulador
  member function updatePosition (time in number) return MovingPoint_Typ,
  static function init(time date, speed number, name varchar2, local
Point_Typ, street number, direction number, simTime number)
    return MovingPoint_Typ);
/

create or replace type body MovingPoint_Typ as

  member function updatePosition (time in number) return MovingPoint_Typ is
  space number;
  actualSpeed number;
  actualDirection number;
  point Point_Typ;
  streetId number;
  street_Obj Street_Typ;
  movingPoint_Obj MovingPoint_Typ;
  begin
    actualSpeed := self.speed;
    actualDirection := self.direction;
    streetId := self.route;
    space := (actualSpeed / 3.6) * time;
    select c.street into street_Obj from city c where c.id = streetId;
    point := street_Obj.pointToGo(self.updateValue,space,actualDirection);
    while (space > 0) loop
      select sc.idNext into streetId from Street_Crossings sc, Crossings c
where sc.idCrossing = c.id and point.getX() = c.crossing.crossPoint.getX()
and point.getY() = c.crossing.crossPoint.getY() and sc.idNext <> streetId;
```

```
      select c.street into street_Obj from City c where c.id = streetId;
      if (point.getX() = street_Obj.getEndPoint().getX() and point.getY() =
street_Obj.getEndPoint().getY()) then
        actualDirection := 0;
      else
        actualDirection := 1;
      end if;
      point := street_Obj.pointToGo(point,space,actualDirection);
    end loop;
    return
(MovingPoint_Typ.init(sysdate,actualSpeed,self.name,point,streetId,actualDi
rection,self.simTime));
  end updatePosition;

  static function init(time date, speed number, name varchar2, local
Point_Typ, street number, direction number, simTime number)
    return MovingPoint_Typ is
  begin
    return(MovingPoint_Typ(time,speed,name,local,street,direction,
simTime));
  end init;

end;
/
```

Street_Typ.sql

```
--tipo rua
create or replace type Street_Typ under LineString_Typ(
  name varchar2(40),
  oneWay number,
  typeStreet number,
  member function getName return varchar2,
  member function measure return number,
  member function routeDistance(p1 in Point_Typ, p2 in Point_Typ) return
number,
  member function pointToGo(pt in Point_Typ, distance in out number,
direction in number) return Point_Typ,
  member function simplePointToGo(pt in Point_Typ, distance in number,
direction in number) return Point_Typ,
  static function init(points in Point_Lst, name in varchar2, oneWay in
number, typeSt in number) return Street_Typ);
/

create or replace type body Street_Typ as

  member function getName return varchar2 is
  begin
    return name;
  end getName;

  member function measure return number is
  point1 Point_Typ;
  point2 Point_Typ;
  acm number;
  i number;
  begin
    i := 1;
```

```
    acm := 0;
    while (i <> self.numpoints()) loop
      point1 := self.getPoint(i);
      point2 := self.getPoint(i+1);
      acm := acm + point1.distance(point2);
      i := i + 1;
    end loop;
    return acm;
  end measure;

  member function routeDistance(p1 in Point_Typ, p2 in Point_Typ) return
number is
  i number;
  j number;
  acm number;
  segment LineString_Typ;
  begin
    acm := 0;
    for i in 1 .. self.numPoints()-1 loop
      segment := self.getSegment(i);
      if (p1.intersects(segment)) and (p2.intersects(segment)) then
        acm := p1.distance(p2);
        goto return_line;
      end if;
      if (p1.intersects(segment))then
        j := i;
        acm := acm + p1.distance(segment.getEndPoint());
        while (j < self.numPoints()) loop
          j := j + 1;
          segment := self.getSegment(j);
          if (p2.intersects(segment)) then
            acm := acm + p2.distance(segment.getStartPoint());
            goto return_line;
          else
            acm                  :=                  acm                 +
segment.getStartPoint().distance(segment.getEndPoint());
          end if;
        end loop;
      end if;
      if (p2.intersects(segment))then
        j := i;
        acm := acm + p2.distance(segment.getEndPoint());
        while (j < self.numPoints()) loop
          j := j + 1;
          segment := self.getSegment(j);
          if (p1.intersects(segment)) then
            acm := acm + p1.distance(segment.getStartPoint());
            goto return_line;
          else
            acm                  :=                  acm                 +
segment.getStartPoint().distance(segment.getEndPoint());
          end if;
         end loop;
      end if;
    end loop;
    <<return_line>>
    return acm;
  end routeDistance;
```

