UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

PROTÓTIPO DE UM BANCO DE DADOS RELACIONAL CLIENTE / SERVIDOR

JOHN CRISTIAN DOERNER

BLUMENAU
2004

2004/1-16

JOHN CRISTIAN DOERNER

PROTÓTIPO DE UM BANCO DE DADOS RELACIONAL

CLIENTE / SERVIDOR

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

Prof. Alexander Roberto Valdameri - Orientador

BLUMENAU
2004

2004/1-16

PROTÓTIPO DE UM BANCO DE DADOS RELACIONAL

CLIENTE / SERVIDOR

Por

JOHN CRISTIAN DOERNER

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

Presidente:	Prof. Alexander Roberto Valdameri – Orientador, FURB
Membro:	Prof. Marcelo José Ferrari, FURB
Membro:	Prof. Maurício Capobiaco Lopes, FURB

Blumenau, junho de 2004

Dedico este trabalho a todos os amigos, especialmente aqueles que me ajudaram diretamente na realização deste.

Os bons livros fazem “sacar” para fora o que a pessoa tem de melhor dentro dela.

Lina Sotis Francesco Moratti

AGRADECIMENTOS

Primeiramente a Deus, por ter me dado a vida, saúde e sabedoria necessária para enfrentar os obstáculos.

Aos meus pais José e Leopoldina, pelo amor, carinho, educação e todo apoio necessário para minha formação.

Aos meus amigos, pelos empurrões e cobranças.

Ao meu orientador, Alexander Roberto Valdameri pelas sugestões e esclarecimentos que foram de grande importância para a elaboração e conclusão deste trabalho, fica a minha eterna gratidão.

RESUMO

A implementação de um repositório para armazenamento de dados requerer a utilização de uma série de conceitos ligados à computação, tais como: compiladores, gerência de memória (volátil e não-volátil), aspectos de concorrência, recuperação e distribuição. Este trabalho tem como objetivo apresentar o processo de desenvolvimento de um sistema de banco de dados relacional para ambiente cliente/servidor. Na especificação deste trabalho foi definida uma gramática para comandos SQL e os controles necessários para o sistema servidor, permitindo identificar, executar e validar todas requisições feitas pela interface cliente. Para a implementação do trabalho utilizou-se o ambiente de programação Delphi.

Palavras chaves: Banco de dados relacional; SGBD; Cliente / Servidor.

ABSTRACT

The implementation of repository data storage require use a series of computation concepts, such as: compilers, management memory (volatile and not-volatile), aspects of competition, backup and distribution. This work has as objective to present development process a database system relationary for client/server environment. In this work specification a grammar for commands SQL was defined and the necessary controls stop the serving system, allowing to identify, execute and validate all solicitations made for the customer interface. For the work implementation was used environment of Delphi programming.

Key-Words: DataBase Relationary; DBMS; client/server.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

DBA – Administrador do Banco de Dados

DDL – Linguagem de Definição de Dados

DML – Linguagem de Modelagem de Dados

SGBD – Sistema Gerenciador de Banco de Dados

UML – Linguagem de Modelagem Unificada

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Com o crescimento da informação no mundo, o uso de um sistema de banco de dados é cada vez mais utilizado em diversos sistemas computacionais. Caracteriza-se por ser um sistema formado de vários conjuntos de dados, associados a um conjunto de programas para acesso a esses dados, que permitem uma interface eficiente para a recuperação e armazenamento das informações dentro do banco de dados.

Conforme Silberschatz, Korth e Sudarschan (1999), os sistemas de banco de dados são projetados para guardar grandes volumes de informações. O gerenciamento dos dados implica na definição das estruturas de armazenamento e na definição dos mecanismos para a manipulação dos dados armazenados. Um sistema de banco de dados deve garantir a segurança dos dados armazenados contra eventuais problemas que possam ocorrer com o sistema, além de impedir o acesso não autorizado. Se os dados são compartilhados por diversos usuários, o sistema deve gerenciar os acessos a fim de evitar problemas de integridade dos dados.

Segundo Molina, Ullman e Widom (2001, p.1), “O poder dos bancos de dados vem de um corpo de conhecimento e tecnologia que se desenvolveu ao longo de várias décadas e é encarnado em um tipo de software especializado chamado sistema de gerenciamento de bancos de dados” (SGBD). Os SGBDs são sistemas de grande complexidade pois necessitam de controle eficiente das informações armazenadas para permitir realizar consultas de acordo com a necessidade.

Neste trabalho foi criado um protótipo de banco de dados relacional, que possui as principais rotinas de um banco de dados como a gerência de armazenamento, gerência de seções, permitindo a manutenção dos dados através de comandos SQL.

1.2 OBJETIVOS

O objetivo deste trabalho é criar um protótipo de um servidor de banco de dados, que compartilhe os dados armazenados.

Os objetivos específicos do trabalho são:

a) criar gerenciador de sessão que controle o acesso e permissão do usuário;
b) criar mecanismo baseado no controle das *threads* do banco de dados;
c) criar mecanismo para processamento de comandos em SQL;
d) criar mecanismo que mantenha a integridade da informação;
e) desenvolver mecanismo para identificar os requisitos pela rede.

1.3 JUSTIFICATIVA

O trabalho justifica-se pelo grande número de "conceitos" relacionados à implementação de um banco de dados, dentre ela pode-se citar: técnicas de compilação, gerência de memória e protocolo de comunicação.

1.4 ORGANIZAÇÃO DO TRABALHO

Este trabalho está divido em forma de capítulos descritos a seguir.

O primeiro capítulo expõe, na introdução, uma justificativa do que originou este trabalho, como também uma síntese do que será tratado no desenvolvimento do trabalho e os objetivos a serem alcançados.

O segundo capítulo apresenta o conceito da origem do banco de dados, com suas principais características como mecanismo de controles, linguagem de acesso e o ambiente cliente / servidor.

O terceiro capítulo apresenta as principais etapas para a implementação de um banco de dados relacional cliente / servidor.

O quarto capítulo apresenta a especificação feita para o desenvolvimento e apresenta o sistema cliente / servidor com suas funcionalidades.

O quinto capítulo expõe as considerações finais após o desenvolvimento do trabalho e algumas sugestões para sua continuação.

2 BANCO DE DADOS RELACIONAL

2.1 CONCEITO

O modelo de dados relacional representa os dados contidos em um Banco de Dados através de relações. Estas relações contém informações sobre as entidades representadas e seus relacionamentos. O Modelo Relacional é claramente baseado no conceito de matrizes, onde as chamadas linhas seriam os registros e as colunas seriam os campos. Os nomes das tabelas e dos campos são de fundamental importância para a compreensão entre o que se pretende armazenar.

O SGBD surgiu como necessidade para simplificar o desenvolvimento de aplicações caracterizadas por uso intensivo de dados, uma vez que os bancos de dados provêm serviços que diminuem o tempo de desenvolvimento das aplicações digitais, permitindo realizar entrada de dados de forma simples, possibilitando manipular dados de acordo com a aplicação, independente de plataforma de hardware ou ambiente operacional.

Com o uso dos bancos de dados, os sistemas de informação foram substituindo a forma de armazenamento dos dados, pois as aplicações utilizavam a gerência de arquivos do sistema operacional, limitando-se às características do mesmo. Os SGBD permitem independência de dados entre programas, o que permite fazer modificações como inclusão de um novo campo sem afetar os programas. Existe uma maior abstração de dados pois permite uma representação conceitual através de um modelo de dados que só usa conceitos lógicos.

O sistema de banco de dados vem evoluindo progressivamente durante os anos, onde foram implementadas novas funcionalidades oferecidas por sistemas de arquivos. Funcionalidades como: registro com tamanho variável, o uso de memória virtual e persistência, a utilização de índices baseado em *hash* e árvove-B e o controle de bloqueio em nível de registro para concorrência.

