Linguagem de Consulta para Aplicações em Tempo-Real

Cicília Raquel Maia Leite

Dissertação de Mestrado submetida à Coordenação do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande - Campus de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Processamento da Informação

Angelo Perkusich, D.Sc.
Orientador

Campina Grande, Paraíba, Brasil

Linguagem de Consulta para Aplicações em Tempo-Real

Cicília Raquel Maia Leite

Dissertação de Mestrado apresentada em Setembro de 2005

Angelo Perkusich, D.Sc.
Orientador

Maria de Fátima Queiroz Vieira, Ph.D.
Componente da Banca
Maria Lígia Barbosa Perkusich, D.Sc.
Componente da Banca
Péricles Rezende Barros, Ph.D.
Componente da Banca

Campina Grande, Paraíba, Brasil, Setembro de 2005

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

L533l 2005 Leite, Cicília Raquel Maia
Linguagem de Consulta para Aplicações em Tempo-Real / Cicília Raquel Maia Leite. Campina Grande, 2005.
78f. : il.

Inclui bibliografia.
Dissertação (Mestrado em Engenharia Elétrica) – Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia.
Orientador: Angelo Perkusich.

1 Sistema em Tempo-Real 2 Sistemas de Gerenciamento de Banco de Dados em Tempo-Real – SGBD-TR 3 SQL-99 I Título

CDU 004.655.3+004.031.43

LINGUAGEM DE CONSULTA PARA APLICAÇÕES EM TEMPO-REAL

CICÍLIA RAQUEL MAIA LEITE

Dissertação Aprovada em 02.09.2005

ANGELO PERKUSICH, D.Sc., UFCG
Orientador

MARIA LÍGIA BARBOSA PERKUSICH, D.Sc., UNICAP
Componente da Banca (Ausente)

Maria de Fatima Q. Vieira Turnell
MARIA DE FÁTIMA QUEIROZ VIEIRA TURNELL, Ph.D., UFCG
Componente da Banca

Péricles Rezende Barros
PÉRICLES REZENDE BARROS, Ph.D., UFCG
Componente da Banca

CAMPINA GRANDE - PB
Setembro - 2005

Dedicatória

Dedico esta dissertação a minha mãe, Luzia Maia, que é a minha maior fonte de inspiração, sabedoria e perseverança. Ela que sempre procura estar ao meu lado me dando forças e coragem para superar cada etapa da minha vida. À você mãe, com muito orgulho te ofereço este trabalho.

Agradecimentos

De diversas formas, e em momentos diferentes, muitas pessoas direta ou indiretamente, contribuíram para o desenvolvimento deste trabalho.

Agradeço a Deus, por permitir que eu atingisse mais este objetivo.

Ao orientador Angelo Perkusich, pela orientação, sem a qual este trabalho não seria realizado.

À Professora Lígia Perkusich, pela grandiosa dedicação, paciência e disposição para solucionar minhas dúvidas. Sua contribuição foi muito importante para elaboração deste trabalho.

Ao Pedro Fernandes, pelo extremo envolvimento na realização deste trabalho, pelas suas orientações e contribuições.

Aos Professores com os quais cursei disciplinas, em especial Maria de Fátima, José Sérgio e Berto Machado me ensinaram bastante.

Aos amigos de mestrado: em especial à Yáskara, a qual todos os dias me incentiva com sua coragem e discernimento, Alfranque, Félix, Jaidilson e Ademar pela amizade e companheirismo.

Aos amigos da UNICAP-Recife, Marcelo, Alexandre e Ched pelas experiências compartilhadas.

A todos os professores e funcionários da COPELE que me ajudaram durante todo esse trabalho.

À minha mãe querida, pelo amor, dedicação, lição de vida e coragem.

Ao meu pai, pelas suas constantes preocupações nessa etapa da minha vida.

Ao meu noivo Júnior, por acreditar e me incentivar na realização deste trabalho e por estar ao meu lado, nas horas mais difíceis.

À minha avó, por suas orações e ensinamentos constantes.

Ao meu avô (em memória), que todos os dias me mostra um caminho de flores a trilhar.

Aos meus tios, Helena, Peita, Francisca, Vicência, Mariazinha, Antônia, Alcides, Lino, Luiz, Júnior e Raimundo Vieira pela preocupação constante. Aos meus primos, que me incentivam sempre.

Aos meus amigos, de forma especial à Cris Carol, Janaina, Jane, Jânio, Suele e Samira pelo apoio incondicional.

Aos amigos do Prédio, em especial Felipe, Tarik, David, Nildo, Batista, Jaqueline, Luciano, Ayslene e Valnir que também estiveram presentes nesta etapa da minha vida.

Ao CNPq, pela bolsa de pesquisa.

Resumo

O processamento de *fluxo contínuo de dados* está surgindo como uma área de pesquisa em expansão e está voltada para o processamento de informações produzidas por dispositivos que geram grandes volumes de dados em alta velocidade e com tempo de vida útil limitado. Por exemplo, as informações geradas por sensores são seqüências contínuas e ilimitadas de dados. Tradicionalmente, tais informações requerem equipamentos e programas especiais para monitorá-las, que processam e reagem à entrada contínua de diversas origens. Entre diversas aplicações que necessitam utilizar sensores pode-se citar: estações de monitoramento de tempo, sistemas para monitoramento de pacientes, sistemas de monitoramento de satélites e muitos outros sistemas de sensoriamento em tempo-real. Este trabalho apresenta uma sintaxe para a declaração de consultas em tempo-real, denominada Linguagem de Consulta para Aplicações em Tempo-Real (LC-ATR), e comumente chamada neste trabalho de Linguagem de Consulta para Banco de Dados em Tempo-Real (LC-BDTR). Também uma interface para permitir a declaração e processamento de consultas utilizando a LC-BDTR foi desenvolvida. Tal interface pode ser utilizada em aplicações que precisem tratar com restrições temporais. Ela foi implementada através da linguagem de programação Java e do SGBD DB2 da IBM. A linguagem Java foi escolhida por disponibilizar mecanismos para tratar com restrições temporais das aplicações e o SGBD utilizado é o DB2 da IBM, por disponibilizar suas versões gratuitamente às Universidades para pesquisa e estar mais próximo do padrão SQL-99. Entretanto, o mesmo pode ser substituído por qualquer outro com as mesmas características. A sintaxe da LC-BDTR permite que os usuários definam suas consultas usando a SQL-99 adicionada de primitivas de tempo-real que correspondam às restrições temporais impostas aos dados utilizados pelas mesmas. Ela fornece uma solução inerente à limitação da arquitetura dos SGBD para processar dados gerados continuamente, provenientes de diversas origens e com restrições temporais. A vantagem desta interface está em permitir usar um SGBD comercial para processar tanto dados convencionais quanto dados gerados continuamente com restrições temporais, o que é fundamental para as empresas que precisam tratar com ambos os tipos de dados.

Abstract

Processing of continuous data streams is emerging as new and expanding area of research and concerns the processing of information from sources that produce data in a fast rate and in a continuous way. For example, information from sensory devices can be considered as a continuously expanding and unlimited sequence of data items without any boundaries. Traditionally, such information required special monitoring applications and equipment that process and react to continual inputs from several sources such as in a weather monitoring station, patient monitoring equipment, etc. This work, presents a syntax for the declaration of real-time queries, called Query Language for Application Real-Time Databases (QL-ARTDB) or Query Language for Real-Time Databases (QL-RTDB). Ahead of the developed syntax we implement a interface to process the QL-RTDB. Such interface can be used in applications that they need to deal with temporal restrictions. It was implemented through the DBMS IBM DB2 and programming language Java. The DBMS that we use is IBM DB2, however, the same can be substituted by any another one with the same characteristics. The syntax of the language allows that the users define its queries using the SQL-99 added of the primitives in real-time that they correspond to the imposed temporal restrictions to the data used for the same ones. The advantage of this interface is to accept a commerce DBMS to process conventional data as conventional data with time restrictions which is necessary to business that need to work with both ways.

Índice

Glossário

τ	Transação
ANSI	*American National Standards Institute*
BD	Banco de Dados
CLI	*Call Level Interface*
DDL	*Data Definition Language*
DML	*Data Manipulation Language*
ISO	*International Organization for Standardization*
JDBC	*Java Database Connectivity*
LC-BDTR	Linguagem de Consulta para Banco de Dados em Tempo-Real
PSM	*Persistent Stored Modules*
RTSQL	*Real-Time SQL*
SGBD	Sistemas de Gerenciamento de Banco de Dados
SGBD-OR	Sistema de Gerenciamento de Bancos de Dados Objeto-Relacional
SGBD-TR	Sistemas de Gerenciamento de Banco de Dados em Tempo-Real
SQL	*Structured Query Language*
STR	Sistemas em Tempo-Real
TVQL	*Temporal Versioned Query Language*

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Os Sistemas de Gerenciamento de Banco de Dados (SGBD) convencionais são projetados para processarem dados fisicamente armazenados em discos e em um nível lógico, vistos como uma coleção de dados persistentes. Tais dados são acessados pelo SGBD através de aplicações dos usuários ou diretamente através da interface do mesmo.

Nos últimos anos surgiram várias aplicações de Banco de Dados (BD) que violam tais princípios. Estas aplicações requerem acesso contínuo a dados produzidos em grandes volumes, alta velocidade e com validade temporal, ou seja, tem um tempo de vida útil limitado. Além disso, as aplicações que utilizam tais dados precisam ser processadas em tempo-real, visando utilizá-los enquanto os mesmos sejam válidos. Eles são usados tipicamente em aplicações tais como: redes de sensores, sistemas de monitoramento médico, sistemas de monitoramento de satélites, sistemas de monitoramento de tempo e muitos outros sistemas de sensoriamento de tempo-real.

Os SGBD convencionais não são apropriados para gerenciar tais aplicações. Os fluxos de dados precisam ser processados através de uma nova abordagem, que considere uma consulta tanto de um ponto de vista lógico quanto temporal. Os Sistemas de Gerenciamento de Banco de Dados em Tempo-Real (SGBD-TR), por exemplo, oferecem suporte para tratar tais aplicações, no entanto, ainda estão em fase de pesquisa e desenvolvimento (FERNANDES et al., 2004; LEITE et al., 2005a; NETO et al., 2004) e os existentes, geralmente são bastante caros (EMBEDDED, 2005; DATABASE, 2005).

No entanto, os SGBD convencionais em conjunto com a linguagem de programação Java apresentam as características necessárias para o desenvolvimento de uma linguagem de consulta para dar suporte às aplicações de sensoriamento de tempo-real, tais como as já citadas.

Este trabalho apresenta uma sintaxe para a declaração de consultas em tempo-real, denominada Linguagem de Consulta para Banco de Dados em Tempo-Real (LC-BDTR). Também apresenta uma implementação de uma interface para permitir a declaração e

processamento de consultas baseada na LC-BDTR. Tal interface, baseada na linguagem padrão SQL-99, pode ser utilizada em aplicações que precisem tratar com restrições temporais. Ela foi implementada através da linguagem de programação Java e do SGBD DB2 da IBM.

Como principais vantagens da substituição dos programas tradicionais usados em sistemas de sensoriamento de tempo-real por um SGBD, podem-se citar: o compartilhamento dos dados entre diversas aplicações concorrentemente e o armazenamento dos mesmos para posteriores análises. Tradicionalmente aplicações de sensores operam apenas com os dados correntes e ignoram os dados históricos.

Mesmo não podendo garantir o mesmo desempenho de um SGBD-TR ou um sistema de sensoriamento dedicado à interface desenvolvida permite reduzir o custo de desenvolvimento de aplicações que devem processar dados e transações com restrições temporais. Observe que através dela é possível desenvolver aplicações, convencionais ou não, sem a necessidade de adquirir um SGBD-TR, e as aplicações existentes podem ser estendidas para tratar com restrições temporais.

1.1 Motivação

Nos últimos anos as pesquisas na área SGBD-TR têm evoluído bastante, através dos pesquisadores da área de banco de dados e através dos pesquisadores da área de Sistemas em Tempo-Real (STR). No entanto, a simples integração destas duas tecnologias, considerando conceitos, mecanismos e ferramentas, não é suficiente para desenvolver um SGBD-TR. Apesar de muitas das técnicas usadas em STR e SGBD poderem ser aplicadas em SGBD-TR, existem muitas diferenças que necessitam adaptações das abordagens usadas nas duas áreas, ou mesmo, o desenvolvimento de novas tecnologias. Questões como técnicas de controle de concorrência, política de escalonamento e linguagem de consultas, estão sendo consideradas e pesquisadas para SGBD-TR (PERKUSICH, 2000; FERNANDES et al., 2004; LINDSTROM, 2003; LEITE et al., 2005a).

A necessidade de recuperar e manipular informações armazenadas em um banco de dados motivou a estender o padrão SQL-99 adicionada de primitivas temporais que levassem ao desenvolvimento de uma interface que permitisse recuperar e manipular dados e transações com restrições temporais, tais como as de um SGBD-TR. Uma vantagem fundamental do desenvolvimento de tal interface é a possibilidade de manipular dados com restrições temporais armazenadas em um banco de dados usando a LC-BDTR, de forma transparente ao usuário.

A linguagem de programação para o desenvolvimento da interface foi a linguagem Java devido a capacidade de disponibilizar mecanismos que dêem suporte a aplicações

que envolvem restrições temporais e por possuir uma interface JDBC (*Java DataBase Connectivity*), que facilita a portabilidade da interface em qualquer SGBD através do JDBC. Desta forma, dão suporte as primitivas de tempo-real necessárias para consultas em tempo-real. Finalmente, o SGBD utilizado foi o DB2, a escolha deve-se ao fato de usar o padrão SQL-99 por ser amplamente utilizado por várias empresas e por oferecer gratuitamente às instituições de ensino para o desenvolvimento de pesquisas.

1.2 Objetivos da Dissertação

O desenvolvimento de uma linguagem de consulta, tal como a SQL, é uma tarefa de alto custo que envolve uma grande quantidade de recursos e profissionais, de forma que apenas os grandes fabricantes de SGBD podem custear (EMBEDDED, 2005; DATABASE, 2005).

No caso de linguagens de consultas para banco de dados em tempo-real, que ainda é uma tecnologia em fase de desenvolvimento, uma alternativa para prover suporte para aplicações em tempo-real é uma especificação de uma sintaxe de uma linguagem de consulta e o desenvolvimento de uma interface que dê suporte a esta sintaxe.

Cujo objetivo deste trabalho é especificar tal linguagem de consulta e implementar a interface. De forma que, esta linguagem possa simular uma linguagem de consulta em tempo-real através da integração de uma linguagem de consulta convencional, tal como a SQL-99 e primitivas de tempo-real, desenvolvida através dos mecanismos que a linguagem de programação Java oferece. Com isso, esta integração torna-se transparente ao usuário, onde os dados e as transações são atendidos dentro de parâmetros temporais especificados pelo mesmo.

1.3 Relevância

A relevância desta dissertação é disponibilizar uma sintaxe para a declaração de consultas em tempo-real, que pode ser utilizada em aplicações que necessitem manter a integridade temporal dos dados e garantir as restrições temporais das transações. E também disponibilizar uma interface que possibilite a declaração e o processamento da mesma.

Uma relevância secundária será a elaboração da documentação da sintaxe e da interface que estendeu o padrão SQL-99, considerando os novos recursos temporais que foram adicionados ao padrão.

1.4 Organização da Dissertação

Essa dissertação está organizada como apresentado a seguir. No Capítulo 2 define-se os conceitos básicos de um SGBD-TR, situando o trabalho no escopo mais específico das primitivas de tempo-real que a linguagem de consulta deverá oferecer. No Capítulo 3 enfatiza-se os SGBD-OR, e alguns conceitos fundamentais para esses sistemas como a SQL-99. O Capítulo 4 é a base deste trabalho e apresenta a sintaxe da linguagem de consulta para aplicações em tempo-real. No Capítulo 5 apresenta-se à descrição geral da interface que dá suporte a LC-BDTR, além de um estudo de caso para validação da mesma. Por último, no Capítulo 6, apresenta-se às conclusões e trabalhos futuros.

Capítulo 2

Sistemas de Gerenciamento de Bancos de Dados em Tempo-Real

2.1 Introdução

Os Sistemas de Gerenciamento de Banco de Dados (SGBD) convencionais são projetados para executar transações concorrentes enquanto mantém a consistência lógica dos dados e transações. Por outro lado, os Sistemas de Gerenciamento de Banco de Dados em Tempo-Real (SGBD-TR) são projetados para executar transações concorrentes com restrições temporais e garantir a consistência lógica e temporal dos dados e transações.

Este Capítulo é organizado como segue. Na Seção 2.2 apresentam-se as características de um SGBD-TR. Nas Seções 2.3 e 2.4 define-se as principais características dos dados e das transações, respectivamente. Na Seção 2.5 apresentam-se as propriedades ACID no contexto de SGBD-TR. Na Seção 2.6 define-se os dados e transações de tempo-real. Finalmente, na Seção 2.7 ressaltam-se os trabalhos relacionados a linguagem de consulta para SGBD.

2.2 Características de um SGBD-TR

Os SGBD-TR podem ser vistos como a integração de um SGBD convencional com um Sistema em Tempo-Real (STR). Como um SGBD processa transações e garante a integridade lógica dos dados e como um STR garante restrições temporais das transações. Portanto, um SGBD-TR possui como principais características o tratamento de dados com validade temporal e transações com restrições explícitas de tempo de execução (PERKUSICH; TURNELL; PERKUSICH, 1999). Estas características são úteis para aplicações com tempo crítico que necessitam coletar, modificar e recuperar grandes volumes de dados compartilhados (TESANOVIC et al., 2002).

Assim, um SGBD-TR deve fornecer capacidade para gerenciar transações em tempo-real, onde as propriedades ACID[1] são aplicadas seletivamente. Ele ainda deve fornecer suporte para os usuários especificarem as restrições temporais dos dados e das transações, isto é, um intervalo de tempo durante o qual um dado é considerado válido, e o tempo máximo que deve ser utilizado para executar uma transação. Nas Seções seguintes são apresentadas as principais características dos dados e das transações de um SGBD-TR.

2.3 Características dos Dados

Os dados usados em um SGBD-TR devem refletir o estado real do ambiente da aplicação, portanto a estrutura desses dados indica que os valores gravados são válidos apenas por um determinado intervalo de tempo. Assim, os intervalos de tempo especificam a validade temporal dos dados. Desta forma, um SGBD-TR busca garantir a consistência temporal dos mesmos através da validação desses intervalos. A consistência temporal pode ser medida através da consistência absoluta dos dados e através da consistência relativa dos dados.

A consistência absoluta é a medida entre o estado real do ambiente e como ele é armazenado no banco de dados. Então, um item de dado deve ser gravado dentro de intervalos de tempo que garantam que ele reflita o estado real do ambiente. Essa medida surge da necessidade de manter a visão do sistema de controle consistente com o estado real do ambiente.

A consistência relativa é a medida entre os dados que são usados na computação de outros dados. Então, um conjunto de itens de dados deve ser gravado dentro de um mesmo intervalo de tempo, que possa representar aproximadamente o mesmo instante de tempo. Essa medida surge da necessidade de produzir novos dados a partir de dados gravados em tempos aproximados.