```
  member function pointToGo(pt in Point_Typ, distance in out number,
direction in number) return Point_Typ is
  i number;
  ind number;
  segment LineString_Typ;
  dist number;
  ponto Point_Typ;
  x number;
  y number;
  begin
    for i in 1 .. self.numPoints()-1 loop
      segment := self.getSegment(i);
      if (pt.intersects(segment)) then
        ind := i;
        ponto := pt;
        goto calcula;
      end if;
    end loop;
    <<calcula>>
    if (direction = 1) then
      while not (ponto.equals(self.getEndPoint())) loop
        dist := ponto.distance(segment.getEndPoint());
      if (distance > dist) then
        distance := distance - dist;
        ponto := segment.getEndPoint();
        if ((ponto.getX() = self.getEndPoint().getX()) and ponto.getY() =
self.getEndPoint().getY()) then
          goto retorno;
        end if;
      elsif (distance = dist) then
        distance := distance - dist;
        return segment.getEndPoint();
      else
        if (segment.getEndPoint().getX() >= ponto.getX()) then
          x := (ABS(segment.getEndPoint().getX() - ponto.getX()) * distance
/ dist) + ponto.getX();
        else
          x := (ABS(ponto.getX() - segment.getEndPoint().getX()) * distance
/ dist) - ponto.getX();
        end if;
        x := ABS(x);
        if (segment.getEndPoint().getY() >= ponto.getY()) then
          y := (ABS(segment.getEndPoint().getY() - ponto.getY()) * distance
/ dist) + ponto.getY();
        else
          y := (ABS(ponto.getY() - segment.getEndPoint().getY()) * distance
/ dist) - ponto.getY();
        end if;
        y := ABS(y);
        distance := 0;
        return (Point_Typ.init(x,y));
      end if;
      if not (ponto.equals(self.getEndPoint())) then
        ind := ind + 1;
        segment := self.getSegment(ind);
      else
        return ponto;
      end if;
      end loop;
```

```
    end if;
    if (direction = 0) then
      while not (ponto.equals(self.getStartPoint())) loop
        dist := ponto.distance(segment.getStartPoint());
      if (distance > dist) then
        distance := distance - dist;
        ponto := segment.getStartPoint();
        if ((ponto.getX() = self.getStartPoint().getX()) and ponto.getY() =
self.getStartPoint().getY()) then
          goto retorno;
        end if;
      elsif (distance = dist) then
        distance := distance - dist;
        return segment.getStartPoint();
      else
        if (segment.getStartPoint().getX() <= ponto.getX()) then
          x   :=   (ABS(ponto.getX()   -   segment.getStartPoint().getX())   *
distance / dist) - ponto.getX();
        else
          x   :=   (ABS(segment.getStartPoint().getX()   -   ponto.getX())   *
distance / dist) + ponto.getX();
        end if;
        x := ABS(x);
        if (segment.getStartPoint().getY() <= ponto.getY()) then
          y   :=   (ABS(ponto.getY()   -   segment.getStartPoint().getY())   *
distance / dist) - ponto.getY();
        else
          y   :=   (ABS(segment.getStartPoint().getY()   -   ponto.getY())   *
distance / dist) + ponto.getY();
        end if;
        y := ABS(y);
        distance := 0;
        return (Point_Typ.init(x,y));
      end if;
      if not (ponto.equals(self.getStartPoint())) then
        ind := ind - 1;
        segment := self.getSegment(ind);
      end if;
      end loop;
    end if;
    <<retorno>>
    return ponto;
  end pointToGo;

  member function simplePointToGo(pt in Point_Typ, distance in number,
direction in number) return Point_Typ is
  i number;
  ind number;
  segment LineString_Typ;
  dist number;
  ponto Point_Typ;
  x number;
  y number;
  auxDistance number;
  begin
    auxDistance := distance;
    for i in 1 .. self.numPoints()-1 loop
      segment := self.getSegment(i);
      if (pt.intersects(segment)) then
```