O uso de sistemas de bancos de dados é cada vez mais essencial pois permite um maior potencial de padronização e flexibilidade no desenvolvimento de aplicativos, permitindo reduzir o tempo no desenvolvimento da aplicação que necessite de armazenamento de dados.

2.2 CARACTERÍSTICAS E FUNCIONALIDADES

Segundo Date (2000, p.37) os SGBD possuem sete características operacionais elementares, descritas a seguir:

a) **definição de dados:** o SGBD deve ser capaz de aceitar definições de dados em forma fonte e convertê-los para a forma objeto apropriada. Deve incluir componentes de processador de DDL para cada uma das diversas linguagens de definição de dados (DDL – *Data Description Language*);

b) **manipulação de dados:** o SGBD deve ser capaz de lidar com solicitações do usuário para buscar, atualizar ou excluir dados existentes no banco de dados, ou para acrescentar novos dados ao banco de dados. Deve incluir um componente processador de DML para lidar com a linguagem de manipulação de dados (DML – *Data Manipulation Language*);

c) **otimização de execução:** as requisições de DML, planejadas ou não-planejadas, devem ser processadas pelo componente otimizador, cujo propósito é determinar um modo eficiente de implementar a requisição. As requisições otimizadas são então executadas sob o controle do gerenciador em tempo de execução (*run time*);

d) **segurança e integridade de dados:** o SGBD deve monitornar requisições de usuários e rejeitar toda tentativa de violar as restrições de segurança e integridade definidas pelo DBA (*database administrador*). Essas tarefas podem ser executadas em tempo de compilação ou em tempo de execução, ou ainda em alguma mistura dos dois;

e) **recuperação e concorrência de dados:** o SGBD, ou , mais provavelmente, algum outro componente de software relacionado, em geral chamado gerenciador de transações deve impor certos controles de recuperação e concorrência;

f) **dicionário de dados:** o SGBD deve fornecer uma função de dicionário de dados. O dicionário de dados pode ser considerado um banco de dados em si. O dicionário contém dados sobre os dados chamados de metadados, ou seja, definições de outros objetos do sistema, em vez de dados brutos somente;

g) **desempenho:** o SGBD deve realizar todas as funções identificadas anteriormente de forma tão eficiente quanto possível;

2.3 LINGUAGEM DE ACESSO

Conforme Harrington (2002, p.66), a origem da linguagem SQL e do modelo de banco de dados relacional reporta ao Dr. E. F. Codd, pesquisador da IBM, primeiro a publicar um artigo sobre a idéia de banco de dados relacional em Junho de 1970. A linguagem SQL for originalmente desenvolvida na IBM em um projeto de sistema de gerenciamento de banco de dados relacional, como uma linguagem de consulta a banco de dados denominada SEQUEL, sigla para (*Structured English Query Language*) (Linguagem de Consulta Estruturada em Inglês). O nome foi posteriormente mudado para SQL (*Structured Query Language*), por razões legais.

No final da década de 70, duas outras companhias iniciaram desenvolvimento de produtos similares, que vieram a ser Oracle e Ingres. A Oracle Corporation introduziu a primeira implementação de SQL comercialmente disponível, e é hoje uma das líderes no mercado de servidores de bancos de dados. A IBM também implementou SQL em seus sistemas de banco de dados DB2 e SQL/DS. Entre os anos 80 e 90, os produtos com SQL se multiplicaram e hoje SQL é largamente implementada e aceita como o padrão de fato da indústria para linguagem de acesso a bancos de dados, desde sistemas *desktop* como o Microsoft Access para Windows até sistemas de gerenciamento e servidores de bancos de dados de médio e grande porte em Unix, NT e *mainframes*.

2.3.1 DDL

A **linguagem de definição dos dados (***Data Description Language***)** define as aplicações, arquivos e campos que irão compor o banco de dados (comandos de criação e atualização da estrutura dos campos das tabelas) é composta pelos comandos destinados à criação de banco de dados, tabelas e relações. Como exemplo de comandos da classe DDL existe os comandos *Create*, *Alter* e *Drop*.

O resultado da compilação de instruções DDL é a definição de um conjunto de tabelas que são armazenadas no Dicionário de Dados. O Dicionário de Dados (ou metadados) é um repositório especial que contém “dados acerca dos dados armazenados” e é freqüentemente acessado pelo sistema de banco de dados.

A estrutura de armazenamento e os métodos de acesso são também especificados por um conjunto de definições através de um tipo especial de DDL. A compilação dessas definições resulta em um conjunto de instruções que especificam os detalhes de implementação dos esquemas.

2.3.2 DML

A **linguagem de manipulação dos dados (***Data Manipulation Language***)** define os comandos de manipulação e operação dos dados, destinadas às operações de consulta, inclusão, exclusão e alteração de registros das tabelas de uma banco de dados. Como exemplo de comandos da classe DML existe os comandos *Select*, *Insert*, *Update* e *Delete*.

2.4 AMBIENTE CLIENTE / SERVIDOR

Um sistema de banco de dados, na sua maioria, pode ser considerado como sendo uma estrutura dividida em duas partes, consistindo em um servidor e um conjunto de clientes. O servidor é o próprio SGBD, o qual admite todas as funções e gerenciamento. Os clientes são as diversas aplicações executadas sobre o SGBD – tanto aplicações escritas por usuários quanto aplicações internas, ou seja, aplicações fornecidas pelos fabricantes do SGBD ou por produtores independentes.

A arquitetura cliente/servidor é atualmente a principal plataforma tecnológica da indústria da tecnologia da informação. A sua popularização se deve aos vários fatores oriundos do processo de achatamento das estruturas organizacionais, fazendo com que muitos dos sistemas fossem descentralizados.

Segundo Renaud (1994, p.3) “cliente/servidor é um conceito lógico, mais precisamente um paradigma, ou modelo para interação entre processos de software em execução concorrente”. Isso significa dizer que a metodologia cliente/servidor foi criada com o objetivo de possibilitar que vários tipos de aplicações, executadas em máquinas distintas, se comuniquem entre si, de forma independente.

Baseado neste conceito, a arquitetura cliente/servidor estabeleceu um novo paradigma de processamento de dados, diversificando o processamento entre dois processos de software distintos (cliente e servidor). Ao mesmo tempo a arquitetura visa fornecer recursos que

coordenem estes processos de forma que, a perda de sincronização, não resulte em alterações ou perda de informações para o sistema.

Seu funcionamento baseia-se no seguinte esquema: o usuário do sistema, através do processo de software cliente, envia o pedido de requisição ao processo de software servidor, que por sua vez devolve ao cliente os resultados solicitados. Em geral, os processos de software do servidor rodam sobre o controle do Sistema Operacional que coordena todos os recursos do sistema computacional utilizado.

Atualmente, mesmo possibilitando a execução dos processos tanto o cliente quanto o servidor em uma única máquina, o que caracteriza realmente o fundamento da arquitetura cliente/servidor é a divisão do poder de processamento, onde sistema servidor é responsável por tratar e responder todos os pedidos enviados pelo sistema cliente. Os dois processos são separados em máquinas distintas e ao mesmo tempo, são interligadas através de uma rede de computadores local (LAN) ou remota (WAN), o que permite às estações de trabalho processarem os dados armazenados no servidor, liberando o mesmo para a execução de outras aplicações.

O sistema cliente é a parte responsável pela tarefa de requisição de pedidos ao servidor e também por toda a parte relativa à interação com o usuário final. Normalmente os sistemas cliente abstraem do usuário todas as funções de rede e do servidor, fazendo parecer que todos os processos estão rodando em um mesmo local.