2.4 Características das Transações

A maioria das transações de um SGBD-TR usa dados com validade temporal, conseqüentemente as mesmas precisam estar associadas às restrições temporais para sua execução, tais como: *tempo de início, tempo de término, tempo de processamento, tempo computacional, prazo e periodicidade.*

As transações podem ser de três tipos: *estrito* (*hard*), *suave* (*soft*) e *firme* (*firm*).

- **Estrito** - Uma transação é do tipo *estrito* quando qualquer resultado que é produzido após seu prazo é inútil para o sistema e é considerada uma falha fatal;

[1] ACID é o acrônimo para Atomicidade, Consistência, Isolamento e Durabilidade

- **Suave** - Uma transação é do tipo *suave* quando o não cumprimento de seus prazos não acarreta problemas sérios, no entanto, o desempenho do sistema diminui à medida que várias transações com prazos *suaves* deixam de atender suas restrições de tempo;

- **Firme** - Uma transação é do tipo *firme* se a perda do prazo final não gerar nenhum efeito ou valor negativo para o sistema. Geralmente estas transações são recomeçadas quando perdem seu prazo final.

Muitas transações em um SGBD-TR são usadas para ler e gravar dados coletados de dispositivos, tais como sensores, visando monitorar um determinado ambiente e gerenciar os eventos ocorridos em tais ambientes.

Geralmente, em SGBD-TR, essas transações podem ser executadas de três diferentes formas: *periódica*, *aperiódica* e *esporádica*.

- **Periódica** - Uma transação é do tipo *periódica* quando ela é executada repetidamente em um intervalo de tempo regular para desempenhar uma função do sistema;

- **Aperiódica** - Uma transação é do tipo *aperiódica* quando não existe um intervalo de tempo regular para a execução da mesma;

- **Esporádica** - Uma transação é do tipo *esporádica* quando ela possui intervalo regular, podendo ou não ser executada.

Ramamrithman (RAMAMRITHMAN, 1993) caracteriza as transações de um SGBD-TR baseado na natureza das transações de três diferentes maneiras:

1. Primeiro, elas são caracterizadas de acordo com as origens das restrições temporais impostas às transações. Algumas restrições vêm dos requisitos de consistência temporal dos dados, e algumas vêm dos requisitos impostos pelo ambiente. Geralmente, elas têm a forma de requisitos de periodicidade. Por exemplo, *A cada 30 segundos verifique se o valor da temperatura medida pelo Sensor de temperatura do ambiente X passou do limite;*

2. Segundo, elas são caracterizadas baseada nos prazos impostos às transações aperiódicas. Por exemplo, *Se o valor da temperatura medida pelo Sensor de temperatura for maior que 35 graus, dentro de 20 segundos, dispare o alarme;*

3. Finalmente, elas são caracterizadas baseadas no efeito de perder seu prazo como *estrita, suave e firme*, exatamente como acontece para um STR.

Quanto ao tipo uma transação pode ser classifica como: transação de sensor (ou escrita), transação de atualização e transação de leitura.

- **Transação de sensores** - é uma transação que obtém o estado do ambiente e escreve os valores no banco de dados;
- **Transação de atualização** - é uma transação que pode ler e escrever no banco de dados;
- **Transação de leitura** - é uma transação que pode ler dados do banco de dados.

2.5 Propriedades ACID

Um projeto de banco de dados convencional deve garantir a manutenção da consistência lógica dos dados, isto é, evitar que diferentes transações escrevam informações contraditórias no banco de dados, visando manter a consistência interna do sistema. Para garantir a consistência interna, deve-se implementar corretamente todas as transações buscando assegurar as propriedades ACID, definidas a seguir, (ELMASRI; NAVATHE, 2005):

- *Atomicidade* - Uma transação deve ser totalmente executada ou nenhum passo dela deve ser considerado, ou seja, qualquer passo realizado deve ser desfeito;
- *Consistência* - A execução de uma transação deve sempre transformar o estado consistente de um banco de dados em outro estado consistente;
- *Isolamento* - Os efeitos gerados por uma transação no banco de dados não devem ser visíveis por nenhuma outra transação até que ela seja comprometida;
- *Durabilidade* - Os efeitos de uma transação devem ser permanentes.

Tais propriedades são bastante restritivas, portanto as mesmas foram redefinidas para SGBD-TR:

- *Atomicidade*- A execução atômica é seletivamente aplicada às subtransações que necessitam tratar com dados totalmente consistentes, ao invés de aplicá-la à transação toda. Por exemplo, considere que uma transação que está atualizando dados a respeito de um ambiente, perde seu prazo e deve parar de executar. Geralmente, é desnecessário desfazer as operações, uma vez que os valores anteriores também estão desatualizados. Além do mais, pode ser mais interessante dispor de um banco de dados parcialmente ou recentemente atualizado que um banco de dados totalmente desatualizado.

- *Consistência* - Algumas transações podem ter restrições temporais e podem usar dados que precisam ser temporalmente consistentes. Assim, os dados gerados por uma transação não terminada podem ser vistos por outras transações para evitar que se tornem desatualizados ou que não satisfaçam suas restrições temporais.

- *Isolamento* - As transações podem precisar se comunicar e sincronizar com outras transações de forma a executar funções de controle, portanto, suas ações podem ser vistas por outras transações mesmo antes delas terem terminado.

- *Durabilidade* - A propriedade de durabilidade, alguns dados têm validade temporal e não precisam ser gravados. Por outro lado, o banco de dados deve refletir o estado do ambiente, portanto, é fácil recriá-lo a partir da leitura dos sensores, ao invés de recriá-lo no tempo em que ocorreu uma falha, pois esses dados provavelmente já serão inválidos.

2.6 Dados e Transações de Tempo-Real

Para atender os requisitos temporais de um SGBD-TR considera-se uma estrutura para os atributos, representado como uma quádrupla: $(ordem, avi, tsr, imprec)$, onde: $ordem$: número de seqüência do atributo; avi: é o intervalo de validade absoluta, ou seja, é o intervalo de tempo no qual o dado é considerado válido temporalmente; tsr: é o rótulo de tempo, isto é, o tempo no qual o dado foi gravado no sistema; $imprec$: é a quantidade de imprecisão acumulada para o atributo. A definição dessa estrutura visa armazenar todas as informações pertinentes a um item de dado do ponto de vista temporal. Também, considera-se uma estrutura para cada transação (τ_i) definida por uma quádrupla (tl_i, tc_i, pr_i, pe_i), onde: tl_i é o tempo de liberação de τ_i , isto é, o momento no qual todos os recursos necessários à execução de τ_i estão disponíveis, a partir desse momento τ_i estará pronta para ser executada; tc_i representa o tempo computacional de τ_i, isto é, o tempo de processamento necessário para a executá-la; pr_i especifica o prazo máximo para a execução de τ_i; pe_i indica a periodicidade de τ_i, isto é, a freqüência que a transação será executada.

Basicamente os SGBD-TR apresentam três características principais:

1. A exigência de tratar dados lógica e temporalmente consistentes;

2. A exigência de satisfazer as restrições temporais das transações;

3. A exigência de permitir a negociação entre restrições lógicas e temporais.

Estas características adicionam novas exigências às linguagens de consultas para SGBD, assim como ao processamento das transações.

Além do suporte convencional de um SGBD, um SGBD-TR deve oferecer suporte para o tratamento dos dados e das transações com restrições temporais. Portanto, um SGBD-TR deve fornecer suporte para os usuários especificarem as restrições temporais dos dados, isto é, um intervalo de tempo durante o qual um dado é considerado válido, além de definir os prazos das transações, ou seja, o tempo no qual uma transação deve ser executada. Esses requisitos são bastante complexos, no entanto várias pesquisas estão sendo realizadas no sentido de atendê-los (LEITE et al., 2005b, 2005a; NETO et al., 2004; FERNANDES et al., 2004).

2.7 Trabalhos Relacionados

Várias pesquisas vêm sendo realizadas no contexto de linguagem de consultas para SGBD. No contexto de SGBD-TR, uma linguagem denominada *Real-Time SQL* (RTSQL) foi desenvolvida em (PRICHARD, 1995). A RTSQL é baseada no modelo relacional de dados e estende o padrão SQL-92 para suportar banco de dados em tempo-real, ou seja, inclui extensões que especificam a consistência temporal dos dados e das transações.

Em (MORO et al., 2002), uma Linguagem de Consulta para Versão Temporal *(Temporal Versioned Query Language - TVQL)* foi desenvolvida. Nessa linguagem, o modelo de versão temporal é baseado no modelo de dados orientado a objetos, com finalidade de armazenar a versão de objeto e, para cada versão, um histórico de seus valores dos atributos dinâmicos e relacionamentos. A linguagem utilizada é a SQL, incorporada de novas características para suportar informações temporais. No entanto, esta linguagem não considera o tratamento de restrições temporais dos dados e das transações e conseqüentemente só pode ser utilizada para banco de dados histórico.

Em (VOSSOUGH, 2004), apresenta-se uma extensão da SQL-99 para tratar com fluxo contínuo de dados, no entanto não garante que as restrições temporais dos dados e das transações sejam satisfeitas.

Considerando as características das aplicações que precisam tratar com fluxo contínuo de dados, é fundamental o desenvolvimento de uma linguagem que permita tratar com tais dados, levando em conta não apenas as restrições lógicas, mas também as restrições temporais. Normalmente, esses dados são gerados em grandes volumes e são válidos apenas por um determinado período de tempo. Conseqüentemente, as transações que acessam esses dados precisam ser definidas levando em consideração tais restrições.

2.8 Conclusões

Neste Capítulo apresentou-se os conceitos básicos e os requisitos necessários aos SGBD-TR. Dessa forma foram identificadas as características desejáveis para o desenvolvimento de uma interface que suporte a declaração e processamento da LC-BDTR.

Capítulo 3

Sistema de Gerenciamento de Bancos de Dados Objeto-Relacional

3.1 Introdução

De acordo com (ELMASRI; NAVATHE, 2005), um SGBD é constituído por um conjunto de dados associados a um conjunto de programas para acessar esses dados. O principal objetivo de um SGBD é proporcionar um ambiente tanto conveniente quanto eficiente para recuperação e armazenamento das informações do banco de dados.

Este Capítulo discute uma classe emergente de SGBD, denominado Sistema de Gerenciamento de Bancos de Dados Objeto-Relacional (SGBD-OR), e alguns conceitos fundamentais para esses sistemas. Os SGBD-OR correspondem a uma extensão dos SGBD relacionais com características das linguagens orientadas a objeto (O´NEIL; O'NEIL, 2000).

Este Capítulo é organizado como segue. Na Seção 3.2 apresenta-se uma visão geral de SGBD e do padrão SQL como uma linguagem de consulta para SGBD, e na Seção 3.3 apresenta-se o padrão SQL-99 que oferece suporte para tratar com SGBD-OR.

3.2 Visão Geral de SGBD e SQL

Os SGBD são projetados para gerenciar grandes volumes de dados compartilhados entre várias aplicações. Um *modelo de dados* é um conjunto de ferramentas conceituais usadas para a descrição de dados, relacionamentos entre dados e regras de consistência (SUDARSHAN; KORTH; SILBERSCHATZ, 2001).

Vários SGBD comerciais foram projetados para dá suporte ao modelo de dados relacional. No entanto, o modelo relacional não é suficiente para atender as exigências das aplicações emergentes, tais como: banco de dados médicos, banco de dados científicos, banco de dados multimídia, entre outras. Tais aplicações possuem tipos de dados com-

plexos, tais como: imagens, voz, texto, entre outros que não são suportados pelo modelo relacional (O´NEIL; O'NEIL, 2000).

Por outro lado, um SGBD-OR dá suporte adequado a essas aplicações por tratar com tais tipos de dados. Tal modelo é o resultado da extensão do modelo relacional com conceitos de orientação a objetos.

3.2.1 Linguagem Estruturada de Dados (SQL)

A SQL tem representado o padrão para linguagens de banco de dados relacionais e objeto-relacionais. Inúmeros produtos dão suporte atualmente para linguagem SQL. A SQL é dividida em diversas partes, conforme (SUDARSHAN; KORTH; SILBERSCHATZ, 2001):

- **Linguagem de Definição de Dados (DDL)**. A DDL proporciona comandos para a definição de esquemas de relações, exclusões de relações, criação de índices e modificação nos esquemas de relações.
- **Linguagem de Manipulação de Dados (DML)**. A DML abrange uma linguagem de consulta baseada tanto na álgebra relacional quanto no cálculo relacional de tuplas. Engloba também comandos para inserção, exclusão e modificação de tuplas no banco de dados.
- **SQL embutida**. A forma de comandos SQL incorporados foi projetada para aplicação em linguagens de programação de uso geral.
- **Definição de Visões**. A DDL possui comandos para definição de visões.
- **Autorização**. A DDL engloba comandos para especificação de direitos de acesso a relações e visões.
- **Integridade**. A DDL possui comandos para especificação de regras de integridade que os dados que serão armazenados no banco de dados devem satisfazer. Atualizações que violarem as regras de integridade serão desprezadas.
- **Controle de Transações**. A SQL inclui comandos para especificação de iniciação e finalização de transações. Algumas implementações também permitem explicitar bloqueios de dados para controle de concorrência.

A Figura 3.1, sumariza as operações básicas suportadas pela SQL, enquanto que a Figura 3.2, sumariza as diversas cláusulas suportadas pela SQL.

A SQL suporta dois tipos de operadores: os operadores lógicos, utilizados para conectar expressões nas operações básicas, normalmente numa cláusula WHERE, e os operadores de comparação, utilizados para comparar os valores de duas expressões. A Figura 3.3, sumariza os tipos de operadores suportados pela SQL.

Linguagem	Operações Básicas	Descrição
DDL	CREATE DROP ALTER	Cria tabelas, campos e índices no banco de dados. Remove tabelas, campos e índices no banco de dados. Altera a estrutura de uma tabela no banco de dados.
DML	SELECT INSERT UPDATE DELETE	Seleciona um conjunto de registros, de uma ou mais tabelas de um banco de dados, que satisfaçam a um determinado critério. Adiciona dados a um banco de dados numa única operação. Atualiza dados de um banco de dados, segundo critérios especificados. Remove registros de um banco de dados.

Figura 3.1: Operações Básicas da SQL

Cláusula	Descrição
FROM	Especifica as tabelas utilizadas como fonte de dados para a seleção dos registros.
WHERE	Especifica as condições que os registros devem satisfazer para compor o subconjunto particular dos dados.
GROUP BY	Separa os registros selecionados em diferentes grupos.
HAVING	Especifica a condição que cada grupo especificado deve satisfazer.
ORDER BY	Ordena os registros selecionados de acordo com o critério especificado.

Figura 3.2: Cláusulas da SQL

Tipo	Operador	Descrição
Operadores Lógicos	AND OR NOT	Conjunção lógica Disjunção lógica Negação lógica
Operadores de Comparação	= <> > < >= <= BETWEEN LIKE IN	Igual a Diferente de Maior que Menor que Maior ou Igual a Menor ou Igual a Especifica um intervalo de valores Especifica um padrão de comparação Especifica registros dentro de um banco de dados

Figura 3.3: Operadores da SQL

Neste trabalho enfatizam-se as linguagens DDL e DML. Maiores detalhes dessas linguagens podem ser encontradas no Apêndice A.

3.2.2 Linguagem de Definição de Dados

A DDL oferece um conjunto de comandos para definição de esquemas de relações, exclusão de relações, criação de índices e modificação nos esquemas de relações (ELMASRI; NAVATHE, 2005). As estruturas básicas da DDL são mostradas a seguir.

Estruturas Básicas da DDL

- Criação de um banco de dados

```
CREATE DATABASE nome_banco
```

  Permite criar um banco de dados onde serão armazenados os dados.

- Criação de uma tabela

```
CREATE TABLE nome_tabela{
(    definição_coluna,...
     [restrição_unicidade,...]
     [restrição_referencial,...]
     [restrição_check,...]
)}
```

  Permite uma criação de uma tabela no banco de dados.

- Remoção de uma tabela

```
DROP TABLE {nome_tabela|nome_view}
```

  Permite uma remoção de uma tabela no banco de dados.

- Remoção de um banco de dados

```
DROP DATABASE {nome_banco}
```

  Permite uma remoção de um banco de dados.

3.2.3 Linguagem de Manipulação de Dados

A linguagem de manipulação de dados permite o acesso aos dados armazenados em um banco de dados. Ela é formada por um conjunto de operações para inserção, exclusão e modificação de tuplas (registros) no banco de dados.

Estruturas Básicas da DML

- O comando SELECT é utilizado para recuperar dados armazenados no banco de dados.

```
SELECT * FROM {nome_tabela}
 [WHERE condição]
```

- O comando INSERT é utilizado para inserir tuplas em uma tabela.

```
INSERT INTO {nome_tabela|nome_view}
   |VALUES {}
```

- O comando UPDATE é utilizado para atualizar tuplas de uma tabela.

```
UPDATE {nome_tabela|nome_view} [nome_corr]
SET cláusula_atribuição
```

- O comando DELETE é utilizado para remover tuplas de uma tabela.

```
DELETE FROM {nome_tabela|nome_view} [nome_corr]
```

3.3 Padrão SQL-99 e seus Componentes

Em 1999, um novo padrão para a linguagem SQL, denominado SQL-99, foi publicado pela *(American National Standards Institute - ANSI)* e *(International Organization for Standardization-ISO)*, para o modelo de dados objeto-relacional e está documentado em (9075-1:1999, 1999; 9075-2:1999, 1999; 9075-3:1999, 1999; 9075-4:1999, 1999; 9075-5:1999, 1999). Atualmente vários SGBD comerciais, tais como: DB2, PostgreSQL, Oracle entre outros (IBM, 2004; POSTGRESQL, 2004; ORACLE, 2004), utilizam o padrão SQL-99 (O´NEIL; O'NEIL, 2000).

O padrão SQL-99 apresenta como principais componentes, (ELMASRI; NAVATHE, 2005):

- Novas estruturas para tratamento temporal (SQL/Temporal) e aspectos de transações em SQL;
- SQL/CLI*(Call Level Interface)* - Interface em Nível de Chamada;
- SQL/PSM *(Persistent Stored Modules)* - Módulos de Armazenamento Persistente;
- *SQL/Framework, SQL/Foundation, SQL/Bindings, SQL/Object.*

A SQL/Temporal trata dados históricos, dados de seqüências temporais e outras extensões temporais.

A SQL/CLI fornece regras que permitem a execução do código da aplicação sem fornecer o código-fonte, evitando a necessidade de pré-processamento. Contém aproximadamente 50 rotinas para as tarefas, como conexão com o servidor SQL, alocação e liberação de recursos, obtenção de diagnóstico e informação de implementação e controle de término de transações.

A SQL/PSM especifica as funcionalidades para particionar uma aplicação entre um cliente e um servidor. Seu objetivo é melhorar o desempenho pela minimização do tráfego de rede.

A SQL/Bindings tem a SQL embutida e a Chamada Direta *(Direct Invocation)*. A SQL embutida foi aperfeiçoada para incluir declarações de exceções adicionais.

A *SQL/Foundation* dá suporte a novos tipos de dados, novos predicados, operações relacionais, cursores, regras e gatilhos *(triggers)*, tipos definidos pelo usuário *(User Defined Types - UDT)* e capacidades de transações.