```
        ind := i;
        ponto := pt;
        goto calcula;
      end if;
    end loop;
    <<calcula>>
    if (direction = 1) then
      while not (ponto.equals(self.getEndPoint())) loop
        dist := ponto.distance(segment.getEndPoint());
      if (auxDistance > dist) then
        auxDistance := auxDistance - dist;
        ponto := segment.getEndPoint();
        if ((ponto.getX() = self.getEndPoint().getX()) and ponto.getY() =
self.getEndPoint().getY()) then
          goto retorno;
        end if;
      elsif (auxDistance = dist) then
        auxDistance := auxDistance - dist;
        return segment.getEndPoint();
      else
        if (segment.getEndPoint().getX() >= ponto.getX()) then
          x   :=   (ABS(segment.getEndPoint().getX()   -   ponto.getX())   *
auxDistance / dist) + ponto.getX();
        else
          x   :=   (ABS(ponto.getX()   -   segment.getEndPoint().getX())   *
auxDistance / dist) - ponto.getX();
        end if;
        x := ABS(x);
        if (segment.getEndPoint().getY() >= ponto.getY()) then
          y   :=   (ABS(segment.getEndPoint().getY()   -   ponto.getY())   *
auxDistance / dist) + ponto.getY();
        else
          y   :=   (ABS(ponto.getY()   -   segment.getEndPoint().getY())   *
auxDistance / dist) - ponto.getY();
        end if;
        y := ABS(y);
        auxDistance := 0;
        return (Point_Typ.init(x,y));
      end if;
      if not (ponto.equals(self.getEndPoint())) then
        ind := ind + 1;
        segment := self.getSegment(ind);
      else
        return ponto;
      end if;
      end loop;
    end if;
    if (direction = 0) then
      while not (ponto.equals(self.getStartPoint())) loop
        dist := ponto.distance(segment.getStartPoint());
      if (auxDistance > dist) then
        auxDistance := auxDistance - dist;
        ponto := segment.getStartPoint();
        if ((ponto.getX() = self.getStartPoint().getX()) and ponto.getY() =
self.getStartPoint().getY()) then
          goto retorno;
        end if;
      elsif (auxDistance = dist) then
        auxDistance := auxDistance - dist;
```

```
        return segment.getStartPoint();
      else
        if (segment.getStartPoint().getX() <= ponto.getX()) then
          x   :=   (ABS(ponto.getX()   -   segment.getStartPoint().getX())   *
auxDistance / dist) - ponto.getX();
        else
          x   :=   (ABS(segment.getStartPoint().getX()   -   ponto.getX())   *
auxDistance / dist) + ponto.getX();
        end if;
        x := ABS(x);
        if (segment.getStartPoint().getY() <= ponto.getY()) then
          y   :=   (ABS(ponto.getY()   -   segment.getStartPoint().getY())   *
auxDistance / dist) - ponto.getY();
        else
          y   :=   (ABS(segment.getStartPoint().getY()   -   ponto.getY())   *
auxDistance / dist) + ponto.getY();
        end if;
        y := ABS(y);
        auxDistance := 0;
        return (Point_Typ.init(x,y));
      end if;
      if not (ponto.equals(self.getStartPoint())) then
        ind := ind - 1;
        segment := self.getSegment(ind);
      end if;
      end loop;
    end if;
    <<retorno>>
    return ponto;
  end simplePointToGo;

  static function init(points in Point_Lst, name in varchar2, oneWay in
number, typeSt in number) return Street_Typ is
  aux LineString_Typ;
  begin
    aux := LineString_Typ.init(points);
    return Street_Typ(aux.geometry,name,oneWay,typeSt);
  end init;

end;
/
```

Crossing_Typ.sql

```
--tipo cruzamento
create or replace type Crossing_Typ as object(
  crossPoint Point_Typ,
  static function init(x in number, y in number) return Crossing_Typ);
/
create or replace type body Crossing_Typ as
  static function init(x in number, y in number) return Crossing_Typ is
  p Point_Typ;
  begin
    p := Point_Typ.init(x,y);
    return Crossing_Typ(p);
  end init;
end;
/
```

StreetType_Typ.sql