Para prover esta interação, o sistema cliente abrange um conjunto de componentes básicos que auxiliam nas funcionalidades, tanto ao nível de aplicação como de sistema. Estes componentes de acordo com Melo (1997, p.28) são agrupados em:

a) hardware de estação: é formado pelos componentes básicos de um sistema de computação, tais como unidade central de processamento (CPU), memória, unidades de disco e dispositivos de entrada e saída de dados (periféricos);

b) sistema operacional: é o software que possui o conjunto de instruções necessárias para gerenciar os recursos de hardware e fornecer os meios necessários para que as aplicações utilizem estes recursos de forma adequada;

c) interface de conectividade: concentra o conjunto de instruções para permitir que os processos cliente interajam com o processo servidor através da rede de comunicação;

d) os programas de aplicação: consistem em um conjunto de programas desenvolvidos com a finalidade de realizar operações que atendam a uma necessidade específica do usuário ou organização;

e) interface gráfica de usuário *(GUI)*: é o principal componente de interação de interação com os usuários finais, pois é o que torna as aplicações serem utilizadas de forma mais simples e intuitiva.

O sistema servidor é a parte responsável de um sistema cliente/servidor que tem a função de receber dos clientes as requisições, processa-las e devolve-las ao mesmo os resultados. A grande vantagem desse sistema é que, por ser totalmente reativo, só é disparado quando recebe alguma requisição do cliente. Isso faz com que o servidor não procure interagir com outros servidores durante um pedido de requisição, o que torna o processo de ativação uma tarefa a ser desempenhada apenas pelo cliente que o solicitou.

Da mesma forma que o sistema cliente, o sistema servidor possui um conjunto de componentes básicos para prover as funcionalidades necessárias ao processamento de informações através da rede. Estes componentes são divididos em:

a) hardware de servidor: normalmente são compostos por sistemas de computação que variam de microcomputadores de alto desempenho até computadores de grande porte. Estes sistemas, para cumprir a função de servidor, devem possuir alta capacidade de armazenamento e grande quantidade de memória para fornecer melhor desempenho aos processos que estarão sempre rodando a espera de requisições;

b) o sistema operacional de rede: consiste em um recurso de software que além de gerenciar os componentes de hardware, fornecem recursos que possibilitam obter o controle total da rede de comunicações através de componentes de controle de acesso, compartilhamento de recursos, administração e gerência, além de outras funções de rede necessárias;

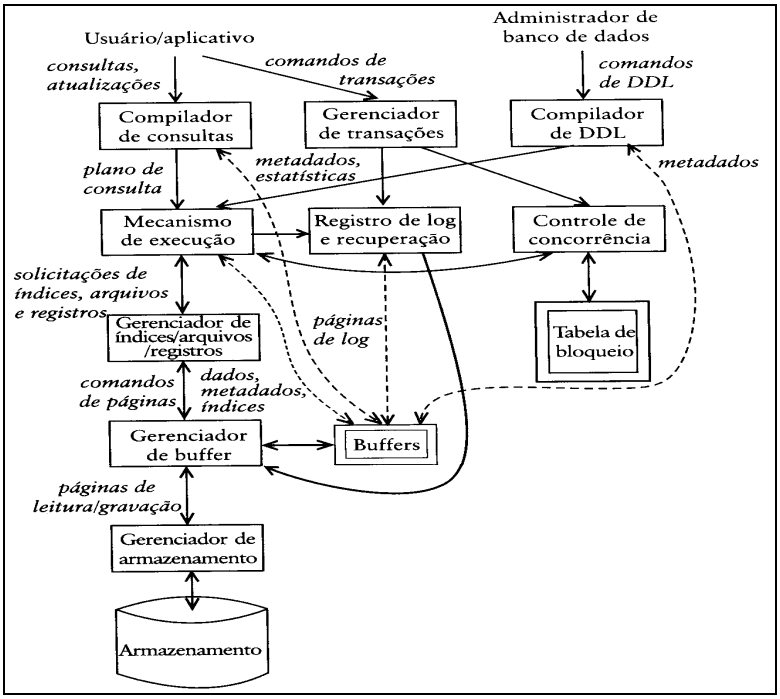
c) interface de conectividade: são caracterizados pelo uso de protocolos de comunicação e de interfaces para acesso a bancos de dados;

d) o SGBD: é o componente do sistema de banco de dados responsável por todo o gerenciamento e controle centralizado dos dados operacionais.

3 ETAPAS PARA IMPLEMENTAÇÃO DE UM BANCO DE DADOS

Os SBDG são caracterizados por sua habilidade de admitir o acesso eficiente à grandes quantidades de dados, os quais persistem ao longo do tempo. Eles também são caracterizados por seu suporte para linguagens de consulta poderosas e transações duráveis que podem ser executadas de forma concorrente de modo atômico e independente de outras transações.

Molina, Ullman e Widom (2001) apresentam os principais componentes de SGBD (figura 1).



Fonte: Molina; Ullman e Widom (2001, p.8)

Figura 1 – Componente de um sistema de gerenciamento de banco de dados

A seguir far-se-á uma breve explanação dos principais componentes, alguns deles implementados neste trabalho.

3.1 RECONHECIMENTO E INTERPRETAÇÃO

O processador de consultas é o componente de um SGBD que transforma os comandos submetidos pelo usuário em uma seqüência de operações sobre o banco de dados para em seguida executá-los. Tendo em vista que a linguagem SQL permite expressar consultas em um alto nível de abstração, o processador de consultas deve fornecer uma grande quantidade

de detalhes a respeito do modo como a consulta deve ser executada. Além disso, uma estratégia de execução ingênua para uma consulta pode levar a um algoritmo de execução da consulta que demorará muito mais tempo que o necessário.

O compilador de consultas é responsável em agilizar as consultas, selecionando um plano de consulta e executando o mesmo sobre os dados armazenados.

3.1.1 O compilador de consultas

A arquitetura do compilador de consultas prevê a seguinte seqüência de operações:

a) a consulta, escrita em uma linguagem baseada na SQL, é analisada, isto é, convertida em uma árvore de análise que representa a estrutura da consulta;

b) a árvore de análise é transformada em uma árvore de expressões de álgebra relacional (ou uma notação semelhante), denominados plano lógico de consulta;

c) o plano lógico de consulta deve ser convertido em um plano físico de consulta, que indica não apenas as operações executadas, mas a ordem em que elas são executadas, o algoritmo usado para executar cada etapa e ainda a maneira como os dados armazenados são obtidos e como os dados são repassados de uma operação para outra. O resultado dessa etapa é a árvore de análise para a consulta.

3.1.2 Uma álgebra para consultas

Segundo Molina, Ullman e Widom (2001) muitas consultas em SQL são expressas com alguns operadores que formam a “álgebra relacional” clássica. Porém, também existem recursos de SQL e de outras linguagens de consulta que não são expressas na álgebra relacional clássica, tais como agrupar, ordenar e selecionar valores distintos.

Além disso, a álgebra relacional foi projetada originalmente como se as relações fossem conjuntos. Na verdade, as relações em SQL são “sacolas” ou conjuntos múltiplos; isto é, o mesmo registro pode aparecer mais de uma vez em uma relação de SQL. Desse modo, deve-se introduzir a álgebra relacional como uma álgebra sobre sacolas. Os operadores da álgebra relacional são:

a) união, interseção e diferença: em conjuntos, esses são os operadores usuais segundo a ISO 99, e correspondem aos operadores de SQL UNION, INTERSECT e EXCEPT.

b) seleção: esse operador produz uma nova relação a partir de uma antiga, selecionando algumas linhas da relação antiga com base em alguma condição ou algum predicado. Ele corresponde em linhas gerais à cláusula WHERE de uma consulta de SQL.
c) projeção: esse operador produz uma nova relação a partir de uma antiga, escolhendo algumas colunas, como a cláusula SELECT de uma consulta de SQL.
d) produto: esse operador é o produto cartesiano (ou produto cruzado) da teoria de conjuntos, que constrói tuplas emparelhando os registros de duas relações de todas as maneiras possíveis. Ele corresponde em SQL à lista de relações em uma cláusula FROM, cujo produto forma a relação à qual são aplicadas a condição da cláusula WHERE e a projeção da cláusula SELECT.
e) eliminação de duplicatas: esse operador transforma uma sacola em um conjunto, como a palavra-chave DISTINCT em uma cláusula SELECT de SQL.
f) agrupamento. esse operador foi criado para imitar o efeito de uma cláusula GROUP BY de SQL, bem como dos operadores de agregação (soma, média e assim por diante) que podem aparecer em cláusulas SELECT de SQL.
g) classificação: esse operador representa o efeito da cláusula ORDER BY de SQL. Ele também é usado como parte de certos algoritmos baseados na classificação para outros operadores, como o de junção.