3.3.1 Características dos SGBD-OR

As principais características definidas pelo padrão ANSI/ISO SQL-99 são:

- Novos **construtores de tipos** para especificar objetos complexos. Entre eles, inclui-se o tipo *row*, que corresponde ao construtor de tuplas e o tipo *array* e *nested table* para representar coleções;
- Especificação de **identidade de objetos** através do uso do tipo referência *(reference type)*;
- O **encapsulamento de operações** que permite que tipos definidos pelo usuário possam incluir operações como parte de suas declarações;
- Mecanismos de **herança**.

Os **Construtores de Tipos**. Os construtores *row e array*, por exemplo, são utilizados para especificar tipos complexos. Eles também são conhecidos como UDT, uma vez que o usuário os define para determinada aplicação. Um **tipo row** pode ser especificado utilizando-se a seguinte sintaxe, conforme exemplo mostrado a seguir.

```
CREATE TYPE nome_do_tipo_row AS [ROW](<declarações de componentes>);
```

A palavra-chave **ROW** é opcional. Um exemplo de especificação de um UDT é ilustrado a seguir:

- Criação do UDT

```
CREATE TYPE tipo_Endereco AS OBJECT(
rua  VARCHAR(30),  - nome da rua
cidade VARCHAR(30),  - nome da cidade
cep  INTEGER,  - número do cep);


CREATE TYPE tipo_Empregado AS OBJECT(
nome  VARCHAR(30),  - nome do empregado
endereco tipo_Endereco,  - UDT Endereço
idade  INTEGER,  - idade do empregado);
```

Observe que pode-se utilizar um UDT definido anteriormente como o tipo de um atributo, como mostrado para o atributo endereço do tipo **tipo_Empregado**. Um **tipo array** pode ser especificado para um atributo cujo valor será uma coleção. Por exemplo, suponha que uma companhia possui no máximo dez localizações, então o tipo *array* para a companhia pode ser definido como:

```
CREATE TYPE tipo_Companhia AS OBJECT(
nomecomp  VARCHAR(30),  - nome da companhia
localizacao VARCHAR(30) ARRAY[10],  - localização da companhia);
```

Um tipo *array* é um conjunto ordenado de itens de dados. Todos os itens são do mesmo tipo e cada um tem um índice, que é o número correspondente no *array*. O número de itens de um *array* é seu tamanho, sendo que este pode ser variável.

Identidade de objetos. Um identificador de objeto serve para identificar unicamente um item de dado dentro de um banco de dados. Por exemplo:

```
CREATE TABLE Empregado of tipo_Empregado REF IS emp_id SYSTEM
GENERATED;
    CREATE TABLE Companhia of tipo_Companhia (
    REF IS comp_id SYSTEM GENERATED,
    PRIMARY KEY (nomecomp));
```

Observe que um atributo componente de uma tupla pode ser uma referência (especificada pelo uso da palavra-chave REF) que pode ser gerada pelo sistema, como ilustrado no exemplo anterior.

Encapsulamento de Operações. A SQL fornece uma função semelhante à definição de classes para que o usuário possa criar determinado UDT com sua própria especificação comportamental pela especificação de métodos (operações) em adição aos atributos. A forma genérica da especificação de um UDT com métodos é:

```
CREATE TYPE <nome do tipo> (
  lista dos atributos
  declaração de funções do próprio gerenciador
  declarações de funções definidas pelo usuário
);
```

Por exemplo, suponha que se deseja extrair o número de apartamentos (se houver) a partir da cadeia de caracteres que forma o atributo membro rua do tipo *row* **tipo _Endereco** declarado anteriormente. Pode-se especificar um método para **tipo_Endereco** como:

```
CREATE TYPE tipo_Endereco AS OBJECT(
  rua VARCHAR(45),
  cidade VARCHAR(25),
  CEP CHAR(5),
)

METHOD nro_apto() RETURNS CHAR(8);
```

O código para implementação do método é:

```
METHOD
CREATE FUCTION nro_apto() RETURNS CHAR(8) FOR tipo_Endereco AS
EXTERNAL NAME '/x/y/nro_apto.class' LANGUAGE 'JAVA';
```

Nesse exemplo, a implementação foi através da linguagem Java, e o código é armazenado no diretório do arquivo especificado. A SQL fornece algumas funções embutidas, conforme documentado em (9075-2:1999, 1999).

Herança. A SQL possui regras para lidar com herança (especificada pela palavra-chave *UNDER*). Para ilustrar a herança de tipos, considere o exemplo a seguir.

```
CREATE TYPE tipo_Gerente UNDER tipo_Empregado AS (
 depto_Gerenciado CHAR(10));
```

Esse tipo herda todos os atributos e os métodos do supertipo **tipo_Empregado** e possui um atributo específico adicional **depto_Gerenciado**.

3.3.2 Especificação de Tempo

A SQL-99 provê tipos de dados pré-definidos, conforme ilustrado na Tabela 3.1 e uma sintaxe e semântica para especificações de expressões temporais. Existem três tipos de dados pré-definidos relacionados a tempo, tais como: *Date*, *Time* e *Timestamp* (9075-1:1999, 1999), que são fundamental importância para o desenvolvimento desse trabalho.

Tabela 3.1: Tipos de Dados Pré-Definidos

Nome	Bytes	Mínimo	Máximo
Smallint	2	-32.768	+32.767
Integer	4	-2.147.483.648	+2.147.483.647
Real	4	-5.4E+79	+5.4E+75
Float/Double	8	-5.4E+79	+5.4E+75
Decimal (m,n)	(m/2) + 1	$1 - 10^{31}$	$10^{31} - 1$
Character (n)	n	1	255
Varchar (n)	n+2	1	4.06 (página de 4k)
CLOB	6	1	2.147.483.647
BLOB	6	1	2.147.483.647
Date	4	01.01.0001	31.12.9999
Time	3	00:00:00	24:00:00
Timestamp	10	01.01.0001.00:00:00.00.000000	31.12.9999.24:00:00.00.000000

O tipo de dados *Date* armazena informações da data (ano, mês e dia). O limite suportado é 1 de Janeiro de 0001 a Dezembro de 9999. O tipo *Time* armazena informações de tempo (horas, minutos e segundos). O limite suportado é 00 horas, 00 minutos e 00 segundos para 24 horas, 00 minutos e 00 segundos. O tipo *Timestamp* armazena

tanto informações de tempo quanto de data (ano, mês, dia, hora, minuto, segundo e microssegundos). O limite suportado é 01 de Janeiro de 0001, 00 horas, 00 minutos, 00 segundos e 000000 microssegundos para 31 de Dezembro de 9999, 24 horas, 00 minutos, 00 segundos e 000000 microssegundos.

Estes tipos de dados podem ser usados para expressar tempo absoluto, como 9h00 min. Podem também ser usados para expressar um intervalo de tempo *INTERVAL*, que pode expressar um período de tempo, tal como 5 minutos.

A SQL-99, também suporta três funções relacionadas ao tempo: (*CURRENT DATE*) que retorna a data atual do sistema, (*CURRENT TIME*) que retorna o tempo atual e *(CURRENT TIMESTAMP)* que retorna a data e o tempo atual conectados.

3.4 Conclusões

Neste Capítulo apresentou-se uma visão geral de SGBD e da linguagem de consulta SQL. Também, apresentou-se uma visão geral de SGBD-OR, além do padrão SQL-99 que dá suporte aos SGBD-OR.

Capítulo 4

Extensão da Linguagem de Consulta

4.1 Introdução

Neste Capítulo, é apresentada uma sintaxe que estende a SQL-99, denominada Linguagem de Consulta para Banco de Dados em Tempo-Real (LC-BDTR). Esta sintaxe pode ser utilizada para dá suporte a um SGBD-TR e aplicações em tempo-real. Portanto, a sintaxe da DDL e DML da SQL-99, será estendida visando considerar além das definições de restrições lógicas para dados e transações, definições de restrições temporais para dados e transações, tais como: *tempo de validade absoluta dos dados e tempo de início, tempo de término, tempo computacional, tempo de processamento e periodicidade das transações.*

A sintaxe da LC-BDTR é uma extensão da SQL-99 com primitivas de STR. Para validar essa extensão foi implementada uma interface desenvolvida através da linguagem de programação Java que acessa um SGBD, neste caso o SGBD utilizado foi o DB2, no entanto qualquer SGBD poderia ser utilizado.

Este Capítulo é organizado como segue. Na Seção 4.2 mostra-se como características encontradas nas linguagens de consulta podem ser incorporadas a uma linguagem de consulta para aplicações em tempo-real. Na Seção 4.3 analisa-se a linguagem de consulta para aplicações em tempo-real. Na Seção 4.4 apresenta-se a extensão da linguagem de consulta.

4.2 Linguagem de Consulta para SGBD-TR

As linguagens de programação em tempo-real fornecem mecanismos para expressar as restrições de tempo na execução das tarefas. Por outro lado, em um SGBD-TR, a DDL precisa disponibilizar primitivas para expressar restrições de tempo dos dados e a DML precisa disponibilizar primitivas para expressar restrições de tempo na execução das operações.

Portanto, as restrições temporais existentes nas linguagens de tempo-real precisam ser incorporadas a uma linguagem de consulta para um SGBD-TR. Um breve sumário destas restrições segue nos parágrafos seguintes.

Restrições de Tempo - são usadas para expressar o tempo de início, o tempo de término, tempo de processamento e a periodicidade da execução das transações.

Restrições de Execução - são usadas para expressar a execução das transações, tais como: tempo computacional, periodicidade e o prazo.

Restrições de Manipulação de Exceção - fornecem mecanismos de recuperação quando uma restrição temporal é violada.

Uma linguagem de consulta para SBGD-TR também é estruturada através de uma DDL e uma DML. A DDL é baseada nas características de consistência temporal dos dados e a DML nas características de consistência temporal das transações, tais características são descritas na Seção seguinte.

4.2.1 Consistência Temporal dos Dados e Transações

Na LC-BDTR, as restrições de consistência temporal dos dados são definidas através da DDL, desta forma ela disponibiliza um tipo de dado definido pelo usuário (UDT), que permite expressar as restrições temporais dos mesmos, um UDT temporal é definido na Seção 5.6.4.

Para isso, foi definida uma função denominada VALIDATE que verifica a consistência absoluta dos dados, ou seja, determina os itens de dados que são válidos temporalmente. Ela é baseada na expressão booleana $((t - tsr) \leq avi)$, onde (t) representa o tempo corrente, (tsr) o tempo no qual o dado foi gravado no sistema e (avi) o intervalo de validade absoluta do dado. Observe que, caso a expressão seja avaliada como verdadeira o item de dado é temporalmente consistente, caso contrário, é temporalmente inconsistente.

As restrições de consistência temporal das transações são declaradas através de primitivas de tempo herdadas das linguagens de tempo-real.

Para isso, foram definidas várias primitivas de tempo-real descritas a seguir:

```
<construtores de restrição temporal das transações >::=
    [INITIALIZE AT <timestamp value expression>]
    [TERMINATE AT <timestamp value expression>]
    [COMPUTATION <timestamp value expression>]
    [PERIOD <interval value expression>
      [INITIALIZE <timestamp value expression>]
      [TERMINATE <timestamp value expression>]
```

- INITIALIZE AT *(timestamp)* - é o tempo no qual a transação deve iniciar, isto é, o momento em que todos os recursos necessários para execução da mesma estão disponíveis. Ela corresponde ao tempo de liberação (tl_i), definido entre as primitivas de tempo-real no Capítulo 2;

- TERMINATE AT *(timestamp)* - é o tempo no qual uma transação deve terminar sua execução. Ela corresponde ao prazo máximo (pr_i), definido entre as primitivas de tempo-real no Capítulo 2;

- COMPUTATION *(timestamp)* - é o tempo computacional de uma a transação, ou seja, é o tempo gasto para execução de uma transação. Ela corresponde ao tempo computacional da transação (tc_i), definido entre as primitivas de tempo-real no Capítulo 2;

- PERIOD *(timestamp)* - esta primitiva permite definir a periodicidade para execução de uma transação. Ela é baseada no parâmetro período (pe_i), definido entre as primitivas de tempo-real no Capítulo 2. É importante observar, que frequentemente o *avi* de um item de dado determina a periodicidade com que uma transação deva ser executada. Por exemplo, considerando que o *avi* de um determinado item de dado seja igual a 3 segundos, isto implica que a transação de leitura desse dado precisa ser executada a cada 3 segundos, visando manter o banco de dados sempre temporalmente consistente.

Observe que o tempo de processamento pode ser derivado através de INITIALIZE AT e TERMINATE AT, conforme expressão a seguir:

- **TP = TT-TI** - O tempo de processamemento é a diferença entre o tempo de término e o tempo de início da transação, que ficou denominado como RETURN_TP.

4.3 LC-BDTR

A LC-BDTR consiste da integração da linguagem SQL-99 com primitivas de linguagens de tempo-real, conforme ilustrada na Figura 4.1. Desta forma, a LC-BDTR disponibiliza aos usuários as funcionalidades básicas de uma linguagem de consulta, além de funcionalidades para a declaração das restrições temporais inerentes às aplicações em tempo-real, tais funcionalidades são definidas a seguir:

- DDL - A SQL DDL disponibiliza comandos para a definição de esquemas de relações, exclusão de relações, criação de índices e modificações nos esquemas de relações;

- DML - A SQL DML disponibiliza comandos para inserção, exclusão e modificação de tuplas no banco de dados;
- Primitivas de Tempo-Real - Disponibiliza através das primitivas de tempo-real a especificação, gerenciamento e consultas com restrições temporais.

A linguagem SQL-99 disponibiliza além da DDL e DML diversas outras partes, tais como: definição de visões, integridade, autorização, controle de transação que não são enfatizados neste trabalho.

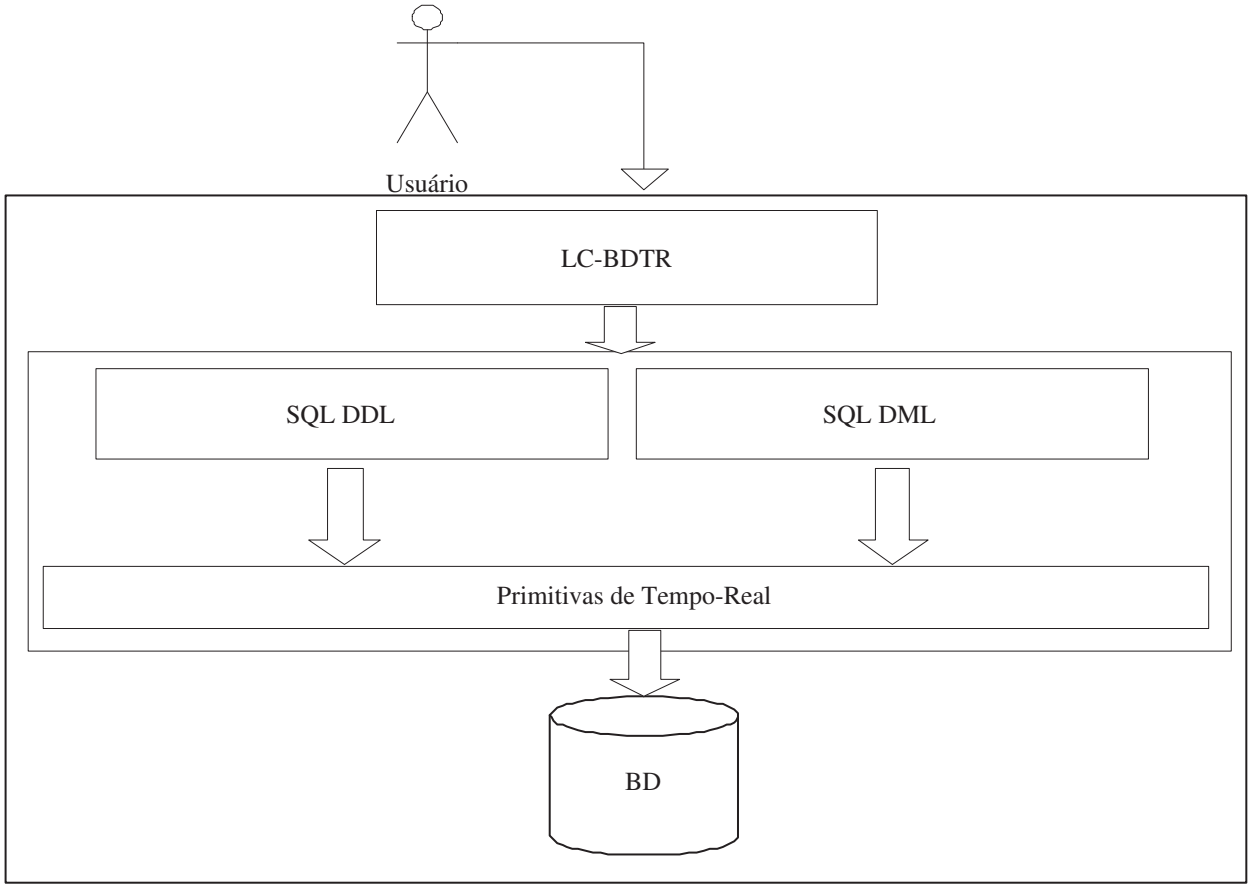

Figura 4.1: Arquitetura da LC-BDTR

4.4 Extensão da SQL-99

Visando validar a sintaxe da LC-BDTR foi definida uma Tabela Sensor, conforme Figura 4.2 e elaboradas várias consultas utilizando à mesma. Tal sintaxe é dividida em duas classes: a sintaxe simplificada e a sintaxe combinada.