```
--tipo para tipo de rua
create or replace type StreetType_Typ as object(
  name varchar(40),
  maxSpeed number);
/
```

Config.sql

```
create table Config (
  startTime date);
```

feedCity.sql

```
create table City (
  id number,
  street Street_Typ);

-----------------------------------------------------------------------
--inserções na tabela de ruas
-----------------------------------------------------------------------
insert into City values (
  1,Street_Typ.init(
    Point_Lst(Point_Typ.init(700,700),Point_Typ.init(1500,900),
    Point_Typ.init(3300,800),Point_Typ.init(4700,600)),
  'Pituba',
  1,
  4)
)
/

insert into City values (
  2,Street_Typ.init(
    Point_Lst(Point_Typ.init(500,2000),Point_Typ.init(1200,1500),
    Point_Typ.init(2700,2200),Point_Typ.init(4100,1800)),
  'Canela',
  1,
  3)
)
/

insert into City values (
  3,Street_Typ.init(
    Point_Lst(Point_Typ.init(800,3500),Point_Typ.init(1300,3000),
    Point_Typ.init(3000,4100),Point_Typ.init(3900,3900)),
  'Barra',
  1,
  4)
)
/

insert into City values (
  4,Street_Typ.init(
```

```
    Point_Lst(Point_Typ.init(700,700),Point_Typ.init(500,2000),
    Point_Typ.init(800,3500)),
  'Amaralina',
  1,
  2)
)
/

insert into City values (
  5,Street_Typ.init(
    Point_Lst(Point_Typ.init(3300,800),Point_Typ.init(2700,2200),
    Point_Typ.init(3000,4100)),
  'Patamares',
  1,
  3)
)
/

insert into City values (
  6,Street_Typ.init(
    Point_Lst(Point_Typ.init(4700,600),Point_Typ.init(4100,1800),
    Point_Typ.init(4500,2900),Point_Typ.init(3900,3900)),
  'Itaigara',
  1,
  4)
)
/
```

feedCrossings.sql

```
create table Crossings (
  id number,
  crossing Crossing_typ);


insert into Crossings values (
1,
Crossing_Typ.init(700,700)
)
/

insert into Crossings values (
2,
Crossing_Typ.init(3300,800)
)
/

insert into Crossings values (
3,
Crossing_Typ.init(4700,600)
)
/

insert into Crossings values (
4,
Crossing_Typ.init(500,2000)
)
/
```

```
insert into Crossings values (
5,
Crossing_Typ.init(2700,2200)
)
/

insert into Crossings values (
6,
Crossing_Typ.init(4100,1800)
)
/

insert into Crossings values (
7,
Crossing_Typ.init(800,3500)
)
/

insert into Crossings values (
8,
Crossing_Typ.init(3000,4100)
)
/

insert into Crossings values (
9,
Crossing_Typ.init(3900,3900)
)
/
```

feedStreet_Crossings.sql

```
create table Street_Crossings (
  idStreet number,
  idCrossing number,
  idNext number);

insert into Street_Crossings values (1,1,4)
/
insert into Street_Crossings values (1,2,5)
/
insert into Street_Crossings values (1,3,6)
/
insert into Street_Crossings values (2,4,4)
/
insert into Street_Crossings values (2,5,5)
/
insert into Street_Crossings values (2,6,6)
/
insert into Street_Crossings values (3,7,4)
/
insert into Street_Crossings values (3,8,5)
/
insert into Street_Crossings values (3,9,6)
/
insert into Street_Crossings values (4,1,1)
/
```