Conforme Molina, Ullman e Widom (2001) a utilização de álgebra relacional é que torna fácil explorar formas alternativas de uma consulta. As diferentes expressões algébricas para uma consulta são chamadas planos de consulta lógica. Freqüentemente, esses planos são representados sob a forma de árvores de expressões. Esses métodos diferem em sua estratégia básica; examinar, misturar, classificar e indexar. Os métodos também diferem em sua suposição sobre a quantidade de memória principal disponível. Alguns algoritmos pressupõem que existe memória principal suficiente disponível para conter pelo menos uma das relações envolvidas em uma operação. Outros pressupõem que os argumentos da operação são muito grandes para caberem na memória, e esses algoritmos têm custos e estruturas significativamente diferentes.

A compilação de consultas pode ser dividida em três etapas principais:

a) análise, na qual é construída uma árvore de análise, representando a consulta e sua estrutura;
b) reescrita de consultas, na qual a árvore de análise é convertida em um plano de consulta inicial, que normalmente é uma representação algébrica da consulta. Esse

plano inicial é então transformado em um plano equivalente que deverá exigir menor tempo para ser executado;

c) geração de plano físico, no qual o plano de consulta abstrato, freqüentemente chamado plano lógico de consulta, é transformado em um plano físico de consulta, pela seleção de algoritmos para implementar cada um dos operadores do plano lógico e pela seleção de uma ordem de execução para esses operadores. O plano físico, como o resultado da análise e do plano lógico é representado por uma árvore de expressões. O plano físico também inclui detalhes como o modo pelo qual as relações consultadas são acessadas, e se e quando uma relação deve ser classificada.

Cada uma dessas opções depende do metadados sobre o SGBD. Os metadados típicos que estão disponíveis para o otimizador de consultas incluem: o tamanho de cada relação, estatísticas como o número aproximado e a freqüência de diferentes valores para um atributo, a existência de certos índices e o *layout* dos dados no disco.

3.2 ESTRUTURA DE ARMAZENAMENTO

Um dos aspectos importantes que distinguem os SGBD de outros sistemas é a habilidade que o mesmo apresenta para lidar de forma eficiente com grande quantidade de dados.

O gerenciador de armazenamento é responsável pelo armazenamento do metadados, dos dados, índices, *logs* e uma série de outras informações a respeito das estruturas. Em paralelo está o gerenciador de buffer, que mantém porções do conteúdo do disco na memória principal.

Conforme Molina, Ullman e Widom (2001) um sistema de computador típico tem vários componentes diferentes nos quais os dados podem ser armazenados. Esses componentes têm capacidades de dados que variam por no mínimo sete ordens de grandeza e também têm velocidade de acesso variado acima de sete ou mais ordens de grandeza. O custo por *byte* desses componentes também variam, embora mais lentamente, com talvez três ordens de grandeza entre as formas de armazenamento mais econômicas e as mais dispendiosas. Não é de surpreender que os dispositivos com menor capacidade também ofereçam a maior velocidade de acesso e tenham o custo por *byte* mais alto.

3.2.1 Armazenamento volátil e não volátil

Uma distinção adicional entre dispositivos de armazenamento os define como voláteis ou não voláteis. Um dispositivo volátil "esquece" o que está armazenado nele quando a energia é desligada. Por outro lado, um dispositivo não volátil deve manter seu conteúdo intacto mesmo por longos períodos quando o dispositivo é desligado ou ocorre uma falha de energia. A questão da volatilidade é importante, porque um dos recursos característicos de um SGBD é a habilidade para reter seus dados, até mesmo na presença de falhas de energia.

Os materiais magnéticos guardarão seu magnetismo na ausência de energia, assim, dispositivos como discos e fitas magnéticas são meios não voláteis. Da mesma forma, dispositivos ópticos como CDs guardam os pontos pretos ou brancos com os quais são impressos, mesmo na ausência de energia. Na verdade, em muitos desses dispositivos é impossível alterar por quaisquer meios o que foi gravado em sua superfície. Desse modo, essencialmente todos os dispositivos de armazenamento secundário e terciário são não voláteis.

Por outro lado, a memória principal em geral é volátil. Ocorre que um *chip* de memória pode ser projetado com circuitos mais simples, se o valor do *bit* puder se degradar no decorrer de aproximadamente um minuto; a simplicidade diminui o custo por *bit* do *chip*. O que acontece na realidade é que a carga elétrica que representa um *bit* é drenada lentamente da região dedicada a esse *bit*. Como resultado, um *chip* chamado de memoria dinâmica de acesso aleatório ou DRAM (*dynamic random-access memory*) precisa ter todo o seu conteúdo lido e regravado de tempos em tempos. Se a energia for desligada, essa renovação não ocorrerá, e o *chip* logo perderá o conteúdo armazenado.

Conforme Molina, Ullman e Widom (2001) um SGBD que funciona em uma máquina com memória principal volátil deve fazer o *backup* de toda mudança em disco, antes da mudança poder ser considerada parte do banco de dados; do contrário, corre-se o risco de perder informações no caso de uma falha de energia. Como conseqüência, as modificações de consultas e bancos de dados devem envolver um grande número de gravações em disco, algumas das quais poderiam ser evitadas se não houvesse a obrigação de preservar todas as informações a toda hora. Uma alternativa é usar uma forma de memória principal que não seja volátil.

3.2.2 Manutenção dos dados

O processo de gravar um bloco é, em sua forma mais simples, análogo a ler um bloco. As cabeças de discos são posicionadas no cilindro apropriado e espera-se que o(s) setor(es) apropriado(s) gire(m) até passar sob a cabeça, mas, em vez de ler os dados sob a cabeça, utiliza-se a cabeça para gravar novos dados. Os tempos mínimo, máximo e médio para gravar são então exatamente iguais aos da leitura.

Modificação de blocos

Não é possível modificar um bloco diretamente no disco. Em vez disso, mesmo quando desejar modificar apenas alguns bytes deve-se proceder da seguinte forma:

a) ler o bloco, transferindo-o para a memória principal;
b) fazer as alterações desejadas no bloco, usando a cópia do bloco na memória principal;
c) gravar o novo conteúdo do bloco no disco;
d) se apropriado, verificar se a gravação foi feita corretamente.

O tempo total para essa modificação de bloco é, portanto, a soma do tempo necessário para ler, o tempo para executar a atualização na memória principal o tempo para gravar e, se for executada a verificação, outro tempo de rotação do disco.

Que tamanho os blocos devem ter

Conforme Molina, Ullman e Widom (2001) existem argumentos afirmando que um tamanho de bloco maior seria vantajoso, pois existe demora em cerca de meio milissegundo a transferência de um bloco de 4 K, enquanto tem-se 14 milissegundos para o tempo de busca médio e a latência rotacional. Ao duplicar o tamanho dos blocos reduz o número de operações de E/S de disco para um algoritmo como a classificação por intercalação de vários caminhos. Por outro lado, a única mudança no tempo para acessar um bloco seria o fato de que o tempo de transferência aumenta para 1 milissegundo. Então, teria-se reduzido aproximadamente à metade o tempo de duração da classificação.