Sensor									
ID	PT				DV		nome_sensor	resp_sensor	local_sensor
	ordem	avi	tsr	imprec	tipo	valor			
1	1	15	20	5	T	30	X1	Alex	LTI
2	2	15	10	5	T	25	X1	Alex	LTI
3	3	15	10	5	T	15	X2	João	LIEC
4	4	20	20	5	T	25	X2	João	LIEC
5	5	30	25	5	T	26	X1	Alex	LTI

Parâmetros Não-Temporais:
Id = Identificação do Sensor
Nome_Sensor = Nome do Sensor
Resp_Sensor = Responsável pelo Sensor
Local_Sensor = Localização do Sensor
Desc_Valor (DV):
tipo = identifica o tipo de atributo
valor = é o valor do sensor gravado no BD
Parâmetros Temporais (PT):
ordem = Ordem de leitura
avi = é o intervalo de validade absoluta
tsr = é o tempo no qual o dado foi gravado no BD
imprec = é a imprecisão acumulada do atributo

Figura 4.2: Tabela com Dados de Sensores

4.4.1 Sintaxe Simplificada

Na sintaxe simplificada, cada declaração em SQL utiliza apenas uma das primitivas de tempo-real, conforme exemplos a seguir:

1. **INITIALIZE**

```
<INITIALIZE AT (TIMESTAMP)>
<[Comandos SQL-99]>
```

A consulta Q1 recupera o nome do sensor (nome_sensor), o valor do sensor (DV. valor) e o intervalo de validade absoluta do mesmo (PT.avi), para o sensor localizado no ambiente LTI (local_sensor = 'LTI'). Observe que a primitiva INITIALIZE AT determina que a transação deverá iniciar no tempo (07/10/2004 16:45:00).

```
(Q1):
INITIALIZE AT(07/10/2004 16:45:00)
SELECT nome_sensor, DV.valor, PT.avi
FROM Sensor
WHERE local_sensor = 'LTI'
```

2. **TERMINATE**

```
<[Comandos SQL-99]>
<TERMINATE AT (TIMESTAMP)>
```

A consulta Q2 recupera o identificador do sensor (id) e o valor do sensor (DV.valor) da Tabela Sensor, para todos os sensores que tenham o valor maior que 20 (DV.valor $>$ 20). Observe que a sua execução deve ser completada no tempo especificado pelo usuário através da primitiva TERMINATE AT, ou seja, (07/10/2004 17:00:00).

```
(Q2):
SELECT id, DV.valor
FROM Sensor
WHERE DV.valor > 20
TERMINATE AT(07/10/2004 17:00:00)
```

3. **COMPUTATION**

```
<[Comandos SQL-99]>
<COMPUTATION(TIMESTAMP)>
```

A consulta Q3 insere na Tabela Sensor uma nova tupla. Observe que, através da primitiva COMPUTATION o usuário determina o tempo computacional da transação, neste caso, a transação deve ser executada em 30 segundos.

```
(Q3):
INSERT into Sensor (id, PT.ordem, DV.tipo, DV.valor,
PT.avi, PT.tsr, PT.imprec, nome_sensor, resp_sensor e
local_sensor) values (6, 4, 'T', 27, 30, 25, 5,
'X2', 'João', 'LIEC')
COMPUTATION(00:30)
```

4. **RETURN_TP**

```
<[Comandos SQL-99]>
<RETURN_TP()>
```

A consulta Q4 é semelhante à consulta Q3 e representa o mesmo comando em SQL-99 da Q3, no entanto através da primitiva RETURN_TP ela recupera o tempo computacional da execução da transação.

```
(Q4):
INSERT into Sensor (id, PT.ordem, DV.tipo, DV.valor,
PT.avi, PT.tsr, PT.imprec, nome_sensor, resp_sensor e
local_sensor) values (6, 4, 'T', 27, 30, 25, 5,
'X2', 'João','LIEC')
RETURN_TC()
```

5. **VALIDATE**

```
<[Comandos SQL-99]>
<VALIDATE <value expression>
```

A consulta Q5 recupera todos os dados temporalmente consistentes da Tabela Sensor, através da primitiva VALIDATE, como definido na Seção 4.2.1.

```
(Q5):
SELECT * from Sensor
VALIDATE
```

4.4.2 Sintaxe Combinada

Na sintaxe combinada, cada declaração em SQL pode utilizar mais de uma primitiva de tempo-real, conforme exemplos a seguir:

1. **PERIOD, INITIALIZE e TERMINATE**

```
<PERIOD (TIMESTAMP)>
<INITIALIZE AT (TIMESTAMP)>
<[Comandos SQL-99]>
<TERMINATE AT (TIMESTAMP)>
```

A consulta Q6 recupera o identificador do sensor (id), o nome do sensor (nome _sensor) e o valor do mesmo (DV.valor) para todos os sensores com o valor superior a 30. Através da primitiva INITIALIZE AT o usuário especifica quando a transação deve ser iniciada a primeira vez. Através da primitiva PERIOD o usuário especifica a periodicidade da execução da mesma. Finalmente, através da primitiva TERMINATE AT, o usuário determinada quando a mesma deve finalizar sua execução.

```
(Q6):
PERIOD (00:10:00)
INITIALIZE AT (10/05/2005 10:30:00)
SELECT id, nome_sensor, DV.valor
FROM Sensor
WHERE DV.valor > 30
TERMINATE AT (10/05/2005 11:30:00)
```

2. **INITIALIZE e TERMINATE**

```
<INITIALIZE AT (TIMESTAMP)>
<[Comandos SQL-99]>
<TERMINATE AT(TIMESTAMP)>
```

A consulta Q7 retorna todos os valores da Tabela Sensor. Ela será inicializada no tempo (01/01/2006 02:00:00), através da primitiva INITIALIZE AT e terminará no tempo (01/01/2006 02:25:00), através da primitiva TERMINATE AT.

```
(Q7):
INITIALIZE AT (01/01/2006 02:00:00)
SELECT * FROM Sensor
TERMINATE AT (01/01/2006 02:25:00)
```

3. **PERIOD, INITIALIZE, TERMINATE e RETURN_TP**

```
<PERIOD (TIMESTAMP)>
<INITIALIZE AT (TIMESTAMP)>
<[Comandos SQL-99]>
<TERMINATE AT (TIMESTAMP)>
<RETURN_TP>()>
```

A consulta Q8 recupera o identificador do sensor (id) e o valor do sensor (DV.valor) da Tabela Sensor. Ela é executada inicialmente no tempo (10/05/2005 10:30:00) e então será executada periodicamente a cada 15 minutos. Observe que, cada vez que ela for executada o tempo de processamento da execução da mesma é recebido através do RETURN_TP().

```
(Q8):
PERIOD (00:15:00)
INITIALIZE AT (10/05/2005 10:30:00)
SELECT id, DV.valor
FROM Sensor
TERMINATE AT (10/05/2005 11:30:00)
RETURN_TP()
```

4. **COMPUTATION e TERMINATE**

```
<[Comandos SQL-99]>
<COMPUTATION(TIMESTAMP)>
<TERMINATE AT (TIMESTAMP)>
```

A consulta Q9 recupera o identificador do sensor (id) e o valor do sensor (DV.valor) da Tabela Sensor. Ela tem um prazo máximo para sua computação através da primitiva COMPUTATION, de 20 segundos. Observe que, ela também terá um prazo máximo através da primitiva TERMINATE AT.

```
(Q9):
SELECT id, DV.valor
FROM Sensor
COMPUTATION (00:20)
TERMINATE AT (10/05/2005 11:30:00)
```

5. **INITIALIZE, COMPUTATION e TERMINATE**

```
<[INITIALIZE AT(TIMESTAMP)>]
<[Comandos SQL-99]>
<COMPUTATION(TIMESTAMP)>
<TERMINATE AT (TIMESTAMP)>
```

A consulta Q10 recupera o identificador do sensor (id), o nome do sensor (nome _sensor) e o valor do mesmo (DV.valor) para todos os sensores com o valor superior a 40. Ela tem um tempo de início através da primitiva INITIALIZE e um prazo máximo para sua computação, através da primitiva COMPUTATION, de 10 segundos. Observe que, ela também terá um prazo máximo através da primitiva TERMINATE AT.

```
(Q10):

INITIALIZE AT (10/05/2005 11:30:00)
SELECT id, nome_sensor, DV.valor
FROM Sensor
WHERE DV.valor > 40
COMPUTATION (00:10)
TERMINATE AT (10/05/2005 11:30:10)
```

4.5 Conclusões

Através da LC-BDTR vários tipos de consultas com restrições temporais podem ser declaradas, tais como: quais transações estão perto de expirar; quais dados estão temporalmente inconsistentes; realizar consultas periodicamente, entre outras. Neste Capítulo apresentou-se a sintaxe da LC-BDTR que estende a SQL-99 com um conjunto de primitivas de tempo-real. Desta forma, as consultas declaram os valores que devem ser manipulados no banco de dados, além de definir o tempo no qual tal manipulação deve ser realizada.

Capítulo 5

Implementação e Estudo de Caso

5.1 Introdução

O desenvolvimento de uma linguagem de consulta, tal como a SQL é uma tarefa de alto custo que envolve uma grande quantidade de recursos e profissionais, de forma que apenas grandes fabricantes de SGBD podem custear.

No caso de linguagens de consultas para banco de dados em tempo-real, que ainda é uma tecnologia em fase de desenvolvimento. Uma alternativa para prover suporte para aplicações em tempo-real, é o desenvolvimento de uma interface que permita simular uma linguagem de consulta de tempo-real, através da integração de uma linguagem de consulta convencional, tal como a SQL-99 e primitivas de tempo-real .

Neste Capítulo é apresentada à descrição geral da interface que dá suporte a LC-BDTR, além de um estudo de caso para validação da mesma.

Este Capítulo é organizado como segue. Na Seção 5.2 apresenta-se a descrição geral da interface. Na Seção 5.3 apresenta-se à interface de conexão. Na Seção 5.4 apresenta-se à interface de consulta. Na Seção 5.5 descreve-se a implementação das primitivas de tempo-real. Finalmente, na Seção 5.6 apresenta-se um estudo de caso para validar a LC-BDTR.

5.2 Descrição Geral da Interface

A interface descrita a seguir dá suporte a elaboração de consultas utilizando a sintaxe da LC-BDTR, permitindo a manipulação de dados armazenados em um SGBD convencional, no entanto, tal manipulação obedece às restrições de tempo. Tal interface disponibiliza duas interfaces: uma interface para realizar conexão com o banco de dados e outra interface para declarar consultas. As mesmas serão definidas com mais detalhes nas Seções 5.3 e 5.4, respectivamente.

A interface foi desenvolvida através da linguagem Java *(JDK 1.4.2)*, que disponibiliza um conjunto de características fundamentais para sua implementação, tais como: portabilidade, segurança, robustez, bibliotecas para acesso a banco de dados e pacotes que disponibilizam primitivas de tempo-real (ECKEL, 2002). O ambiente de desenvolvimento utilizado foi o *Eclipse (3.0)*. O SGBD utilizado foi o *DB2 da IBM*, entretanto qualquer gerenciador baseado no modelo objeto-relacional poderia ter sido usado.

5.2.1 Especificação Java

A linguagem Java disponibiliza um pacote *java.util* que contém uma Classe denominada *Timer* e uma Classe denominada *TimerTask*.

A Classe *Timer* disponibiliza um conjunto de primitivas de tempo e métodos para o agendamento de tarefas. O método *schedule()*, por exemplo, pode receber até três parâmetros: uma tarefa, uma data para a tarefa iniciar e a freqüência que a mesma deve ser realizada.

A Classe *TimerTask* disponibiliza recursos para tratar com tarefas que devam ser tratadas em tempos futuros e/ou executadas repetidas vezes.

A Classe *TimerTask* possui um método denominado *run()*, que é invocado pela Classe *Timer*, através do método *schedule()*. O método *run()* deverá conter o código a ser executado cada vez que a tarefa for escalonada.

5.2.2 Conexão com o SGBD

O JDBC é uma interface (*driver*) que permite que aplicações Java acessem banco de dados e outros arquivos. O JDBC implementa em Java a funcionalidade definida pelo padrão SQL/CLI (ALLEN et al., 2001). É a forma mais rápida e prática para conectar BD a um programa Java.

A linguagem Java disponibiliza no pacote Java.sql, uma classe denominada *DriverManager* que permite: fazer a conexão com o banco de dados; gerenciar o conjunto de drivers JDBC correspondente; controlar o *login* e controlar as mensagens entre o banco de dados e a aplicação.

O exemplo a seguir ilustra a conexão de uma aplicação Java com o SGBD DB2 da IBM.

```
public class DriverManager {

1    public final String DATABASE_URL = "jdbc:db2:";
2    public final String DATABASE_DRIVER = "com.ibm.db2.jcc.DB2Driver";
```

.
.
.

}

5.3 Interface de Conexão

Na Figura 5.1 está ilustrada a definição da conexão de uma aplicação Java com um SGBD, através da interface desenvolvida. O usuário passa como parâmetros o seu nome *(User)*, uma senha *(Password)* e o nome de um banco de dados válido *(Database)*. Caso os parâmetros sejam validados, a conexão com o banco de dados é estabelecida e uma mensagem de conexão estabelecida é exibida no campo *Output* da interface. Caso contrário uma mensagem de erro será exibida no mesmo campo *Output* da interface.

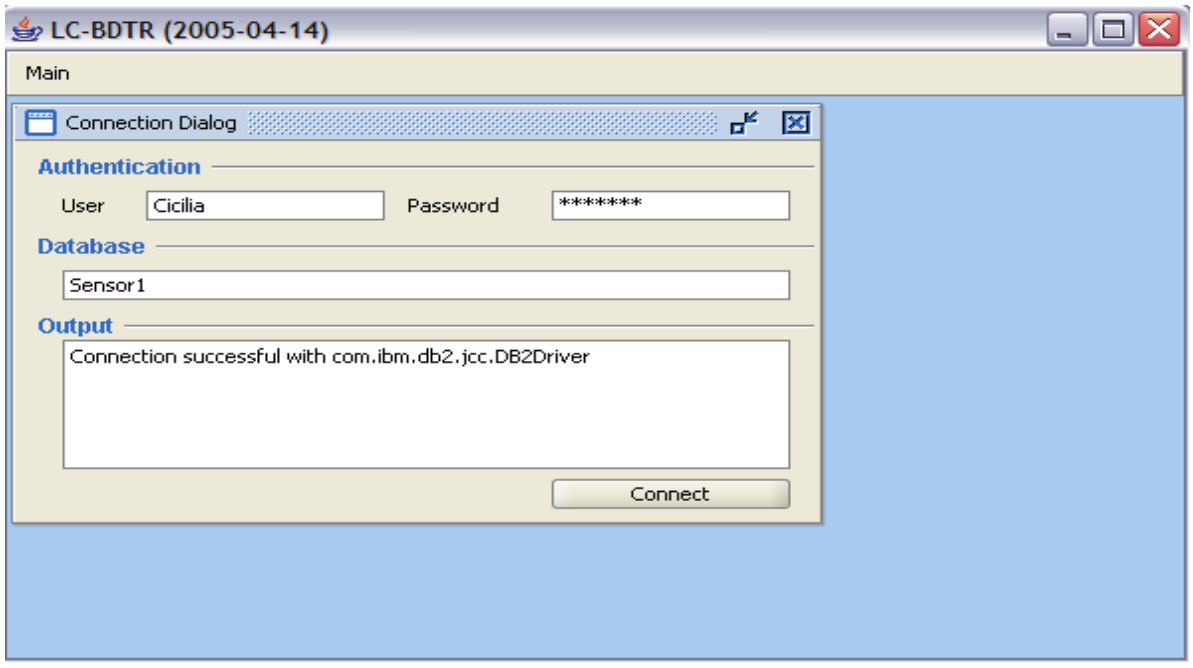

Figura 5.1: Interface de Conexão de uma Aplicação Java com um SGBD

5.4 Interface de Consulta

A interface de consulta permite aos usuários definirem consultas que acessam um banco de dados. A interface de consulta está ilustrada na Figura 5.2. Ela é composta basicamente por duas partes: uma para declarar e processar os comandos *(Query Window)* e uma para visualizar os resultados obtidos após o processamento destes *(Execute Result)*.

A *Query Window* disponibiliza duas janelas: uma para selecionar o banco de dados *(database)* e outra para declarar o comando que deve ser executado (*statement*). Em adição, disponibiliza dois botões: *(Execute e Update)*, onde o *Execute* realiza consultas (select) no banco de dados e o *Update* realiza qualquer outro tipo de manipulação no banco de dados, tais como: *insert, update, create table*, entre outros.

O *Execute Result* disponibiliza uma janela para apresentar os resultados obtidos após o processamento do comando. Na Figura 5.2, está ilustrado como a interface de consulta pode ser utilizada. Observe que na janela *database* o banco Sensor1 foi definido para consulta. Em seguida, na janela (*statement*) o comando foi declarado. Finalmente, após o processamento do comando os resultados são apresentados na janela *Execute Result*.

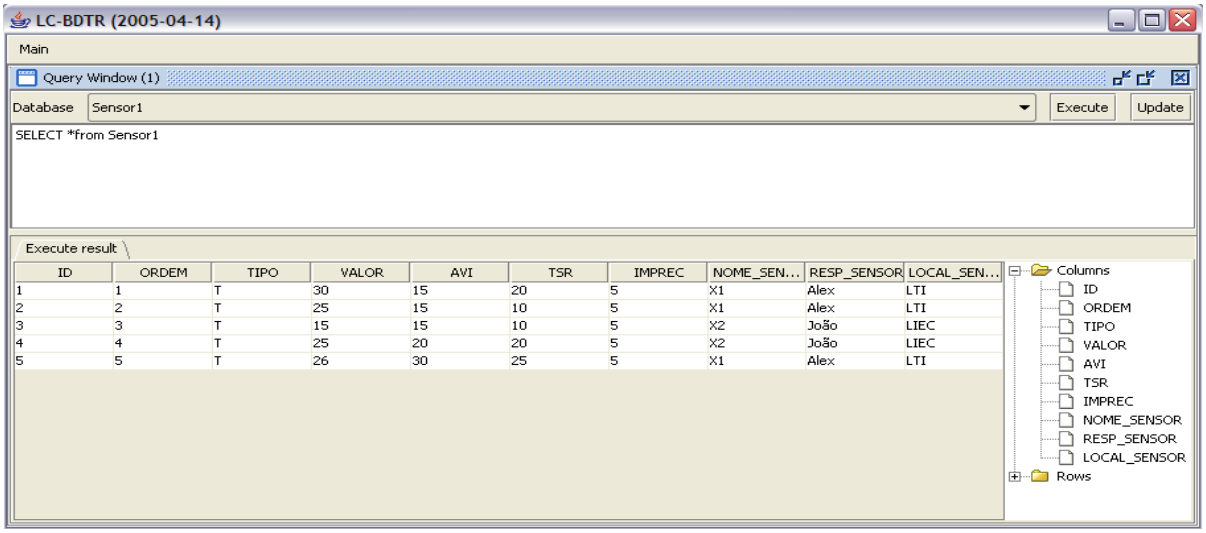

Figura 5.2: Interface de Consulta

5.5 Descrição da Implementação

Para implementar a interface, o primeiro passo foi aplicar o padrão de projeto *Observer*, visando manter o relacionamento entre a interface e as regras de negócio. Desta forma, foi desacoplado o código da interface, do código da lógica de negócio, fazendo com que a comunicação entre eles ocorra através de eventos.

O segundo passo foi diferenciar os comandos SQL-99 dos comandos com restrições temporais específicos da LC-BDTR, sem a existência de um compilador. Portanto, foi criada uma classe que se comporta como um interpretador, que recebe um comando, o interpreta, visando detectar se o mesmo é um comando da SQL-99 ou da LC-BDTR. Caso seja da LC-BDTR, a mesma extrai as informações temporais necessárias para execução do comando.

Para cada tipo de comando é definido um método para a execução do mesmo. A seguir, será detalhado o processo de implementação para cada primitiva temporal. Para maiores detalhes da implementação da interface veja as principais classes no Apêndice B. As primitivas a seguir foram implementadas através da Classe SQL descrita no Apêndice B.1.1.

5.5.1 INITIALIZE

A primitiva INITIALIZE é baseada em duas classes Java *Timer e TimerTask*. O *Timer* é a classe que agenda a execução do *TimerTask*. O agendamento é baseado na data extraída a partir do *timestamp* da sintaxe desta primitiva obtido na linha 1 do código a seguir.

```
1    Initialize initialize = new Initialize(commands.get(3).toString(),
     query, type, id);
2    Timer timer = new Timer(true);
3    timer.schedule(initialize,date);
```

Na 1^a linha, foi declarado um objeto Initialize. Na 2^a linha, foi declarado um objeto *Timer*. E na 3^a linha, foi agendada a execução do objeto Initialize. A relevância deste trecho está no fato de apresentar a característica fundamental da primitiva, onde o tempo de início para começar a execução de uma transação é assegurada.