```
insert into Street_Crossings values (4,4,2)
/
insert into Street_Crossings values (4,7,3)
/
insert into Street_Crossings values (5,2,1)
/
insert into Street_Crossings values (5,5,2)
/
insert into Street_Crossings values (5,8,3)
/
insert into Street_Crossings values (6,3,1)
/
insert into Street_Crossings values (6,6,2)
/
insert into Street_Crossings values (6,9,3)
/
```

feedStreetType.sql

```
create table StreetType (
  id number,
  type StreetType_Typ);


insert into StreetType values (1,StreetType_Typ('Viela',20))
/

insert into StreetType values (2,StreetType_Typ('Secundaria',40))
/

insert into StreetType values (3,StreetType_Typ('Principal',60))
/

insert into StreetType values (4,StreetType_Typ('Mao Dupla',80))
/

insert into StreetType values (5,StreetType_Typ('Auto Estrada',100))
/
```

feedMovingPoints.sql

```
create table MovingPoints(
  id number,
  object MovingPoint_Typ);


insert into MovingPoints values (
1,
MovingPoint_Typ.init(null,60,'v1',Point_Typ.init(1500,900),1,1,0))
/

insert into MovingPoints values (
2,
MovingPoint_Typ.init(null,55,'v2',Point_Typ.init(4601.00505,
614.142136),1,0,0))
/
```

```
insert into MovingPoints values (
3,
MovingPoint_Typ.init(null,50,'v3',Point_Typ.init(1200,1500),2,0,0))
/

insert into MovingPoints values (
4,
MovingPoint_Typ.init(null,45,'v4',Point_Typ.init(1300,3000),3,1,0))
/

insert into MovingPoints values (
5,
MovingPoint_Typ.init(null,60,'v5',Point_Typ.init(4500,2900),6,0,0))
/

create table MovingPointsHist(
  id number,
  object MovingPoint_Typ,
  time date);
```

consulta1.sql

```
-- 1 - Em quanto tempo o veículo 1 chegará ao ponto (3300,800)?

variable tempo number;

declare
  veiculo MovingPoint_Typ;
  tDecorrido number;
  local Point_Typ;
  velocidade number;
  idRua number;
  espaco number;

begin
  select mp.object into veiculo from MovingPoints mp where mp.id = 1;
  select ((sysdate - mp.object.updateTime) * 86400) into tDecorrido from
MovingPoints mp where mp.id = 1;
  veiculo := veiculo.updatePosition(tDecorrido);
  local := veiculo.updateValue;
  velocidade := veiculo.speed;
  idRua := veiculo.route;
  select   c.street.routeDistance(local,   Point_Typ.init(3300,800))   into
espaco from City c where c.id = idRua;
  :tempo := espaco / (velocidade / 3.6);
end;
.
run;

print :tempo;
```

consulta2.sql

```
-- 2 - Onde estava o veículo 1 às 14:02:20?

variable x number;
variable y number;

declare
  veiculo MovingPoint_Typ;
  tempo date;
  tempoAnt date;
  seconds number;

begin
  select to_date('29-JAN-2003 14:02:20','DD-MON-YYYY HH24:MI:SS') into
tempo from dual;
  select max(mph.time) into tempoAnt from MovingPointsHist mph where
mph.time < tempo and mph.id = 1;
  select mph.object into veiculo from MovingPointsHist mph where mph.time =
tempoAnt;
  select (tempo - tempoAnt) * 86400 into seconds from dual;
  veiculo := veiculo.updatePosition(seconds);
  :x := veiculo.updateValue.getX();
  :y := veiculo.updateValue.getY();
end;
.
run;

print :x;
print :y;
```

consulta3.sql

```
-- 3 - Onde estará o veículo 1 às 14:05:00?