Existem razões para manter o tamanho do bloco relativamente pequeno. Primeiro, não se pode usar de forma eficiente blocos que cobrem várias trilhas. Em segundo lugar, relações pequenas ocupariam apenas uma fração de um bloco, e assim poderia haver muito espaço

desperdiçado no disco. Também há certas estruturas de dados para a organização do armazenamento secundário que preferem dividir dados entre muitos blocos e assim não funcionam tão bem quando o tamanho do bloco é muito grande. De fato quanto maiores os blocos, menor o número de registros que se pode classificar pelo método de vários caminhos e duas fases. Apesar disso, à medida que as máquinas se tornam mais rápidas e discos têm maior capacidade, há uma tendência para aumentar os tamanhos de blocos.

O uso eficiente do espaço de armazenamento secundário

Conforme Molina, Ullman e Widom (2001) na maioria dos estudos de algoritmos, supõem-se que os dados estão na memória principal e que o acesso a um item de dados demora tanto tempo quanto o acesso a qualquer outro. Esse modelo de computação é chamado com freqüência "modelo de RAM" ou modelo de acesso aleatório de computação. Porém, ao se implementar um SGBD, deve-se supor que os dados não cabem na memória principal. Assim, deve-se levar em consideração o uso de espaço de armazenamento secundário e ,talvez até terciário no projeto eficiente de algoritmos. Desse modo, os melhores algoritmos para processar quantidades de dados muito grandes difere freqüentemente dos melhores algoritmos de memória principal para o mesmo problema.

Em particular, há uma grande vantagem em projetar algoritmos que limitam o número de acessos de disco, ainda que as ações executadas pelo algoritmo sobre os dados na memória principal não sejam aquilo que se poderia considerar o melhor uso da memória principal.

Registros de tamanho variável

Uma situação complexa ocorre quando os registros não têm um esquema fixo. Isto é, os campos ou sua ordem não são completamente determinados pela relação ou classe cujo registro ou cujo objeto representa. A representação mais simples de registros de tamanho variável é uma seqüência de campos marcados, cada um dos quais consiste em:

a) informações sobre a função desse campo, como:
   - o nome do atributo ou do campo,
   - o tipo do campo, se ele não for aparente a partir do nome de campo e de alguma informação prontamente disponível sobre o esquema,
   - o comprimento do campo, se ele não for aparente a partir do tipo;

b) o valor do campo.

Registros que não se encaixam em um bloco

Outro problema cuja importância vem aumentando à medida que os SGBDs são usados com maior freqüência para gerenciar tipos de dados com valores grandes: valores que com freqüência não cabem em um único bloco.

Com freqüência, esses valores grandes têm um comprimento variável mas, mesmo que o comprimento seja fixo para todos os valores do tipo, faz-se necessária usar algumas técnicas especiais para representar esses valores.

Os registros de amplitude também são úteis em situações nas quais os registros são menores que blocos, mas a compactação de registros inteiros em blocos desperdiça quantidades significativas de espaço.

Por essas duas razões, às vezes é desejável permitir que registros se dividam por dois ou mais blocos. A parte de um registro que aparece em um bloco é chamada fragmento de registro. Um registro com dois ou mais fragmentos é chamado registro com faixas e registros que não cruzam o limite de um bloco são registros sem faixas.

Modificações de registros

Muitas vezes, inserções, exclusões e atualizações de registros geram problemas especiais. Esses problemas são mais drásticos quando os registros mudam de comprimento, mas surgem até mesmo quando todos os registros e campos têm comprimento fixo.

Inserção

Primeiro, considerar-se á inserção de novos registros em uma relação (ou de forma equivalente, na extensão atual de uma classe). Se os registros de uma relação não são mantidos em nenhuma ordem particular, pode-se simplesmente encontrar um bloco com algum espaço vazio ou obter um novo bloco se não existir nenhum e co locar o registro nesse bloco. Normalmente, existe algum mecanismo para se localizar todos os blocos que contêm registros de uma dada relação ou objetos de uma classe.

Ocorre mais de um problema quando os registros devem ser mantidas em alguma ordem fixa; por exemplo, classificadas por sua chave primária. Há uma boa razão para manter os registros classificados, pois isso facilita a resposta a certos tipos de consultas. Ao inserir

um novo registro, primeiro deve-se localizar bloco apropriado para esse registro. É possível que exista espaço no bloco para inserir o novo registro. Tendo em vista que os registros devem ser mantidos em ordem, talvez tenha-se de deslizar registros pelo bloco para tornar disponível algum espaço no ponto apropriado.

Se fosse possível encontrar espaço para o registro inserido no bloco examinado, simplesmente deslizar-se-ia os registros dentro do bloco e ajustando os ponteiros na tabela de deslocamentos. O novo registro será inserido no bloco e um novo ponteiro para o registro será adicionado à tabela de deslocamentos correspondente ao bloco.

Porém, pode não haver espaço no bloco para o novo registro, nesse caso, deve-se de encontrar espaço fora do bloco. Há duas abordagens importantes para resolver esse problema, bem como combinações dessas abordagens.

a) encontrar espaço em um bloco "vizinho". Por exemplo, se o bloco B não tiver espaço disponível para um registro que precisa ser inserido em ordem classificada nesse bloco, então examine o bloco B seguinte na ordem classificada dos blocos. Se houver espaço em B mova o(s) registro(s) mais alto(s) de B para B disponível e deslize os registros em ambos os blocos. Porém, se houver ponteiros externos para registros, deve-se ficar atento e deixar um endereço de encaminhamento na tabela de deslocamentos de B para informar que um certo registro foi movido para B e onde está sua entrada na tabela de deslocamentos de B Em geral, a permissão de endereços de encaminhamento aumenta o espaço necessário para as entradas da tabela de deslocamentos;

b) criar um bloco de estouro. Nesse esquema, cada bloco B tem em seu cabeçalho um lugar referente a um ponteiro para um bloco de estouro, no qual podem ser colocados registros adicionais que teoricamente pertence a B. O bloco de estouro para B pode apontar para um segundo bloco de estouro e assim por diante.

Exclusão

Quando se exclui um registro, pode-se recuperar seu espaço. Se for usado uma tabela de deslocamentos, se os registros puderem deslizar pelo bloco, será possível compactar o espaço no bloco de tal forma que sempre exista uma região não utilizada no centro.

Se os registros não puderem deslizar, deve-se manter uma lista de espaços disponíveis no cabeçalho do bloco. Então, serão conhecidas as regiões disponíveis e qual o seu tamanho se um novo registro tiver de ser inserido no bloco.

Quando um registro é excluído, pode-se suprimir um bloco de estouro. Se registro for excluído de um bloco B ou de qualquer bloco em sua cadeia de estouro, pode-se considerar o espaço total usado em todos os blocos da cadeia. Se os registros couberem em um número menor de blocos e pude-se mover registros com segurança entre blocos da cadeia, então poderá ser executada uma reorganização da cadeia inteira.

Entretanto, existe uma complicação adicional relacionada com a exclusão, e deve-se lembrá-la independentemente do esquema que será usado para reorganizar blocos. Pode haver ponteiros para o registro excluído e, nesse caso, não se desejo que esses ponteiros oscilem ou acabem apontando para um novo registro que venha a ser inserido no lugar do registro excluído. A técnica habitual é colocar uma lápide no lugar do registro. Essa lápide é permanente. Ela deve permanecer até o banco de dados inteiro ser reconstruído.

O local em que a lápide é colocada depende da natureza dos ponteiros de registros. Se os ponteiros indicam locais fixos a partir dos quais a localização do ponteiro é encontrada, então deve-se inserir a lápide nesse local fixo.

Se precisar substituir registros por lápides, será aconselhável ter bem no início do cabeçalho do registro um *bit* que sirva como uma lápide; isto é, ele será 0 se o registro não for excluído, enquanto 1 significará que o registro foi excluído. Então, apenas esse *bit* deve permanecer onde o registro costumava começar, e os *bytes* subseqüentes poderão ser reutilizados em outro registro.