5.5.2 TERMINATE

A execução de comandos com esta primitiva requer a validação da data que o comando começou a executar, obtida através do *timestamp* da mesma. O trecho de código essencial para o funcionamento dessa primitiva é:

```
1  Calendar end = GregorianCalendar.getInstance();
2  if(dateTerminate.before(end.getTime())){
```

Na 1^a linha, é marcado o tempo de término da transação. Na 2^a linha, ocorre a validação temporal do comando. Para isso, o tempo de término do comando é comparado com o *timestamp* passado como parâmetro na primitiva. Se ele for posterior, o comando é avaliado em falso, caso contrário ele é avaliado em verdadeiro.

5.5.3 PERIOD

A sintaxe desta primitiva vem acompanhada de outras duas primitivas, o INITIALIZE e o TERMINATE. Desta forma, foi necessário criar uma classe *Period* que é um *TimerTask*.

Como o *PERIOD* indica a freqüência na qual o comando deve ser executado, ao agendar esta tarefa no *Timer*, informa-se a freqüência, que é extraída a partir da sintaxe da primitiva *PERIOD*. O trecho de código essencial para o funcionamento dessa primitiva é:

```
1  initialize = dateFormat.parse(commands.get(4).toString() + " " +
   commands.get(5).toString());
2  Timer timer = new Timer(true);
3  Period period = new Period(commands.get(6).toString(),query, type,id,
   timer);
4  Integer hour = (Integer)commands.get(1);
5  Integer min = (Integer)commands.get(2);
6  Integer sec = (Integer)commands.get(3);
7  long frequency = ((hour.longValue()*3600)
   + (min.longValue()*60) + sec.longValue()) * 1000;
8  timer.schedule(period,initialize,frequency);
```

A 1^a linha desta primitiva é semelhante a 1^a linha da primitiva INITIALIZE, definido na Seção 5.5.1. Na 2^a linha é declarado um objeto *Timer*. Na 3^a linha, é declarado um objeto *period*. Da 4^a a 6^a linha, as variáveis para hora, minuto e segundo são declaradas. Na 7^a linha, a freqüência com que o comando deve ser executado é calculado. E finalmente, na 8^a linha, uma tarefa é agendada através do objeto *Timer*. Esse agendamento requer como parâmetro a data de início *(initialize)*, o *TimerTask* que deve ser executado *(period)* e a freqüência da execução *(frequency)*.

5.5.4 COMPUTATION

Para implementar esta primitiva é necessário registrar o tempo de início e o tempo de término e comparar a diferença desses dois valores com o valor passado pela mesma. O trecho de código essencial para o funcionamento dessa primitiva é:

```
1    begin = GregorianCalendar.getInstance();
2    statement = DatabaseManager.connection.createStatement();
3    statement.getConnection().setAutoCommit(false);
4    ResultSet rs = statement.executeQuery(commands.get(3).toString());
5    end = GregorianCalendar.getInstance();
6    computation = (Calendar) begin.clone();
7    computation.add(Calendar.MINUTE,((Integer)commands.get(1)).intValue());
8    computation.add(Calendar.SECOND,((Integer)commands.get(2)).intValue());
9    if (computation.before(end)){
```

Na 1^a linha, é marcado o tempo de início do processamento do comando. Da 2^a a 4^a linha, é gerenciado a conexão com o banco de dados. Na 5^a linha, é marcado o tempo de término do processamento do comando. As linhas 6, 7 e 8 foram utilizadas para viabilizar a implementação desta primitiva. Para isso, foi criando um clone do tempo de início e adicionado a esse tempo os minutos e segundos do *timestamp* da primitiva. Com isso, foi obtida a data limite na qual essa computação poderia terminar para obedecer ao tempo de computação requerido. Desta forma, na 9^a linha, compara-se o tempo de término com o tempo limite. Se o tempo limite *(computation)* for anterior ao tempo de fim *(end)*, então o comando não é validado, já que sua computação ultrapassou o tempo requerido. Se o tempo limite for posterior, então o comando é validado.

5.5.5 RETURN_TP

Para implementar esta primitiva é necessário registrar o tempo de início e tempo de fim da execução do comando e subtrair tais tempos. Visando obter o tempo de processamento do mesmo. O trecho de código essencial para o funcionamento dessa primitiva é:

```
1  begin = Calendar.getInstance().getTime();
2  int code = statement.executeUpdate(sql);
3  end = Calendar.getInstance().getTime();
4  result = "Result code is " + code +"\n tp = " + (end.getTime() -
   begin.getTime()) + "milisegundos";
```

Na 1^a linha, é marcado o tempo de início do processamento do comando. Na 2^a linha, o comando é executado. Na 3^a linha, é marcado o tempo de término do comando. Finalmente, na 4^a linha, a subtração dos tempos é realizada e o tempo de processamento do comando é obtido.

5.5.6 VALIDATE

Para implementar esta primitiva os atributos *avi e tsr* são recuperados. Ou seja, essa primitiva é restrita apenas para realizar consultas que retornam tuplas contendo todas as colunas das tabelas, visando garantir a presença das colunas *avi e tsr*. Portanto, o funcionamento desta primitiva consiste na varredura das tuplas recuperadas e validação dos dados com base nos valores de *avi e tsr*. O trecho de código essencial para o funcionamento dessa primitiva é:

```
1  if(end.getTime().getTime() - tsr.getTime() <= avi_inMillis)
```

Na linha 1, o valor do *tsr* é subtraído do valor do tempo corrente e comparado ao valor do *avi*, visando verificar a validade temporal dos dados.

5.6 Estudo de Caso

Com o objetivo de validar a extensão da linguagem de consulta proposta neste trabalho foi escolhido como estudo de caso redes de sensores. Este estudo de caso foi escolhido por apresentar um grande número de informações que mudam com o passar do tempo, onde as consultas com restrições temporais são bastante importantes. A aplicação precisa ler dos sensores os atributos de temperatura e luminosidade de um determinado ambiente.

No entanto é importante ressaltar que algumas aplicações precisam garantir tanto a consistência lógica quanto a consistência temporal do banco de dados. Desta forma, fica visível a necessidade do desenvolvimento de uma linguagem de consulta para aplicações de tempo-real, onde os usuários estabelece parâmetros temporais de acordo com as necessidades dos mesmos.

5.6.1 Redes de Sensores

Diante das várias aplicações que podem ser beneficiadas com o desenvolvimento de um SGBD-TR foram destacadas as redes de sensores. Através destas, o mundo real está se tornando uma plataforma de processamento, tais como aplicações: médicas e científicas, controle de tráfego aéreo, controle de sistemas de manufatura, sistemas de transportes e sistemas de sensoriamento, estão ficando cada vez mais automatizados (NETO et al., 2004).

Uma rede de sensores consiste de um grande número de dispositivos conectados por interfaces de comunicação que possam se comunicar entre si através de protocolos de roteamento. Cada sensor tem uma CPU de propósito-geral para desempenhar processamento e um pequeno espaço de armazenamento. Alguns sensores são fixos e outros são móveis, podendo estar sempre conectado a rede ou serem intermitentes.

As redes de sensores são capazes de coletar sinais físicos tais como calor, luz, som, pressão, magnetismo, ou um simples movimento de um objeto. Estas propriedades, agregadas a capacidade de processamento, são úteis para um grande número de aplicações. Nas redes de sensores convencionais, os sensores são pré-programados para enviarem os dados coletados para um servidor de banco de dados central, permitindo análises e consultas off-line. Porém, o fluxo contínuo de uma grande quantidade de dados incorre em um custo muito elevado, principalmente considerando as restrições de recursos físicos encontradas nestas plataformas.

Dentre as restrições pode-se, segundo (YAO; GEHRKE, 2003): (a) comunicação: as redes conectando sensores perdem pacotes freqüentemente; (b) consumo de energia: os sensores têm fornecimento limitado de energia e assim a conservação de energia é um fator importante; (c) processamento: tais dispositivos possuem poder limitado de processamento e tamanho de memória e; (d) incerteza nos dados: sinais detectados em sensores

físicos herdam incertezas podendo gerar dados imprecisos (BONNET et al., 1999).

Com base nestas restrições, torna-se bastante atrativo, reduzir a quantidade do fluxo de dados nas redes por processamento local. Assim, uma parte do processamento pode ser realizado nos sensores, reduzindo o consumo de energia por diminuir o tráfego de dados e, conseqüentemente, aumentando o tempo de vida da rede (BONNET; SESHADRI, 2000).

5.6.2 Dados e Transações para Redes de Sensores

As transações de sensores possuem rótulos de tempo. Seus resultados devem ser atualizados freqüentemente, já que dados de sensores se tornam rapidamente inválidos devido às restrições temporais. Consultas longas, que recomputam os resultados das transações de sensores periodicamente, são uma possibilidade para manter estes resultados atualizados.

Inerentes aos dados coletados por sensores estão às incertezas sobre o real valor destes. Por exemplo, considere um sensor de temperatura que registra uma temperatura estimada t da Temperatura corrente T; $[T - t, T + t]$. Como alternativa para diminuir esta imprecisão, diversos nós são dispostos com a mesma funcionalidade, i.e., medem o mesmo fenômeno físico. Desta forma, obtém-se como resposta a fusão dos dados, onde estas possuem uma variação menor que as leituras de nós individuais.

Há três tipos de consultas nas redes de sensores: (1) consulta de dados históricos: estas são consultas agregadas de dados históricos obtidos da rede; (2) consultas instantâneas: consultas na rede em um dado instante de tempo e; (3) consultas longas: consultas na rede em um dado intervalo de tempo. Para realizar as consultas nas redes de sensores foi utilizada a linguagem SQL-99.

5.6.3 A Aplicação

O domínio da aplicação são as redes de sensores, uma vez que essas têm sido bastante utilizadas em uma série de aplicações, conforme citado na 5.6.1.

Na Figura 5.3, está ilustrado um cenário de um ambiente de aplicação genérico de redes de sensores. Ele é composto por: uma Rede de Sensores, LC-BDTR, SGBD-OR e Usuários.

Tem-se a seguinte descrição dos eventos deste estudo de caso:

1. Rede de Sensores - Os sensores coletam dados do ambiente e armazenam por um determinado período;

2. SGBD-OR - Periodicamente os dados dos sensores são enviados para o SGBD, uma vez que os sensores possuem poder limitado de armazenamento;

3. Usuário - O usuário, por sua vez, faz declarações de consultas com restrições temporais;

4. LC-BDTR - Através da LC-BDTR, o usuário declara suas consultas, baseado na SQL-99 e nas primitivas temporais;

5. A declaração, processamento e resultado das consultas é disponibilizado através da interface desenvolvida que tem com base a sintaxe da LC-BDTR.

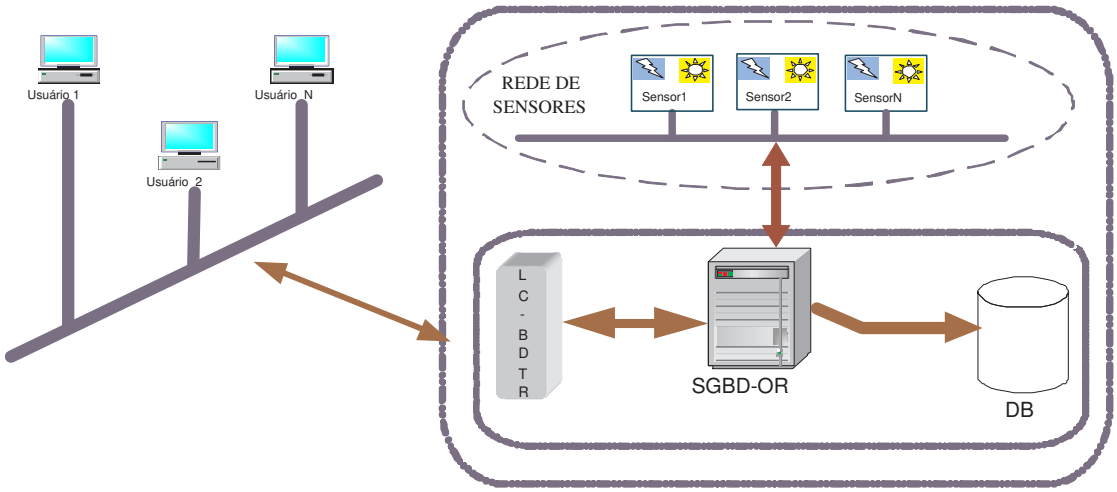

Figura 5.3: Ilustração de um Cenário para um Ambiente de Aplicação com Redes de Sensores

Desta forma, pode-se definir uma rede de sensores que consiste de um grande número de dispositivos conectados entre si. Periodicamente, dados gerados pelos sensores são transferidos para o armazenamento permanente no banco de dados. Os usuários interagem com a LC-BDTR, para escrever consultas ou programas de aplicação, que acessam diretamente os sensores ou o banco de dados, de acordo com a necessidade dos mesmos. Através do estudo de caso, foi possível especificar restrições temporais e lógicas para os dados e transações.

5.6.4 Dados dos Sensores

Os dados em uma rede de sensores possuem um tempo de vida útil limitado. No caso da aplicação em questão, tal restrição temporal é considerada, tanto no caso da recuperação quanto no processamento dos mesmos.

Os dados de sensores do estudo de caso são de sensores de luminosidade e temperatura, onde esses dados são lidos de três sensores que estão monitorando um determinado ambiente. Os dados, relativos às medidas de luminosidade e temperatura, são armazenados no

banco de dados a cada 30 segundos. É comum que dados lidos a partir de sensores sejam perdidos. O esquema para obtenção dos dados dos sensores é ilustrado na Figura 5.4.

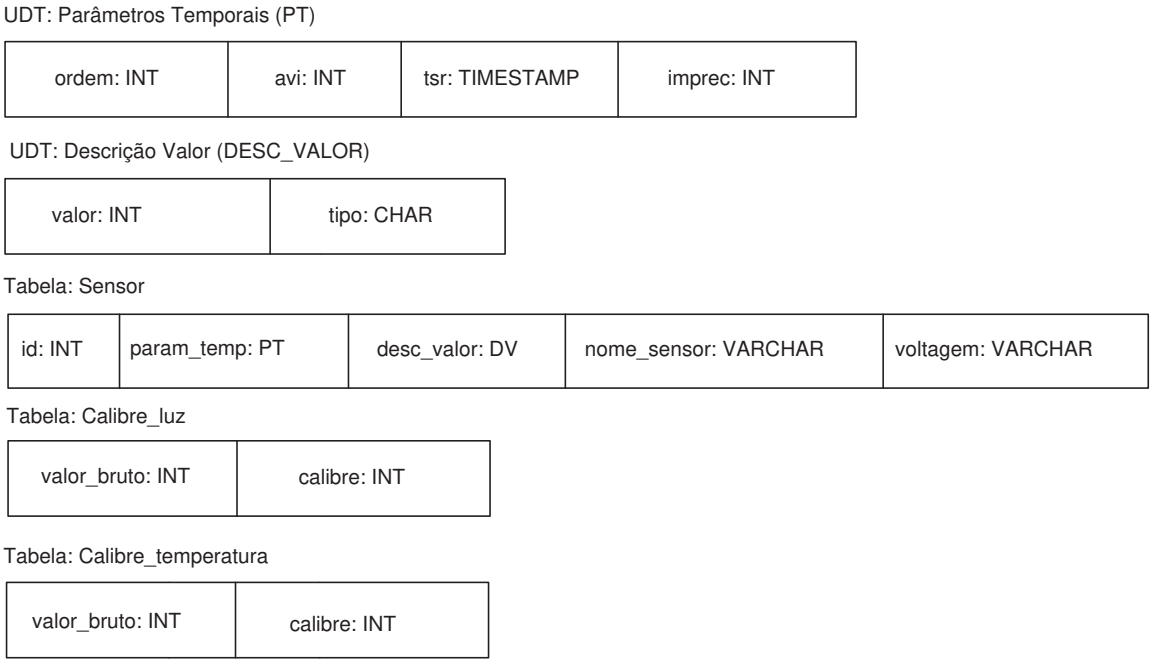

Figura 5.4: Esquemas das Tabelas com os Dados dos Sensores

O SGBD-OR é baseado no modelo de dados objeto-relacional, que usa um conjunto de tabelas para representar tanto os dados quanto o relacionamento entre eles. Cada tabela possui múltiplas colunas e cada uma possui um nome único. Na Figura 5.5 é apresentada uma Tabela denominada Sensor_Tabela, que armazena itens de dados adquiridos de um determinado sensor. A tabela possui 9 colunas das quais quatro (ordem, avi, tsr, imprec) correspondem a atributos temporais e cinco (id, tipo, valor, nome_sensor, voltagem) correspondem a atributos não-temporais. Como citado no Capítulo 3, a SQL-99 permite que os usuários definam seus próprios tipos (UDT). Portanto, neste trabalho os atributos temporais são definidos através de um UDT, que pode ser utilizado por qualquer tabela do banco de dados que use dados temporais.

- Criando o UDT param_temp

```
CREATE TYPE param_temp AS OBJECT (
      ordem INTEGER,  - ordem de leitura
      avi   TIME,   - intervalo de validade absoluta
      tsr   TIMESTAMP,   - rótulo de tempo
      imprec  INTEGER,   - limite de imprecisão)
```

- Criando o UDT desc_valor

```
CREATE TYPE desc_valor AS OBJECT (
      tipo  VARCHAR(30),  - tipo
      valor INTEGER, - valor)
```

Os outros tipos também devem ser criados através de UDT para posteriormente serem utilizados para a criação de tabelas.

- Criando tabela Sensor_Tabela

```
CREATE TABLE Sensor_Tabela(
      id  INTEGER,  - identificação
      pt  PARAM_TEMP, - UDT definido
      valor DESC_VALOR, - UDT definido
      nome_sensor  VARCHAR(30),  - nome do sensor
      voltagem  INTEGER, - voltagem do sensor
      ) NESTED TABLE desc_valor store AS desc_valor_tab;
```

Depois de definir os UDT e as tabelas, tem-se a descrição de cada atributo da tabela Sensor_Tabela, como mostrado a seguir:

- *id* - identificação do nó sensor que fez uma leitura;
- *ordem* - o número de seqüência de registro do sensor. O atributo ordem também serve para verificar a consistência relativa dos dados, visando garantir que os dados foram lidos ao mesmo tempo, ou seja, os dados relativos de cada sensor (luminosidade e temperatura) serão considerados corretos se os mesmo tiverem sido lidos ao mesmo tempo;
- *avi* - é o intervalo de validade absoluta, ou seja, é o intervalo de tempo no qual o dado é considerado válido temporalmente;
- *tsr* - é o rótulo de tempo, isto é, o tempo no qual o dado foi gravado no sistema;
- *imprec* - é a quantidade de imprecisão acumulada para o atributo. A definição dessa estrutura visa armazenar todas as informações pertinentes a um item de dado, tanto do ponto de vista lógico quanto temporal;
- *tipo* - é o tipo do sensor, neste caso são sensores que capturam informações de temperatura e luz;
- *valor* - é o valor lido do sensor, seja valor de luz ou valor de temperatura;
- *nome_sensor* - é o atributo que representa o nome do sensor;

- *voltagem* - é a voltagem de cada sensor envolvido.