variable x number;
variable y number;

declare
  veiculo MovingPoint_Typ;
  tempo date;
  seconds number;

begin
  select to_date('29-JAN-2003 14:05:00','DD-MON-YYYY HH24:MI:SS') into
tempo from dual;
  select mp.object into veiculo from MovingPoints mp where mp.id = 1;
  select (tempo - veiculo.updateTime) * 86400 into seconds from dual;
  veiculo := veiculo.updatePosition(seconds);
  :x := veiculo.updateValue.getX();
  :y := veiculo.updateValue.getY();
end;
.
run;

print :x;
print :y;
```

consulta4.sql

```
-- 4 - Qual veículo está mais próximo do ponto (1300,850)?

variable veiculo number;
variable espera number;

declare
  tDecorrido number;
  distance number;
  id number;
  momentprov date;
  v MovingPoint_Typ;

begin

  select ((sysdate - mp.object.updateTime) * 86400) into tDecorrido from
MovingPoints mp where mp.id = 1;

  select mp.object into v from MovingPoints mp where mp.id = 1;
  select
c.street.routeDistance(mp.object.updatePosition(tDecorrido).updateValue,
Point_Typ.init(1300,850))
    into distance from MovingPoints mp, City c where mp.id = 1 and c.id =
1;

  select mp.object into v from MovingPoints mp where mp.id = 2;
  select
c.street.routeDistance(mp.object.updatePosition(tDecorrido).updateValue,
Point_Typ.init(1300,850))
    into distance from MovingPoints mp, City c where mp.id = 2 and c.id =
1;

  select
min(c.street.routeDistance(mp.object.updatePosition(tDecorrido).updateValue
, Point_Typ.init(1300,850)))
  into distance from MovingPoints mp, City c where c.id = mp.object.route
and c.id = 1;

  select mp.id into id from MovingPoints mp, City c
  where
(c.street.routeDistance(mp.object.updatePosition(tDecorrido).updateValue,
Point_Typ.init(1300,850))
     between (distance - 0.005) and (distance + 0.005)) and
mp.object.route = 1;

  :veiculo := id;

end;
.
run;

print :veiculo;
```

consulta5.sql

```
-- 5 - Em quanto tempo ocorrerá a interseção das trajetórias dos veículos 1
e 2?

variable tempo number;

declare
  v1 MovingPoint_Typ;
  v2 MovingPoint_Typ;
  distance number;
  speed number;
  seconds number;

begin
  select mp.object into v1 from MovingPoints mp where mp.id = 1;
  select (sysdate - v1.updateTime) * 86400 into seconds from dual;
  v1 := v1.updatePosition(seconds);

  select mp.object into v2 from MovingPoints mp where mp.id = 2;
  select (sysdate - v2.updateTime) * 86400 into seconds from dual;
  v2 := v2.updatePosition(seconds);

  select     c.street.routeDistance(v1.updateValue,v2.updateValue)     into
distance from City c where c.id = v1.route;
  speed := v1.speed + v2.speed;
  :tempo := distance / (speed / 3.6);
end;
.
run;

print :tempo;
```

consulta6.sql

```
-- 6 - Em que ponto ocorrerá a interseção das trajetórias dos veículos 1 e
2?

variable x number;
variable y number;

declare
  v1 MovingPoint_Typ;
  v2 MovingPoint_Typ;
  distance number;
  speed number;
  seconds number;
  tempo number;
  espaco number;
  destino Point_Typ;

begin
  select mp.object into v1 from MovingPoints mp where mp.id = 1;
  select (sysdate - v1.updateTime) * 86400 into seconds from dual;
  v1 := v1.updatePosition(seconds);
  select mp.object into v2 from MovingPoints mp where mp.id = 2;
  select (sysdate - v2.updateTime) * 86400 into seconds from dual;
  v2 := v2.updatePosition(seconds);
```

```
  select     c.street.routeDistance(v1.updateValue,v2.updateValue)     into
distance from City c where c.id = v1.route;
  speed := v1.speed + v2.speed;
  tempo := distance / (speed / 3.6);
  espaco := (v2.speed / 3.6) * tempo;
  select   c.street.simplePointToGo(v2.updateValue,   espaco,   v2.direction)
into destino from City c where c.id = v2.route;
  :x := destino.getX();
  :y := destino.getY();
end;
.
run;

print :x;
print :y;
```