Atualização

Quando um registro de comprimento fixo é atualizado, não há nenhum efeito sobre o sistema de armazenamento, porque sabe-se que ele pode ocupar exatamente o mesmo espaço que ocupava antes da atualização. Contudo, quando um registro de comprimento variável é atualizado, tem-se todos os problemas associados com a inserção e a exclusão, exceto pelo fato de nunca ser necessário criar uma lápide para a versão antiga do registro.

Se o registro atualizado for maior que a versão antiga, pode-se ter necessidade de criar mais espaço em seu bloco. Esse processo pode envolver o deslizamento de registros ou mesmo a criação de um bloco de estouro. Se porções de comprimento variável do registro estiverem armazenadas em outro bloco, talvez seja necessário mover elementos nesse bloco ou criar um novo bloco para armazenar campos de comprimento variável. Reciprocamente, se o registro for reduzido em conseqüência da atualização, tem-se as mesmas oportunidades que no caso de uma exclusão para recuperar ou consolidar espaço, ou ainda para eliminar blocos de estouro.

3.3 ACESSO AOS DADOS

Comumente, um banco de dados consiste em um processo servidor que fornece dados de armazenamento secundário a um ou mais processos clientes que são aplicativos que usam os dados. Os processos entre o servidor e cliente podem estar em uma única máquina, ou o servidor e os diversos clientes podem estar distribuídos por muitas máquinas.

Conforme Molina, Ullman e Widom (2001) o aplicativo cliente utiliza um espaço de endereços “virtual” convencional, em geral de 32 bits, ou cerca de 4 bilhões de endereços diferentes. O sistema operacional ou SGBD decide que partes do espaço de endereços estão localizadas atualmente na memória principal, e o hardware mapeia o espaço de endereços virtual para localizações físicas na memória principal.

Os dados do servidor residem em um espaço de endereços do banco de dados. Os endereços desse espaço se referem a blocos e possivelmente a deslocamentos dentro do bloco. Os endereços desse espaço de endereços podem ser representados de várias formas:

Endereços físicos são *strings* de *bytes* que permitem determinar a localização dentro do sistema de armazenamento secundário na qual o bloco ou o registro pode ser encontrado. Um ou mais bytes do endereço físico são usados para indicar cada um dos seguintes itens:

- o host ao qual o espaço de armazenamento está associado (se o banco de dados estiver armazenado em mais de uma máquina),
- um identificador para o disco ou outro dispositivo no qual o bloco está localizado,
- o número do cilindro do disco,
- o número da trilha dentro do cilindro (se o disco tiver mais de uma superfície),
- o número do bloco dentro da trilha,
- endereços lógicos. Cada bloco ou registro tem um "endereço lógico", um string de bytes arbitrário com algum comprimento fixo. Uma tabela de mapas, armazenada no disco em um local conhecido, relaciona endereços lógicos e físicos;

Os endereços físicos são longos, oito bytes é o mínimo que se poderia usar se fosse incorporado todos os elementos listados. Alguns sistemas utilizam até 16 *bytes*. Por exemplo, imagine um banco de dados de objetos projetado para durar 100 anos. No futuro, o banco de dados poderá crescer para englobar um milhão de máquinas, e cada máquina poderá ser rápida o bastante para criar um objeto a cada nanossegundo. Esse sistema criaria objetos que exigiriam no mínimo 10 *bytes* para representar endereços. Tendo em vista que provavelmente iríamos preferir reservar alguns *bytes* para representar o *host*, outros para representar a unidade de armazenamento e assim por diante, uma notação de endereços racional provavelmente utilizaria muito mais de 10 *bytes* para um sistema dessa escala.

4 DESENVOLVIMENTO DO TRABALHO

Este capitulo apresenta os requisitos, especificação e as etapas para o desenvolvimento do trabalho.

4.1 REQUISITOS DO SOFTWARE

O protótipo deverá:

a) permitir atribuir permissões para um usuário (requisito funcional - RF);

b) permitir a criação de banco de dados (RF);

c) permitir a criação de tabelas dentro do banco de dados (RF);

d) permitir a inserção de dados da tabela (RF);

e) permitir a exclusão de tabelas do banco de dados (RF);

f) permitir a realização de consultas a um banco de dados usando a linguagem SQL (RF);

g) funcionar em ambiente Windows (requisito não funcional - RNF);

4.1.1 ESPECIFICAÇÃO

Para realizar a especificação do protótipo de banco de dados cliente / servidor, utilizou-se a ferramenta Rational Rose, através da Linguagem de Modelagem Unificada (UML).

A seguir são apresentados os diagramas de caso de uso, diagramas de classes e diagrama de seqüência.

4.1.2 DIAGRAMA DE CASOS DE USO

A seguir (figura 2) é apresentado o diagrama de caso de uso do protótipo implementado.

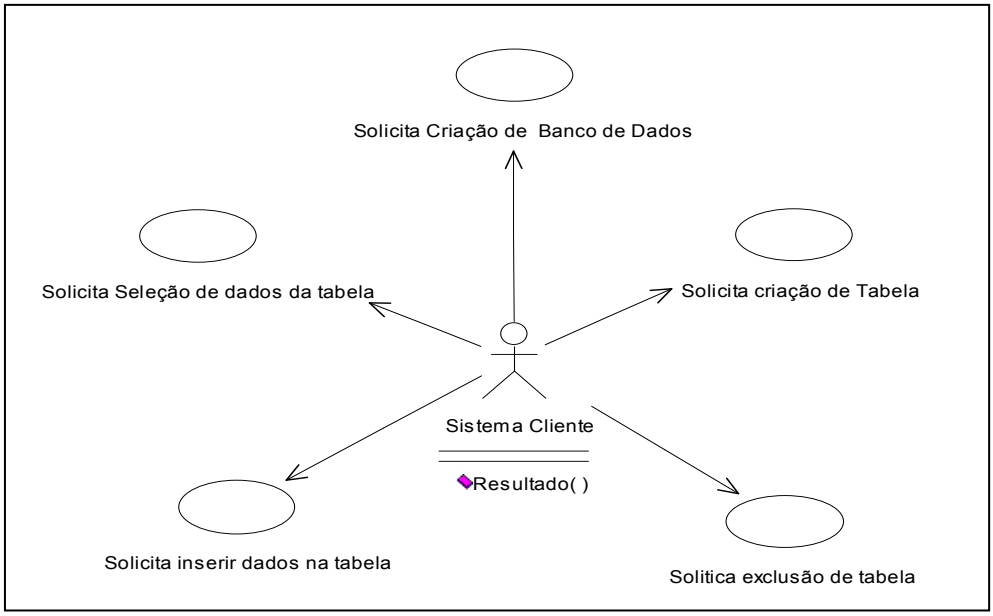


Figura 2 – Diagrama de caso de uso do sistema cliente e servidor

No protótipo desenvolvido existem dois personagens principais:

a) sistema cliente: é o responsável por enviar todas as solicitações para o sistema servidor em formato de comando SQL, e este aguarda o retorno para exibir o resultado para o usuário;

b) sistema servidor: é o responsável por receber todos os comandos SQL dos clientes e fazer o controle de identificação e autenticação do mesmo, para validar e executar os comandos SQL para retornar o resultado para o cliente solicitante.

4.1.3 DIAGRAMA DE CLASSES

Para especificação do protótipo dividiu-se em dois diagramas de classes. O primeiro trata do gerenciador de armazenamento dos dados e o segundo do *parser* (compilador).

A seguir (figura 3) é apresentado o diagrama de classes do gerador de armazenamento dos dados.