Sensor_Tabela

id	Param_Temp				Desc_Valor		nome_sensor	voltagem
	ordem	avi	tsr	imprec	tipo	valor		
1	1	30	2005.05.11 10:20:00	5	T	30	X1	220
					L	25		
2	2	30	2005.05.11 10:25:00	5	T	33	X2	220
					L	28		
3	3	30	2005.05.11 10:40:00	5	T	35	X3	220
					L	31		
1	1	30	2005.05.11 11:00:00	5	T	30	X1	220
					L	25		
2	2	30	2005.05.11 11:00:00	5	T	33	X2	220
					L	28		
3	3	30	2005.05.11 11:00:00	5	T	35	X3	220
					L	31		

Parâmetros Não-Temporais
Id = identificação do Sensor
Nome_Sensor = nome do sensor
Voltagem = voltagem do sensor

Parâmetros Temporais (PT):
ordem = ordem de leitura
avi = é o intervalo de validade absoluta
tsr = é o tempo no qual o dado foi gravado no BD
imprec = é a imprecisão acumulada do atributo
Desc_Valor (DV):
tipo = identifica o tipo do atributo
valor = é o valor do sensor gravado no BD

Figura 5.5: Sensor_Tabela

Considerando a Tabela Sensor_Tabela, ilustrada na Figura 5.5 e sua definição em SQL-99, a sintaxe estendida da SQL-99 adicionada de primitivas de tempo-real, serão mostradas através de exemplos.

5.6.5 Exemplos de Consultas

1. **INITIALIZE** - Recupere todos os identificadores de sensores (id), tempos (PT.tsr) e o valor de luminosidade que ultrapasse de 20 e que esta consulta seja inicializada no tempo (25/05/2005 14:32:50), através da primitiva temporal INITIALIZE AT.

   (Q1):

   ```
   INITIALIZE AT (25/05/2005 14:32:50)
   SELECT  id, PT.tsr, DV.valor, DV.tipo
   FROM Sensor_Tabela
   WHERE DV.tipo = 'L' AND DV.valor > 20
   ```

   A consulta Q1 recupera o identificador do sensor (id), o rótulo de tempo no qual o valor foi gravado no banco de dados (DV.tsr), o valor do sensor (DV.valor), e tipo do

sensor (DV.tipo) da Tabela Sensor_Tabela. Onde tem que ser do tipo luminosidade (DV.tipo = 'L') e o valor tem que ser superior a vinte (DV.valor > 20). Observe que a primitiva INITIALIZE AT determina que a transação deverá iniciar no tempo (25/05/2005 14:32:50), conforme ilustrada na Figura 5.6.

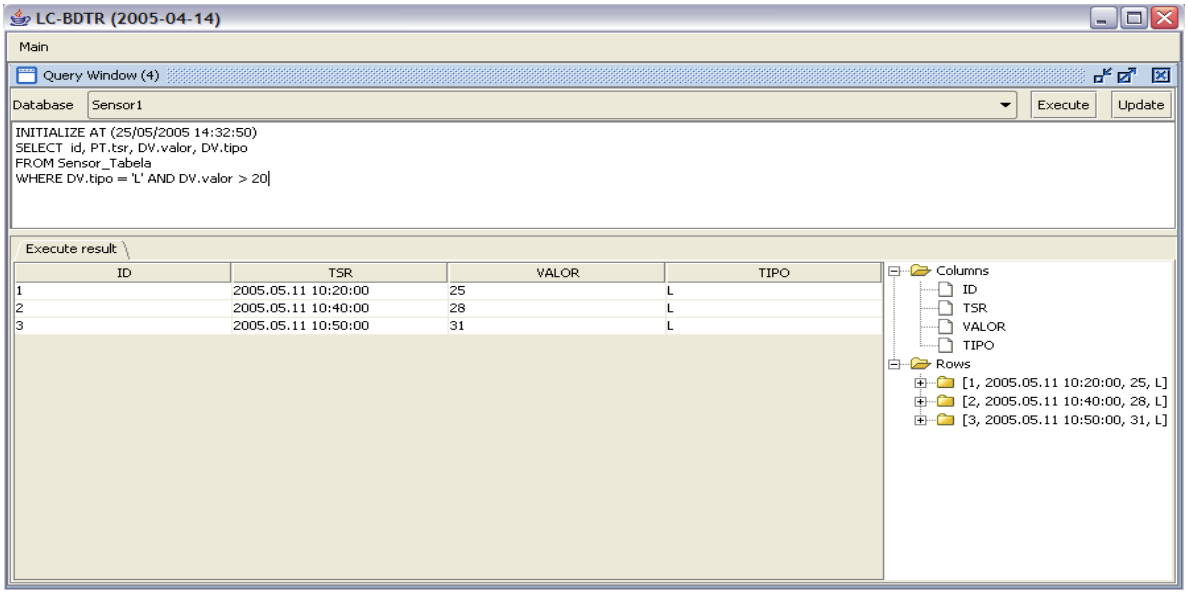

Figura 5.6: Consulta com a Primitiva INITIALIZE

2. **TERMINATE** - Recupere o valor médio do valor do sensor (DV.valor), do tipo luminosidade (DV.tipo), do Sensor cujo id é igual a um (id=1), onde seus valores tenham sido coletados no tempo 2005.05.11 10:10:00 - 2005.05.11 10:40:00 (PT.tsr) e que esta consulta seja completada no tempo (25/05/2005 15:12:50), através da primitiva temporal TERMINATE AT.

```
(Q2):

SELECT AVG(DV.valor)
FROM Sensor_Tabela
WHERE DV.tipo = 'L'
AND id = 1
AND PT.tsr > '2005.05.11 10:10:00'
AND PT.tsr < '2005.05.11 10:40:00'
TERMINATE AT (25/05/2005 15:12:50)
```

A consulta Q2 recupera o valor médio do sensor cujo id é igual a 1 (id = 1), do tipo luminosidade (DV.tipo = 'L'), que está dentro de um intervalo de tempo de 2005.05.11 10:10:00 - 2005.05.11 10:40:00. Observe que a cláusula TERMINATE

AT determina que a transação deverá ser concluída no máximo em (25/05/2005 15:12:50), conforme ilustrada na Figura 5.7.

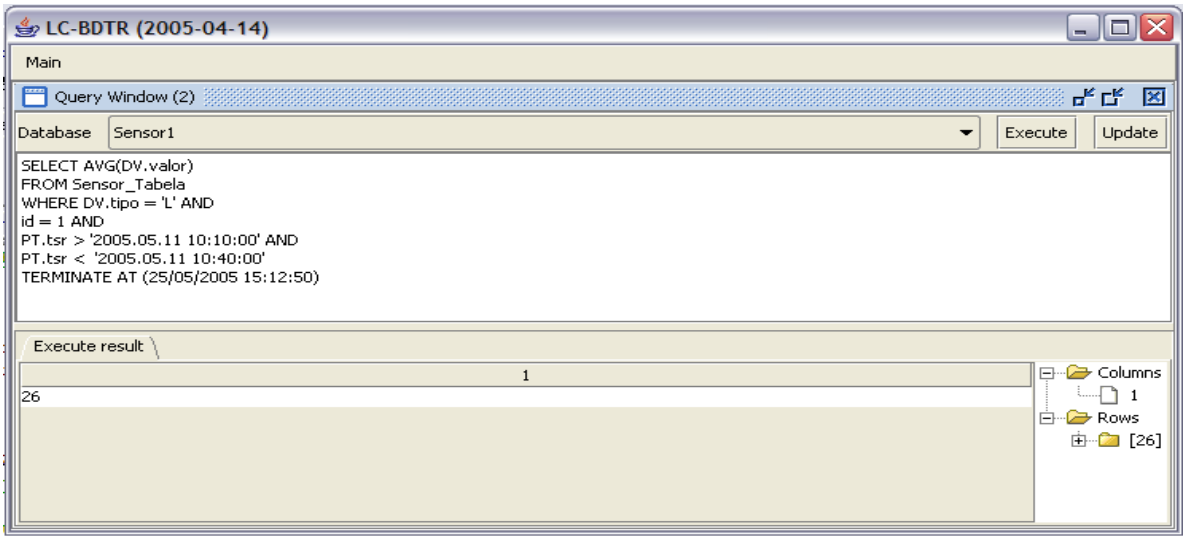

Figura 5.7: Consulta com a Primitiva TERMINATE

3. **PERIOD** - Recupere o identificador do sensor (id), o nome do sensor (nome_sensor) e o valor do mesmo (DV.valor) para todos os sensores com o valor superior a 30.

```
(Q3):
PERIOD (00:00:10)
INITIALIZE AT (25/05/2005 15:39:40)
SELECT id, nome_sensor, DV.valor
FROM Sensor_Tabela
WHERE DV.valor > 30 AND DV.tipo = T
TERMINATE AT (25/05/2005 15:40:00)
```

A consulta Q3 através da primitiva INITIALIZE AT o usuário especifica quando a transação deve ser iniciada a primeira vez (25/05/2005 15:39:40). Através da primitiva PERIOD o usuário especifica a periodicidade (00:00:10) com que a mesma deve ser executada. Finalmente, através da primitiva TERMINATE AT o usuário determinada quando a mesma deve finalizar sua execução (25/05/2005 15:40:00), conforme ilustrada na Figura 5.8.

Uma análise de desempenho das primitivas temporais foi realizada, com o intuito de validar a LC-BDTR e testar a interface. Para isso, foi elaborada diversas consultas com diferentes primitivas temporais para a aplicação definida, a fim de verificar se as primitivas temporais estavam cumprindo com seus prazos. Algumas das situações de

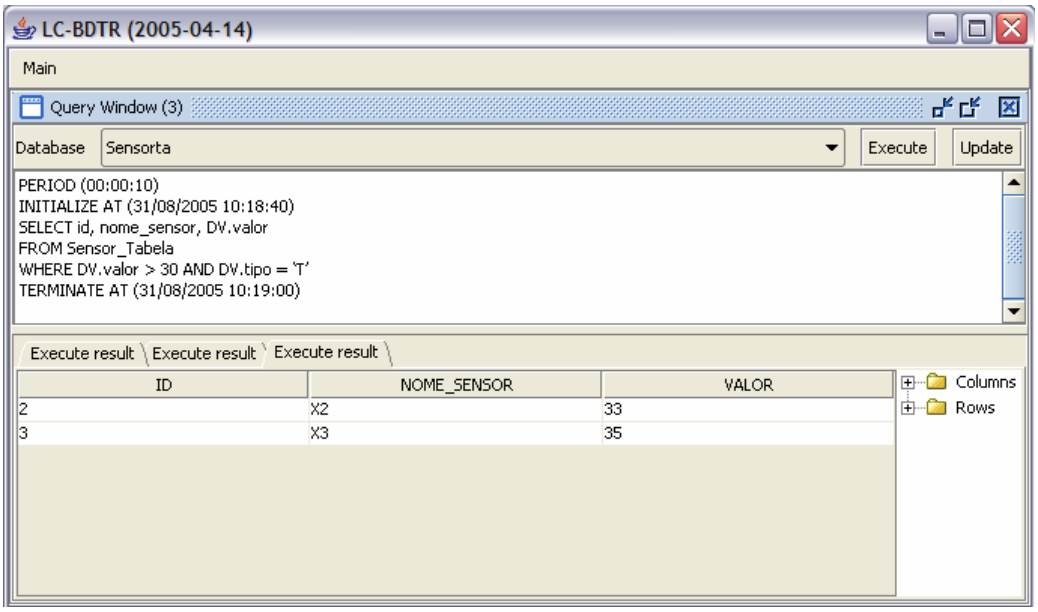

Figura 5.8: Consulta com a Primitiva PERIOD

consultas com a sintaxe da LC-BDTR (simplificada e combinada) foram consideradas e seus resultados produzidos atenderam os objetivos definidos na Seção 1.2. Portanto, as consultas declaradas, seus processamentos e resultados foram executados com sucesso, de forma que o desempenho em geral foi bastante satisfatório.

5.7 Conclusões

Os exemplos de consultas definidos no Capítulo 4 poderão ser executados através da interface implementada neste trabalho, permitindo que os desenvolvedores incluam as cláusulas do SQL-99 sobre aplicações de banco de dados em tempo-real que necessitam manter a integridade lógica e temporal dos dados e transações. Além de poder recuperar, atualizar, inserir dados com características temporais através de consultas SQL-99, utilizando o que está disponível para manipular os dados.

Neste Capítulo, apresentou-se o desenvolvimento da interface e como as primitivas da LC-BDTR foram implementadas. E através de um estudo de caso no contexto de redes de sensores foi possível validar a sintaxe da LC-BDTR e testar a interface implementada.

Capítulo 6

Conclusões

Neste trabalho apresentou-se uma sintaxe para a declaração de consultas de tempo-real, denominada Linguagem de Consulta para Banco de Dados em Tempo-Real (LC-BDTR). Tal sintaxe é uma extensão da SQL-99 adicionada de primitivas de tempo-real.

Também foi desenvolvida uma interface para permitir a declaração e processamento de consultas utilizando a LC-BDTR. Esta interface pode ser utilizada em aplicações que precisem tratar com restrições temporais. Ela foi implementada através da linguagem de programação Java e do SGBD DB2 da IBM. A linguagem Java foi escolhida por ser uma linguagem multiplataforma, e disponibilizar mecanismos para tratar com restrições temporais das aplicações.

Para testar a interface e validar a LC-BDTR foi desenvolvida uma aplicação para redes de sensores, conforme apresentado na Seção 5.6.3. Atualmente a aplicação está sendo ampliada, visando identificar novas primitivas de tempo-real para otimizar a LC-BDTR. As primitivas de tempo-real também estão sendo implementadas, pelo Grupo BDSensor (Projeto financiado pelo CNPq), baseada na Especificação Java para Tempo-Real (SUN MICROSYSTEMS, 2003), utilizando máquina virtual do Jamaica (JAMAICAVM, 2005). Com isso, será permitida a avaliação do desempenho da interface, comparando-se as duas implementações.

6.1 Contribuições

A relevância deste trabalho é disponibilizar a LC-BDTR, que pode ser utilizada em aplicações que necessitem manter a integridade temporal dos dados e garantir as restrições temporais das transações. Também disponibilizar uma interface que possibilite a declaração e o processamento da mesma.

A interface permite reduzir o custo de desenvolvimento de aplicações que devem processar dados e transações com restrições temporais. Observe que através dela é possível

desenvolver aplicações, convencionais ou não, sem a necessidade de adquirir um SGBD-TR, e as aplicações existentes podem ser estendidas para tratar com restrições temporais.

Portanto, através deste trabalho, duas contribuições na área de SGBD-TR são destacadas:

1. Uma sintaxe de uma linguagem de consulta para processamento em tempo-real, a LC-BDTR;

2. Uma interface que permite a declaração e o processamento da LC-BDTR.

A implementação atual da interface garante o funcionamento correto da sintaxe da LC-BDTR, como citado no Capítulo 4, para transações do tipo *suave* e *firme*. No entanto, considerando que a máquina virtual não executa em tempo-real, as transações do tipo *hard* não podem ser garantidas. Portanto, pesquisas precisam ser desenvolvidas visando incorporar novos recursos à LC-BDTR e à interface no sentido de aperfeiçoá-la.

6.2 Perspectivas Futuras

Durante o desenvolvimento deste trabalho, foram identificados alguns aspectos que podem ser melhor explorados ou estendidos a partir do estudo apresentado nesta dissertação.

Considerando o exposto na Seção 6.1, faz-se necessário estender a LC-BDTR visando adaptá-la a nova versão da SQL-99, além de utilizar os novos recursos disponibilizados através do JamaicaVM (JAMAICAVM, 2005), que garante o processamento de transações do tipo *hard*. A seguir, destacam-se alguns tópicos de pesquisa que precisam ser realizados:

1. Substituir o interpretador de comandos da interface por um compilador, que faça análise sintática e semântica da sintaxe da LC-BDTR, visando otimizar ainda mais o tempo gasto na declaração de consultas e/ou realizando a verificação de possíveis erros da sintaxe da linguagem;

2. Estender a aplicação para rede de sensores inteligentes, a fim de realizar o estudo de caso com dados reais coletados do ambiente;

3. Desenvolver uma interface gráfica com o intuito de melhorar a usabilidade da linguagem, por exemplo, disponibilizando as palavras reservadas da linguagem e impedindo a declaração de palavras não existentes na mesma;

4. Estender a interface para que o usuário defina o SGBD com o qual deseja fazer a conexão, de forma que não ficará limitada a nenhum SGBD.

A implementação atual da interface contemplou o funcionamento correto das principais primitivas de tempo-real definidas no Capítulo 4.

Referências Bibliográficas

9075-1:1999, A. Framework (sql/framework. *Information technology–Database languages–SQL–Part 1: Framework (SQL/Framework)*, September 1999.

9075-2:1999, A. Foundation (sql/foundation). *Information technology–Database languages–SQL–Part 2: Foundation (SQL/Foundation)*, September 1999.

9075-3:1999, A. Call- level interface (sql/cli). *Information technology – Database language – SQL – Part 3: Call- Level Interface (SQL/CLI)*, September 1999.

9075-4:1999, A. Persistent stored modules (sql/psm). *Information technology – Database language – SQL – Part 4: Persistent Stored Modules (SQL/PSM)*, September 1999.

9075-5:1999, A. Host language bindings (sql/bindings). *Information technology–Database languages – SQL– Part 5: Host Language Bindings (SQL/Bindings)*, September 1999.

ALLEN, P. R. et al. *J2EE Unleashed*. 1. ed. 2001. 138-179 p.

BONNET, P. et al. Query Processing in a Device Database Systems. Outubro 1999. Technical report UCB/ERL No. M99/63.

BONNET, P.; SESHADRI, P. Device Database Systems. *Proceedings of the International Conference on Data Engineering ICDE'99*, Março 2000. San Diego, CA.

DATABASE, T. E. R.-T. The embedded real-time database. Available in web, http://www2.empress.com. April 2005.

ECKEL, B. *Thinking in Java*. 3rd. ed. 2002. Available in web, http://www.bruceeckel.com, Access in May.

ELMASRI, R.; NAVATHE, S. B. *Sistemas de Banco de Dados*. 4. ed. 2005. 504-543 p.

EMBEDDED, R. Relx embedded. Available in web, http://www.relexus.com. April 2005.

FERNANDES, Y. et al. In: 2004 ieee systems, man and cybernetics conference, 2004, hague. proc. of 2004 ieee systems, man and cybernetics conference. *IEEE Systems, Man and Cybernetics Conference*, v. 6, 2004.

IBM. Db2 universal database. IBM da DB2. Acesso em Maio 2004. Disponível em: <http://www.ibm.com/db2>.

JAMAICAVM. Jamaica. Available in web, http://www.aicas.com. April 2005.

LEITE, C. M. et al. Ql-rtdb: Query language for real-time database. *In: 7th International Conference on Enterprise Information Systems*, v. 1, p. 420–423, Maio 2005.

LEITE, C. M. et al. Processamento de consultas para fluxo contínuo de dados. *Anais VII Simpósio Brasileiro de Automacao Inteligente*, p. 08, Setembro 2005. Http://www.dee.ufma.br/sbailars.