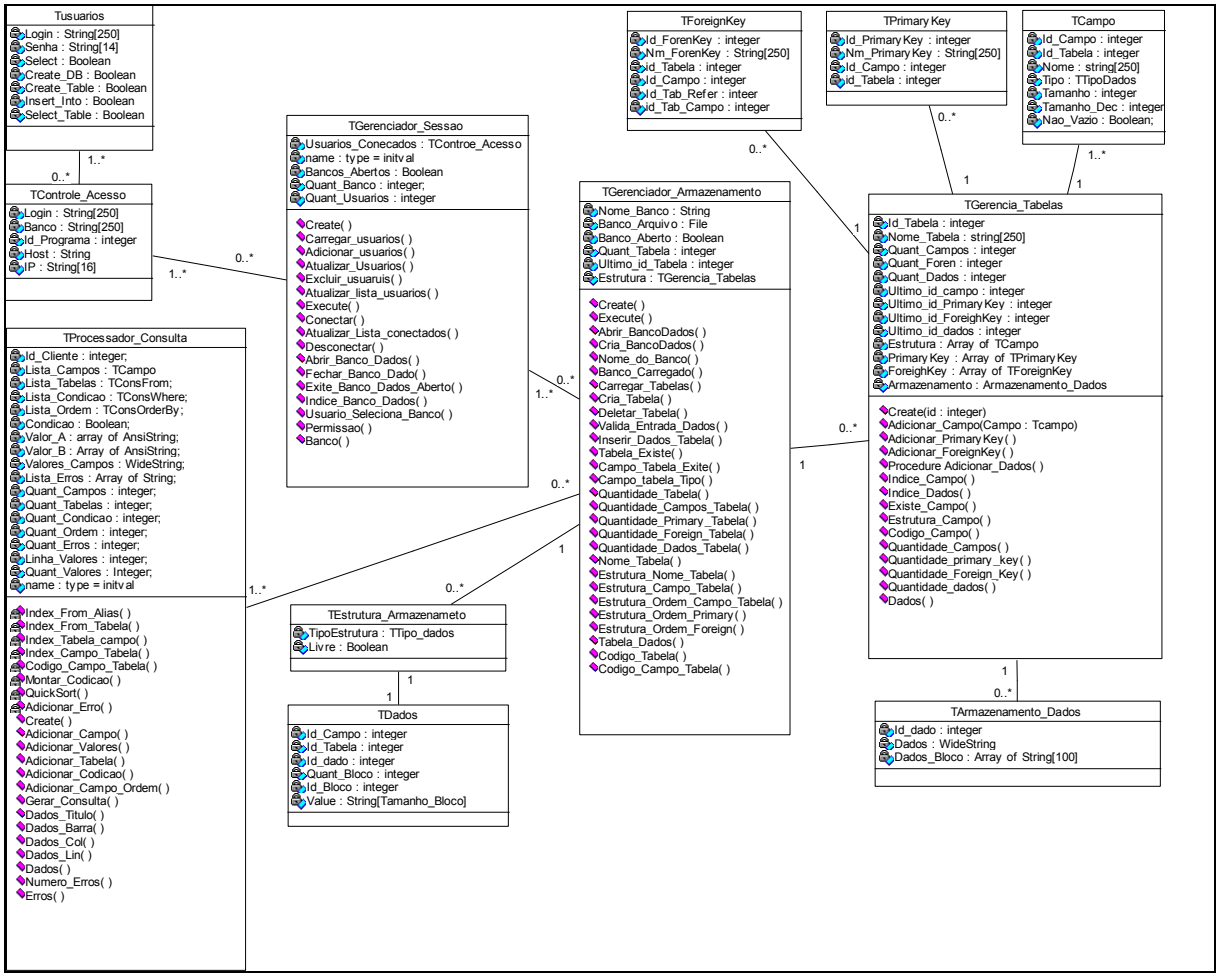


Figura 3 – Diagrama de classes responsáveis por gerenciar os dados

A seguir são descritas as classes do modelo:

Tusuário: esta classe é responsável por guardar o dados do usuário.

TControle_Acesso: esta classe é responsável por guardar o dados de todos os usuários conectados dentro do banco de dados.

TGerenciador_Secao: esta classe é responsável em manter atualizados todos os dados de clientes e bancos conectados, o que permite ao servidor identificar o usuário e a que banco ele esta conectado e que permissão ele possui dentro do servidor para poder transmitir de forma íntegra os pedidos solicitados.

TProcessador_Consultas: a classe é responsável por coletar todas as informações do comando SELECT, para permitir a geração da consulta usando todos os dados solicitados pelo cliente como campos, tabelas, condições e ordem.

TForeignKey : a classe é responsável em manter informação referente ao relacionamento entre as tabelas.

TPrimaryKey: a classe é responsável em manter informação do(s) campo(s) que não poderão ter duplicação de valores.

Tcampo: a classe é responsável em manter informação da estrutura de todo o campo existente dentro da tabela;

TArmazenamento_Dados: a classe é responsável por armazenar os dados extraídos da base de dados.

TGerenciador_Armazenamento: a classe é responsável em manter controle de todas as tabelas e armazenamento de toda a estrutura das tabelas e dados nela armazenados.

TGerencia_Tabelas: a classe é responsável em manter a estrutura e os dados armazenados.

TDados: a classe é responsável por remontar os dados armazenados em disco.

TEstrutura_Armazenamento: a classe é responsável por identificar o tipo de estrutura a ser armazenado em disco.

A seguir (figura 4) é apresentado o diagrama de classes do *parser* (compilador) do protótipo.