LINDSTROM, J. *Optimistic Concurrency Control Methods for Real-Time Database.* Tese (Doutorado) — Department of Computer Science, University of Helsinki Finland, January 2003.

MORO, M. M. et al. Tvql - temporal versioned query language. *Dexa - 13Th International Conference on Database and Expert Systems Applications*, September 2002. Aix en Provence, France.

NETO, P. R. et al. Uma aplicação de bancos de dados em tempo-real para redes de sensores. *VI Workshop de Tempo Real(WTR) - SBRC*, p. 45–52, 2004.

O´NEIL, P.; O'NEIL, E. *Database Principles, Programming, Performance.* 2. ed. 2000. 173-257 p.

ORACLE. Oracle. Acesso em Maio 2004. Disponível em: <htpp://www.oracle.com.br>.

PERKUSICH, M. *Um Método Baseado em Redes de Petri para a Modelagem de Bancos de Dados para Aplicações em Tempo-Real.* Tese (Doutorado) — Departamento de Engenharia Elétrica, 2000.

PERKUSICH, M.; TURNELL, M. d. F.; PERKUSICH, A. Modelagem de Banco de Dados em Tempo-real. *XIV Simpósio Brasileiro de Banco de Dados - SBBD'99*, 1999.

POSTGRESQL. PostgreSQL. Acesso em Maio 2004. Disponível em: <http://www.postgresql.org.br/>.

PRICHARD, J. *"RTSQL": Extending The "SQL"Standard to Support Real-Time Databases.* Tese (Doutorado) — Department of Computer Science and Statistics, University of Rhode Island, 1995.

RAMAMRITHMAN, K. *Real-Time Databases. Distributed and Parallel Databases.* v. 1, 1993. 199-226 p.

SUDARSHAN, S.; KORTH, F. H.; SILBERSCHATZ, A. *Database System Concepts*. 4. ed. 2001. 109-288 p.

SUN MICROSYSTEMS. *The Real Time Specification for Java*. [S.l.], Acesso em Março 2003. Disponível na Web, http://rtsj.dev.java.net.

TESANOVIC, A. et al. *Embedded databases for embedded real-time systems: A component-based approach*. [S.l.], 2002.

VOSSOUGH, E. *Processing of Continuous Queries Over Infinite Data Streams*. Tese (Doutorado) — Institution University of Wollongong, 2004.

YAO, Y.; GEHRKE, J. E. Query Processing for Sensor Networks. *To appear in the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Janeiro 2003. Asilomar, California.

Apêndice A

Principais Cláusulas do DB2

A.1 Estruturas Básicas da DDL no DB2

A.1.1 CREATE DATABASE

Criar um banco implica na criação de um diretório onde serão armazenados os dados e também, qual gerenciador de banco de dados será utilizado para armazenar as informações.

```
CREATE DATABASE nome_db
     [BUFFERPOOL nome_bp] [INDEXBP nome_bp]
     [AS {WORKFILE|TEMP} [FOR nome_membro]
     [STOGROUP {nome_sg|SYSDEFLT}]
     [CCSID nome_ccsid]
```

De acordo com a sintaxe mostrada anteriormente, tem-se o significado de cada parâmetro utilizado, conforme ilustrado na Tabela A.1.

Tabela A.1: Criar um Banco de Dados

Parâmetro	**Significado**
DATABASE nome_db	Banco de Dados a ser criado.
BUFFERPOOL nome_bp	*Buffer Pool default.*
INDEXBP nome_bp	*Buffer Pool default* para índices.
AS WORKFILE|TEMP	Define tipos especiais de BD.
WORKFILE	Indica que é o BD de arquivos de trabalho de um membro do data *sharing.*

Continua na próxima página

Parâmetro	**Significado**
TEMP	Indica que é um BD para ser utilizada pelas tabelas globais temporárias.
STOGROUP nome_sg	*Storage Group default.*
SYSDEFLT	SG definido na instalação do DB2.
CCSID nome_ccsid	Nome do conjunto de caracteres.

A.1.2 ALTER DATABASE

O comando ALTER DATABASE é utilizado para mudar o valor padrão de uma variável de configuração do próprio banco de dados em tempo de execução.

```
ALTER DATABASE nome_db
     [BUFFERPOOL nome_bp]
     [INDEXBP nome_bp]
     [STOGROUP nome_sg]
     [CCSID nome_ccsid]
```

De acordo com a sintaxe mostrada anteriormente, tem-se o significado de cada parâmetro utilizado, conforme ilustrado na Tabela A.2.

Tabela A.2: Alterar um Banco de Dados

Parâmetro	**Significado**
DATABASE nome_db	Banco de Dados a ser alterado.
BUFFERPOOL nome_bp	*Buffer Poll default.*
INDEXBP nome_bp	*Buffer Poll default* para índices.
STOGROUP nome_sg	*Storage Group default.*
CCSID nome_ccsid	Nome do conjunto de caracteres.

A.1.3 DROP DATABASE

O comando DROP DATABASE remove o banco de dados existente e automaticamente remove o diretório que contém os dados, ressaltando que este comando não pode ser desfeito. E por último, não pode ser executado enquanto conectado ao banco de dados de destino.

```
DROP {ALIAS nome_alias
     |DATABASE nome_db
     |DISTINCT TYPE nome_tipo_usuário RESTRICT
     |FUNCTION nome_fun [(tipo_dado[ AS LOCATOR],...)]
```

```
        RESTRICT
|INDEX nome_índice
|PACKAGE nome_collection.nome_package[VERSION
id_vers]
|PROCEDURE nome_procedure RESTRICT
|SPECIFIC FUNCTION nome_específico_função RESTRICT
|STOGROUP nome_sg
|SYNONYM nome_sunonym
|TABLESPACE [nome_db]nome_tablespace
|TRIGGER nome_trigger
|VIEW nome_view}
```

De acordo com a sintaxe mostrada anteriormente, tem-se o significado de cada parâmetro utilizado, conforme ilustrado na Tabela A.3.

Tabela A.3: Remover um Banco de Dados

Parâmetro	**Significado**
ALIAS	Identifica o ALIAS. Não tem efeito em nenhuma VIEW ou SYNONYM que foi definida usando o ALIAS.
Autorização	OWNER|SYSADM|SYSCTRL.
DATABASE	Identifica o DATABASE. Todos os *tablespaces*, tabelas e índices são dropados.
Autorização	SYSADM|SYSCTRL|DBADM|DBCTRL.
Privilégio	DROP *database*.
DISTINCT TYPE	Tipo definido pelo usuário.
RESTRICT	Não dropa o objeto caso exista algum outro que o utilize.
Autorização	OWNER|SYSADM|SYSCTRL.
Privilégio	DROPIN para o SCHEMA ou todos os SCHEMAS.
FUNCTION	Indica uma função definida pelo usuário.
RESTRICT	Não remove o objeto caso exista algum outro que o utilize.
Autorização	OWNER|SYSADM|SYSCTRL.

Continua na próxima página

Parâmetro	**Significado**
Privilégio	DROPIN para o SCHEMA ou todos os SCHEMAS.
INDEX	Identifica o índice que será dropado.
Autorização	OWNER|SYSADM|SYSCTRL|DBADM .
PACKAGE	Identifica o *package* a ser dropado.
VERSION	Indica a versão do *package*.
Autorização	OWNER|SYSADM|SYSCTRL|DBADM|PACKADM para a coleção ou todas coleções.
Privilégio	BINDAGENT dado pelo OWNER.
PROCEDURE	Identifica a *stored procedure* a ser dropada.
RESTRICT	Não dropa o objeto caso exista algum outro que o utilize.
Autorização	OWNER|SYSADM|SYSCTRL.
Privilégio	DROPIN para o SCHEMA ou todos os SCHEMAS.
SPECIFIC FUNCTION	Indica o nome específico de uma função definida pelo usuário.
RESTRICT	Não dropa o objeto caso exista algum outro que o utilize.
Autorização	OWNER|SYSADM|SYSCTRL .
Privilégio	DROPIN para o SCHEMA ou todos os SCHEMAS.
STOGROUP	Identifica o *storage group* a ser dropado.
Autorização	OWNER|SYSADM|SYSCTRL.
SYNONYM	Identifica o *synonym* a ser dropado.
Autorização	OWNER|SYSADM|SYSCTRL.
TABLE	Identifica a tabela a ser dropada.
Autorização	OWNER|SYSADM|SYSCTRL|DBADM.
TABLESPACE	Identifica o *tablespace* a ser dropada.
Autorização	OWNER|SYSADM|SYSCTRL|DBADM.
TRIGGER	Indica a *trigger* a ser dropada.
Autorização	OWNER|SYSADM|SYSCTRL.
Privilégio	DROPIN para o SCHEMA ou todos os SCHEMAS.

Continua na próxima página

Parâmetro	**Significado**
VIEW	Indica a visão a ser dropada.
Autorização	OWNER|SYSADM|SYSCTRL|DBADM.

A.1.4 CREATE TABLE

O comando CREATE TABLE cria uma tabela nova, inicialmente vazia, no banco de dados atual.

```
   CREATE TABLE nome_tabela{
   (    definição_coluna,...
        [restrição_unicidade,...]
        [restrição_referencial,...]
        [restrição_check,...])
   | LIKE{nome_tabela|nome_view}
        [INCLUDING IDENTITY [COLUMN ATTRIBUTES]]}
   IN {[nome_db] nome_ts/DATABASE nome_db}
   [EDITPROC {nome_programa|NULL}]
   [VALIDPROC {nome_programa|NULL}]
   [AUDIT {NONE|CHANGES|ALL}]
   [OBID inteiro]
   [DATA CAPTURE{NONE|CHANGES}]
   [WITH RESTRICT ON DROP]
   [CCSID {ASCII|UNICODE|EBCDID}]
definição_coluna:
   nome_coluna {tipo_dado_embutido|tipo_dado_usuário}
   [NOT NULL]
   [restrição_unicidade]
   [restrição_referencial]
   [restrição_check]
   [WITH DEFAULT
        [{constante|USER|CURRENT SQLID|NULL|nome_função_cast
                (constante|USER|CURRENT SQLID|NULL)}]
   [GENERATED {ALWAYS|BY DEFAULT} definição_identity]
   [FIELDPROC nome_programa [(constante,...)]
definição_identity:
    AS IDENTITY (
    [START WITH {1|constante}]
    [INCREMENT BY {1|constante}]
```

```
    [CACHE 20|NO CACHE| CACHE inteiro]
    [NO CYCLE|CYCLE]
    [MAXVALUE constante]
    [MINVALUE constante])
tipo_de_dado_embutido:
    {SMALLINT
    |{INTEGER|INT}
    |{DECIMAL|DEC|NUMERIC}[(precisão, [escala])]
    |{FLOAT (inteiro)|REAL|DOUBLE[PRECISION]}
    |{  {CHARACTER|CHAR}[(inteiro)]
        |{CHARACTER|CHAR}VARYING [(inteiro)]
        |{VARCHAR [(inteiro)]}
        FOR {SBCS|MIXED|BIT}DATA}
    |{  {CHARACTER|CHAR} LARGE OBJECT
        |CLOB} [(inteiro{K|M|G})]}
    |{BINARY LARGE OBJECT | BLOB][(inteiro{K|M|G})]}
    |{  {GRAPHIC [(inteiro)]
        |VARGRAPHIC (INTEIRO)
        |DBCLOB [(inteiro{K|M|G})]
    |{DATE|TIME|TIMESTAMP}
    |ROWIND
restrição_de_unicidade:
    [CONSTRAINT nome_restrição] {PRIMARY KEY|UNIQUE}
    (nome_coluna,...)
restrição_referencial:
    [CONSTRAINT nome_restrição] {PRIMARY KEY|UNIQUE}
    (nome_coluna,...) REFERENCES
    nome_tabela [(nome_coluna,...)]
    ON DELETE{RESTRICT, NO ACTION, CASCADE, SET NULL}
restrição_check:
    [CONSTRAINT nome_restrição] CHECK (condição_check)
```

De acordo com a sintaxe mostrada anteriormente, tem-se o significado de cada parâmetro utilizado, conforme ilustrado na Tabela A.4.

Tabela A.4: Criar uma Tabela

Parâmetro	**Significado**
nome_tabela	Nome da tabela a ser criada.
nome_coluna	Nome da coluna.
NOT NULL	Restrição para valores não nulos.
WITH DEFAULT	Define que a coluna possui um valor *default*.
constante	O valor *default* é uma constante.
USER	O valor *default* é o CURRENT USER.
CURRENT SQLID	O valor *default* é o CURRENT SQLID.
NULL	O valor *default* é NULL.
GENERATED	Geração de Valores automática.
ALWAYS	Sempre o valor será gerado.
BY DEFAULT	O valor será gerado apenas se não for informado.
START WITH	Valor inicial.
MAXVALUE	Valor Máximo.
MINVALUE	Valor Mínimo.
FIELDPROC	Procedimento de validação para coluna.
CONSTRAINT nome_restrição	Nome da restrição.
PRIMARY KEY	Identifica restrição de chave primária.
UNIQUE	Identifica restrição de unicidade.
FOREIGN KEY	Identifica restrição de integridade referencial.
REFERENCES	Identifica a tabela referenciada pela FK.
ON DELETE	Define a regra de deleção para o relacionamento.
RESTRICT	Não é possível deletar se possuir filhos.
NO ACTION	Não é possível deletar se possuir filhos.
CASCADE	A deleção do pai será propagada para os filhos.
SET NULL	A deleção do pai será implicará na atualização dos filhos para NULL.
CHECK	Identifica uma restrição de integridade de domínio.
condição_check	Condição de *check* para o domínio.
LIKE	Define a tabela baseada na definição de uma outra tabela ou *view*.
INCLUDING IDENTITY COLUMN ATTRIBUTES	Copia também a definição das colunas *identity*.

Continua na próxima página

Parâmetro	**Significado**
VALIDPROC	Procedimento de validação para tabela.
nome_programa	Programa para validação.
EDITPROC	Define um procedimento de edição para tabela.
AUDIT	Altera o atributo de auditoria da tabela.
NONE	Não é realizada nenhuma auditoria.
Parâmetro	Significado
CHANGES	Operações de INSERT, UPDATE e DELETE são auditadas.
ALL	Maior nível de auditoria.
OBID inteiro	Identifica o OBID da tabela.
DATA CAPTURE	Trata do *log* de informações adicionais .
NONE	Não é feito nenhum log adicional.
CHANGES	Operações de INSERT, UPDATE e DELETE são registradas no LOG com informações adicionais.
WITH RESTRICT ON DROP	Não permite que o comando DROP seja executado enquanto este indicador existir para a tabela. Para retirar este indicador é necessário executar o comando ALTER TABLE.
CCSID	Identifica o conjunto de caracteres.

A.2 Estruturas Básicas da DML no DB2

A.2.1 SELECT

O comando SELECT pode ser escrito de três diferentes formas:

1.*subselect* - é um subconjunto do *fullselect*.

2.*fullselect* - é um subconjunto do comando SELECT.

3.Comando SELECT - implementa toda potencialidade de pesquisa.

SUBSELECT - subconjunto do *fullselect*

```
SUBSELECT ::=
    cláusula_select
    cláusula_from
    [cláusula_where]
    [cláusula_group_by]
```

```
    [cláusula_having]
cláusula_select::=
    SELECT[ALL|DISTINCT]{*|
        {expressão[AS nome_coluna],...|
        {nome_tabela|nome_view|nome_correlação}.*},...}
cláusula_from::= FROM espec_tabela,...
espec_tabela::=
        {nome_tabela|nome_view|referência_localizador_tabela
        |[TABLE] (fullselect)
        |TABLE (nome_função
         (expressão,...TABLE nome_tabela_transição,...))
        }[[AS] nome_correlação][(nome_coluna,...)]
        |junção_tabela}
junção_tabela::=
        {espec_tabela
        [INNER|{LEFT|RIGHT|FULL} [OUTER]]
        espec_tabela
        ON condição_join
        |(junção_tabela)}
condição_join (INNER, LEFT, RIGHT)::= condição_pesquisa
condição_join (FULL):=
        {{expressão_full_join = expressão_full_join} [AND]}...
expressão_full_join::=
        {{nome_coluna|função_cast}
        |{COALASCE|VALUE}(
        {nome_coluna|função_cast},
        {nome_coluna|função_cast},...)}
cláusula_where::= condição de pesquisa
cláusula_group_by::= GROUP BY nome_coluna,...
cláusula_having::= HAVING condição_pesquisa
```

De acordo com a sintaxe mostrada anteriormente, tem-se o significado de cada parâmetro utilizado, conforme ilustrado na Tabela A.5.

Tabela A.5: Comando *Select*

Parâmetro	**Significado**
ALL	Retorna todas as linhas.
DISTINCT	Retira linhas duplicada do conjunto resultado.
*	Retorna todas as colunas.
JOINS	Utilizado para retornar colunas de mais de uma tabela.
INNER	Conhecido como junção natural, retorna as linhas que possuem correspondências nas duas tabelas que participam do JOIN.
LEFT OUTER	Retorna todas as linhas do INNER, mais as linhas da tabela da esquerda que não possui correspondência.
RIGHT OUTER	Retorna todas as linhas do INNER, mais as linhas da tabela da direita que não possui correspondência.
FULL OUTER	Retorna todas as linhas do LEFT e RIGHT. Note que neste tipo de JOIN só pode ser usado a igualdade na condição de junção e o operador AND na concatenação dos predicados do JOIN.
COALASCE	Função utilizada no FULL OUTER JOIN para retornar o primeiro parâmetro não nulo.
VALUE	Mesmo efeito da função COALESCE.
WHERE	Especificam os predicados que filtram as linhas.
GROUP BY	Especifica as colunas que servirão de base para sumariar as linhas.
HAVING	Especificam os predicados que filtram os grupos sumariados.

SELECT (Parte II - *fullselect*)

O *fullselect* é um componente dos comandos INSERT, CREATE, VIEW e SELECT. O *fullselect* também é componente de alguns predicados, que por sua vez são componentes do *subselect* (veja *select* - parte I). Basicamente, o *fullselect* permite combinar o resultado de duas tabelas gerando uma nova tabela.

```
((fullselect)|subselect)
{UNION|UNION ALL}
((fullselect)|subselect)
```

De acordo com a sintaxe mostrada anteriormente, tem-se o significado de cada parâmetro utilizado, conforme ilustrado na Tabela A.6.

Tabela A.6: Comando *Select* (I)

Parâmetro	**Significado**
UNION	Gera uma tabela resultado combinando duas outras tabelas, retirando as duplicidades.
UNION ALL	Gera uma tabela resultado combinando duas outras tabelas, preservando todas as linhas de cada tabela podendo gerar linhas repetidas.

SELECT (Parte III - *SELECT* completo) - O comando *select* agrega algumas cláusulas ao *fullselect*.

```
fullselect
    [cláusula_order_by]
    [cláusula_read_only]
    [cláusula_update]
    [cláusula_optimize]
    [cláusula_with]
    [cláusula_queryno]
    [cláusula_fetch_first]
cláusula_order_by::=
    ORDER BY [nome_coluna|inteiro|expressão][ASC|DESC]
cláusula_read_only::= FOR{FETCH|READ] ONLY
cláusula_update::= FOR UPDATE [OF nome_coluna,...]
cláusula_optimize::= OPTIMIZE FOR inteiro {ROWS|ROW}
claúsula_with::=
    WITH{CC|UR|RR|[KEEP UPDATE LOCKS]
    |RS[KEEP UPDATE LOCKS]}
cláusula_queryno::= QUERYNO inteiro
cláusula_fetch_first::=
    FETCH FIRST{1|inteiro}{ROWS|ROW} ONLY
```

De acordo com a sintaxe mostrada anteriormente, tem-se o significado de cada parâmetro utilizado, conforme ilustrado na Tabela A.7.

Tabela A.7: Comando *Select* (II)

Parâmetro	**Significado**
ORDER BY	Ordena o conjunto resultado.
ASC	Ordena de forma ascendente.
DESC	Ordena de forma descendente.
FOR	Tipo de acesso.
FETCH ONLY	Informa que será feito apenas a leitura.
UPDATE	Informa que poderá ser feito um UPDATE.
OPTIMIZE	Solicita uma otimização da *query* onde será priorizada uma certa quantidade de linhas.
WITH	Nível de isolamento.
CS	*Cursor Stability* - Libera e adquire os *locks* a medida que o cursor vai sendo lido.
UR	*Uncommitted Read* - Não faz e não considera os *locks* existentes.
RR	*Repeatable Read* - Garante que o que foi lido não será alterado, mantendo os *locks* nas páginas/linhas.
RS	*Read Stability* - Garante que o que foi lido não será alterado mas permite que novas linhas sejam consideradas em novas leituras com o mesmo critério de filtro.
KEEP UPDATE LOCKS	Será solicitado diretamente *lock* X (exclusivo) ao invés do U(*update*) ou S(compartilhado).
QUERYNO	Define um número para ser utilizado pelo comando EXPLAIN.
FETCH FIRST	Limita o número de linhas serão lidas.

SELECT INTO - O comando SELECT INTO é utilizado para retornar no máximo uma linha. É mais eficiente do que declarar, abrir, ler e fechar um cursor.

```
cláusula_select
    INTO var_host,...
    cláusula_from
    [cláusula_where]
    [cláusula_group_by]
    [cláusula_having]
    [WITH{RR|RS|CS|UR}]
    [cláusula_queryno]
```

```
    [cláusula_fetch_first]
```

`cláusula_select, cláusula_from, cláusula_where, cláusula_group_by e cláusula_having ::= veja comando SELECT (PARTE I - subselect) cláusula_queryno, cláusula_fetch_first::= veja SELECT (PARTE III - SELECT completo)`

De acordo com a sintaxe mostrada anteriormente, tem-se o significado de cada parâmetro utilizado, conforme ilustrado na Tabela A.8.

Tabela A.8: Comando *Select* (III)

Parâmetro	**Significado**
WITH	Nível de isolamento.
CS	*Cursor Stability* - Libera e adquire os *locks* a medida que o cursor vai sendo lido.
UR	*Uncommitted Read* - Não faz e não considera os *locks* existentes.
RR	*Repeatable Read* - Garante que o que foi lido não será alterado, mantendo os *locks* nas páginas/linhas.
RS	*Read Stability* - Garante que o que foi lido não será alterado mas permite que novas linhas sejam consideradas em novas leituras com o mesmo critério de filtro.

A.2.2 INSERT

O comando INSERT permite a inclusão de novas linhas na tabela. Pode ser incluída uma linha de cada vez, ou várias linhas resultantes de uma consulta. As colunas da lista de inserção podem estar em qualquer ordem.

```
INSERT INTO {nome_tabela|nome_view}
     [(nome_coluna,...)]
     [OVERRINDING USER VALUE]
     {fullselect[WITH{RR|RS|CS}][QUERYNO inteiro]
     |VALUES {
        {expressão|DEFAULT|NULL}
        |(expressão,...DEFAULT,...NULL,...)
     }
```

```
fullselect:
    Ver comando SELECT.
```

De acordo com a sintaxe mostrada anteriormente, tem-se o significado de cada parâmetro utilizado, conforme ilustrado na Tabela A.9.

Tabela A.9: Comando *Insert*

Parâmetro	**Significado**
nome_tabela	Tabela onde as linhas serão inseridas.
nome_view	Visão onde as linhas serão inseridas
OVERRINDING USER VALUE	Define que o valor especificado para uma coluna que foi definida como GENERATED ALWAYS é ignorado deixando a responsabilidade de gerar valor para o sistema.
VALUES	Especifica os valores da nova linha.
DEFAULT	Será utilizado o valor *default* da coluna.
expressão	Será utilizado o resultado da expressão que deve ser do tipo compatível com o da coluna.
NULL	Será utilizado o valor NULL.
WITH	Define o nível de isolamento.
RR	Isolamento *Repeat Read*.
RS	Isolamento *Read Stability*.
CS	Isolamento *Cursor Stability*.
QUERYNO	Define um número para ser utilizado pelo comando EXPLAIN.

A.2.3 UPDATE

O comando UPDATE muda os valores das colunas especificadas em todas as linhas que satisfazem a condição. Somente as colunas a ser modificadas devem aparecer na lista de colunas da declaração.

```
 UPDATE {nome_tabela|nome_view} [nome_corr]
 SET cláusula_atribuição
 {[WHERE condição_pesquisa]}
  [WITH{RR|RS|CS}]
  [QUERYNO inteiro]
|[WHERE CURRENT OF nome_cursor]}
cláusula_atribuição::=
```

```
{nome_coluna = {expressão|NULL| (fullselect_escalar)
|(nome_coluna,...) = {({expressão|NULL},...)
                    |(fullselect_linha)}
}
fullselect_escalar::=fullselect que retorna apenas 1 valor
fullselect_linha::=fullselect que retorna apenas 1 linha
```

De acordo com a sintaxe mostrada anteriormente, tem-se o significado de cada parâmetro utilizado, conforme ilustrado na Tabela A.10.

Tabela A.10: Comando *Update*

Parâmetro	**Significado**
nome_tabela	Tabela onde as linhas serão atualizadas.
nome_view	Visão onde as linhas serão atualizadas.
nome_corr	Nome de correlação.
condição_pesquisa	Predicados que qualificam as linhas que serão deletadas.
WITH	Define o nível de isolamento.
RR	Isolamento *Repeat Read*.
RS	Isolamento *Read Stability*.
CS	Isolamento *Cursor Stability*.
QUERYNO	Define um número para ser utilizado pelo comando EXPLAIN.
WHERE CURRENT OF	Atualiza a linha na qual o cursor está posicionado.

A.2.4 DELETE

O comando DELETE exclui as linhas que satisfazem a cláusula WHERE na tabela especificada. Por padrão o DELETE exclui as linhas da tabela especificada e de todas as suas descendentes. Para atualizar somente a tabela especificada deve ser utilizada a cláusula ONLY.

```
DELETE FROM {nome_tabela|nome_view} [nome_corr]
    {[WHERE condição_pesquisa]}
     [WITH{RR|RS|CS}]
     [QUERYNO inteiro]
    |[WHERE CURRENT OF nome_cursor]
    }
```

De acordo com a sintaxe mostrada anteriormente, tem-se o significado de cada parâmetro utilizado, conforme ilustrado na Tabela A.11.

Tabela A.11: Comando *Delete*

Parâmetro	**Significado**
nome_tabela	Tabela onde as linhas serão deletadas.
nome_view	Visão onde as linhas serão deletadas.
nome_corr	Nome de correlação.
condição_pesquisa	Predicados que qualificam as linhas que serão deletadas.
WITH	Define o nível de isolamento.
RR	Isolamento *Repeat Read*.
RS	Isolamento *Read Stability*.
CS	Isolamento *Cursor Stability*.
QUERYNO	Define um número para ser utilizado pelo comando EXPLAIN.
WHERE CURRENT OF	Deleta a linha na qual o cursor está posicionado.

Apêndice B

Código-Fonte das Classes

B.1 Descrição do Funcionamento das Classes

B.1.1 *SQL*

Classe que recebe pedidos de execução de consultas. Utiliza a Classe *SQLCompiler* para descobrir o tipo de comando e decidir a maneira que as consultas devem ser executadas. Caso os comandos iniciem com PERIOD ou INITIALIZE, faz uso das Classes *Period e Initialize* para execução dos mesmos, do contrário executa os comandos de acordo com o exigido pelas demais primitivas. Possui métodos para avisar aos interessados *(SQL-Listener's)* sobre o resultado da execução de um comando. E parte da Classe SQL é implementada conforme Código a seguir:

```
public class SQL {

    public final int ERROR = 0;
    public final int INITIALIZE = 1;
    public final int TERMINATE = 2;
    public final int PERIOD_INITIALIZE = 3;
    public final int COMPUTATION = 4;
    public final int COMPUTATION_TIME = 5;

    public final int INITIALIZE_TERMINATE = 6;
    public final int INITIALIZE_COMPUTATION_TERMINATE = 7;
    public final int PERIOD_INITIALIZE_TERMINATE = 8;
    public final int COMPUTATION_TERMINATE = 9;
    public final int VALIDATE = 10;
```

```
public final void executeQuery(String sql, Integer id) {
   if(!compiler.hasPrimitive(sql)){
       Object result = executeQuery(sql);
       fireQueryCompleted(id,sql,result);
   }else{

       ArrayList commands = compiler.compile(sql);
       int method = Integer.parseInt(commands.get(0).toString());

switch (method) {

   case ERROR :
       fireQueryCompleted(id,sql,commands.get(1).toString());
       break;

   case INITIALIZE :
       try {
           executeInitialize("QUERY",sql, commands, id);
       } catch (InvalidDateException e) {
           log.warn("SQL.executeQuery()- " + e.getMessage());
           fireQueryCompleted(id,sql,e.getMessage());
       }
       break;

   case TERMINATE :
       try {
           executeTerminate("QUERY",sql,commands,id);
       } catch (InvalidDateException e) {
           log.warn("SQL.executeQuery() - " +e.getMessage());
           fireQueryCompleted(id,sql,e.getMessage());
       }
       break;

   case PERIOD_INITIALIZE :
       try {
           executePeriodInitialize("QUERY",sql,commands,id);
```

```
        } catch (InvalidDateException e) {
            log.warn("SQL.executeQuery() - " + e.getMessage());
            fireQueryCompleted(id,sql,e.getMessage());
        }
        break;

    case COMPUTATION :
        executeComputation("QUERY",sql, commands,id);
        break;

    case INITIALIZE_TERMINATE :
        try {
            executeInitializeTerminate("QUERY",sql,commands,id);
        } catch (InvalidDateException e) {
            log.warn("SQL.executeQuery() - " +e.getMessage());
            fireQueryCompleted(id,sql,e.getMessage());
        }
        break;

    case INITIALIZE_COMPUTATION_TERMINATE :
        try {
            executeInitializeComputationTerminate("QUERY",sql,commands,id);
        } catch (InvalidDateException e) {
            log.warn("SQL.executeQuery() - " + e.getMessage());
            fireQueryCompleted(id,sql,e.getMessage());
        }
        break;

    case PERIOD_INITIALIZE_TERMINATE :
        try {
            executePeriodInitializeTerminate("QUERY",sql,commands,id);
        } catch (InvalidDateException e) {
            log.warn("SQL.executeQuery() - " + e.getMessage());
            fireQueryCompleted(id,sql,e.getMessage());
        }
        break;
```

```
    case COMPUTATION_TERMINATE :
        try {
            executeComputationTerminate("QUERY",sql,commands,id);
        } catch (InvalidDateException e) {
            log.warn("SQL.executeQuery() - " + e.getMessage());
            fireQueryCompleted(id,sql,e.getMessage());
        }
        break;

    case VALIDATE :
            executeValidate(sql,commands.get(1).toString(),id);
        break;

    default :
        break;
        }
    }
                              .
                              .
                              .
}
```

B.1.2 *SQLCompiler*

Esta Classe funciona como uma espécie de interpretador. Sua função é receber um comando e separar a parte da SQL-99 das primitivas. Para cada combinação possível de primitivas, ela faz um tratamento específico. É importante ressaltar que esta Classe não apresenta programação defensiva, ou seja, ela funciona corretamente desde que a informação passada esteja correta. E parte da Classe *SQLCompiler* é implementada conforme Código a seguir:

```
public class SQLCompiler {

    private Logger log;
    private SQL sql;

    private final String INITIALIZE = "INITIALIZE";
```

```
    private final String TERMINATE = "TERMINATE";
    private final String PERIOD = "PERIOD";
    private final String COMPUTATION = "COMPUTATION";
    private final String RETURN_TC = "RETURN_TC";
    private final String VALIDATE = "VALIDATE";

    public SQLCompiler(SQL sql) {
        this.sql = sql;
        log = Logger.getLogger(SQLCompiler.class);
        BasicConfigurator.configure();
        log.setLevel(Level.WARN);
    }

    public boolean hasPrimitive(String command) {
        BufferedReader reader = new BufferedReader(new
        StringReader(command));
        String line = "";
        boolean isPrimitive = false;
        try {
            line = reader.readLine();
            while (!isPrimitive && line != null) {
                StringTokenizer tokenizer = new StringTokenizer(line);
                if(tokenizer.countTokens() > 0)
                    isPrimitive = isPrimitive(tokenizer.nextToken());
                line = reader.readLine();
            }
        } catch (IOException e) {
            log.warn(e.getMessage());
        }
        return isPrimitive;
    }

public ArrayList compile(String sql2) {

     ArrayList lines = readLines(sql2);//quebra o comando em linhas

    if(lines.size() < 2){
```

```
            lines.add(new Integer(sql.ERROR));
            lines.add("Verifique se o comando SQL-99 está
            na mesma linha da primitiva");
            return lines;
        }

        String firstLine = ((String) lines.get(0)).toUpperCase();
        String lastLine = ((String) lines.get(lines.size() - 1)).toUpperCase();

        if (firstLine.startsWith(INITIALIZE))
            return initialize(lines, firstLine);
        if (firstLine.startsWith(PERIOD))
            return period(lines, firstLine);
        if (lastLine.startsWith(VALIDATE))
            return validate(lines, lastLine);
        if (lastLine.startsWith(TERMINATE))
            return terminate(lines, lastLine);
        if (lastLine.startsWith(COMPUTATION))
            return computation(lines, lastLine);
        if (lastLine.startsWith(RETURN_TC))
            return computationTime(lines);
        log.debug("Não deveria chegar aqui");
        return null;
    }
                                .
                                .
                                .
    }
```

B.1.3 *SQLListener*

A Classe *SQLListener*, é a interface para os interessados nos eventos lançados pela Classe *SQL*, os chamados *SQLEvent's*. É composto de cinco métodos: um para recuperação da identificação do *listener* e quatro para o aviso de eventos propriamente ditos. E a Classe *SQLListener* é implementada conforme Código a seguir:

```
package rtdbms.db;
```

```
import java.util.EventListener;

public interface SQLListener extends EventListener {

    public void queryCompleted(SQLEvent source);
    public void queryWithPrimitiveCompleted(SQLEvent source);
    public void updatedCompleted(SQLEvent source);
    public void updatedWithPrimitiveCompleted(SQLEvent source);
    public Integer getId();

}
```

B.1.4 *SQLEvent*

Esta Classe é utilizada na implantação do padrão *Observer*, representando o evento no mesmo. É criada(instanciada) pela classe SQL e passada como parâmetro nos métodos da classe SQLListener, que representa os interessados nesse tipo de evento. A Classe *SQLEvent* é implementada conforme Código a seguir:

```
package rtdbms.db;

import java.util.EventObject;

public class SQLEvent extends EventObject{

    private String query = "";
    private Object result;

    public SQLEvent(SQL arg0, String query, Object result) {
        super(arg0);
        this.query = query;
        this.result = result;
    }

    public String getQuery() {
        return query;
```

```
    }

    public Object getResult() {
        return result;
    }
}
```

B.1.5 *Initialize*

É um *TimerTask* que realiza alguma consulta em uma hora pré-determinada. O agendamento desta tarefa é feito utilizando-se um objeto *Timer*, que será responsável por acionar o método *run* desta classe no horário especificado. É sempre utilizada com comandos que iniciam com a primitiva INITIALIZE. E parte da Classe *Initialize* é implementada conforme Código a seguir:

```
public class Initialize extends TimerTask {

    private int command_type = 0;

    private String sql, query, type;
    private Integer id, minutes_computation, seconds_computation;
    private Date end;
    private Logger log;

  public Initialize(String sql, String query, String
        type, Integer id ) {
        this(sql,query,type, id, null, null, null);
    }

  public Initialize(String sql, String query, String type,
        Integer id, Date end) {
        this(sql,query,type, id, null, null, end);
    }
  public Initialize(String sql, String query, String type,
        Integer id, Integer minutes_computation, Integer
        seconds_computation, Date end) {
        super();
```

```
        this.sql = sql;
        this.query = query;
        this.type = type;
        this.id = id;
        this.minutes_computation = minutes_computation;
        this.seconds_computation = seconds_computation;
        this.end = end;

        log = Logger.getLogger(Initialize.class);
        BasicConfigurator.configure();
        log.setLevel(Level.WARN);

        if(end == null)
            command_type = 1;
        else if(minutes_computation == null)
            command_type = 2;
            else command_type = 3;
    }

public void run(){

            if(sql == null)
                return;
            if(type.equalsIgnoreCase("QUERY")){
                if(command_type == 1)
                    executeQueryCommand1();
                else
                    if(command_type == 2)
                        executeQueryCommand2();
                    else executeQueryCommand3();
                return;
            }
            if(type.equalsIgnoreCase("UPDATE")){
                if(command_type == 1)
                    executeUpdateCommand1();
                else
                    if(command_type == 2)
```

```
                    executeUpdateCommand2();
                else executeUpdateCommand3();
            return;
        }
                        .
                        .
                        .
}
```

B.1.6 *Period*

Assim como *Initialize*, é um *TimerTask* que realiza alguma consulta em uma hora pré-determinada. O agendamento desta tarefa é feito utilizando-se um objeto *Timer*, que será responsável por acionar o método *run* desta classe no horário especificado. É sempre utilizada com comandos que iniciam com a primitiva PERIOD. E parte da Classe *Period* é implementada conforme Código a seguir:

```
public class Period extends TimerTask{

    private Logger log;
    private String sql, query, type;
    private Integer id;
    private Date end;
    private Timer timer;

    public Period(String sql, String query, String type, Integer id,
    Timer timer) {
        this(sql, query,type,id,null,timer);
    }

    public Period(String sql, String query, String type, Integer id,
    Date end, Timer timer) {
        super();
        this.sql = sql;
        this.query = query;
        this.type = type;
```

```
        this.id = id;
        this.end = end;
        this.timer = timer;

        log = Logger.getLogger(Initialize.class);
        BasicConfigurator.configure();
        log.setLevel(Level.WARN);
    }

    public void run(){
        if(sql == null)
            return;
        if(type.equalsIgnoreCase("QUERY")){
            try {
                executeQuery();
            } catch (Throwable e) {
                log.warn(e.getMessage());
            }
            return;
        }
        if(type.equalsIgnoreCase("UPDATE")){
            try {
                executeUpdate();
            } catch (Throwable e) {
                log.warn(e.getMessage());
            }
            return;
        }
    }
```