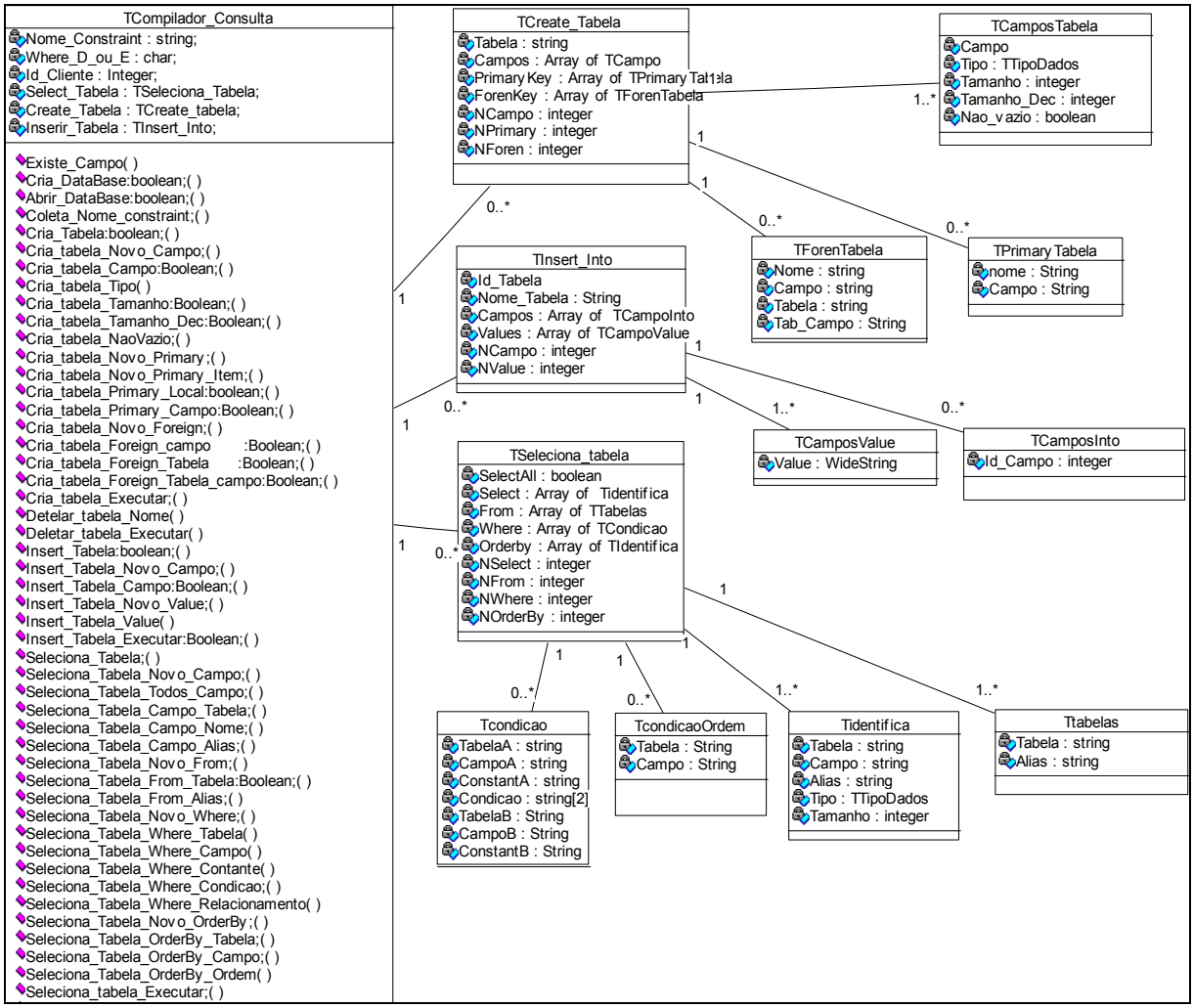


Figura 4 – Diagrama de classe responsável pelo compilador de comandos

A seguir são descritas as classes do modelo:

TCompilador_Consulta: a classe é responsável por coletar os *tokens*, analisar e executar os comandos recebidos.

TCamposTabela: a classe é responsável por manter os dados dos campos coletados pelo comando *CREATE TABLE*.

TPrimaryTabela: a classe é responsável por manter os dados das *primary key* existentes dentro comando *CREATE TABLE*.

TForenTabela: a classe é responsável por manter os dados das *foreign key* existentes dentro comando *CREATE TABLE*.

TCreate_Tabela: a classe é responsável por manter a estrutura existente dentro comando *CREATE TABLE*.

TCamposInto: a classe é responsável por manter os dados dos campos a serem inseridos pelo comando *INSERT INTO*.

TCamposValue: a classe é responsável por manter o valor dos dados dos campos a serem inseridos pelo comando *INSERT INTO*.

TInsert_Into: a classe é responsável por manter a estrutura existente dentro comando *INSERT INTO*.

Tidentificao: a classe é responsável por manter os dados dos campos selecionados pelo comando *SELECT*.

TTabelas: a classe é responsável por manter os dados das tabelas selecionadas pelo comando *SELECT*.

Tcondicao: a classe é responsável por manter os dados das condições selecionadas pelo comando *SELECT*.

TcondicaoOrdem: a classe é responsável por manter os dados das ordens dos campos selecionadas pelo comando *SELECT*.

TSeleciona_tabela: a classe é responsável por manter a estrutura existente dentro comando *SELECT*.

4.1.4 DIAGRAMA DE SEQUENCIA

A seguir (figura 5) é apresentado o diagrama de seqüência para criação de sessão entre o cliente e o servidor.

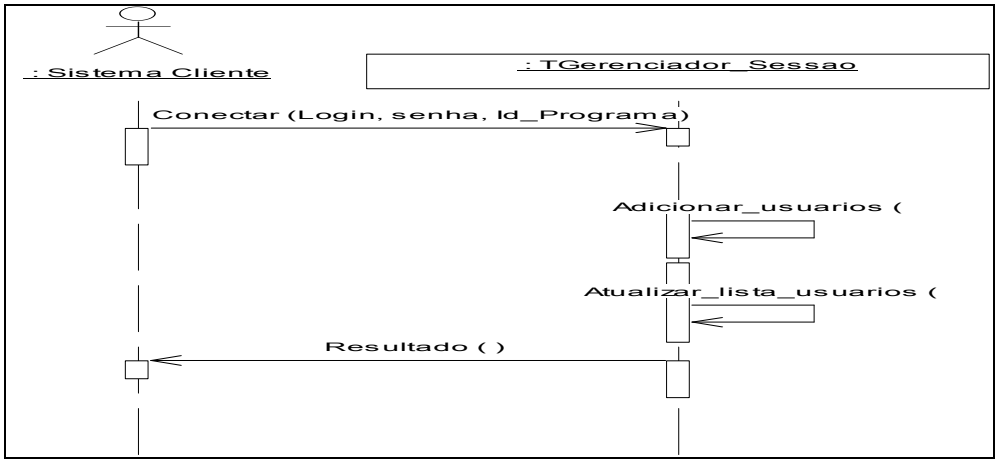


Figura 5 – Diagrama de seqüência para criação de sessão entre o sistema cliente / servidor.

A seguir (figura 6) é apresentado o diagrama de seqüência para criação de um novo banco de dados.

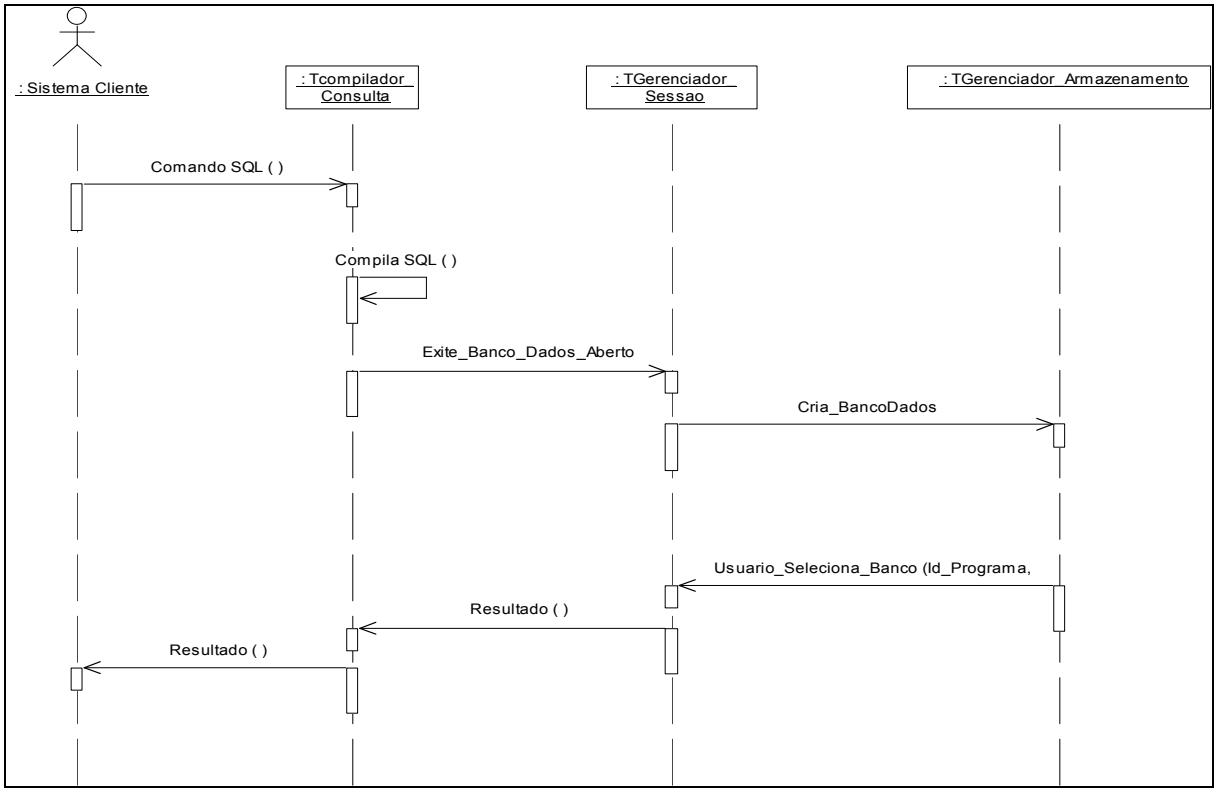


Figura 6 – Diagrama de seqüência para criação de um novo banco de dados.

A seguir (figura 7) é apresentado o diagrama de seqüência para criação de tabela.

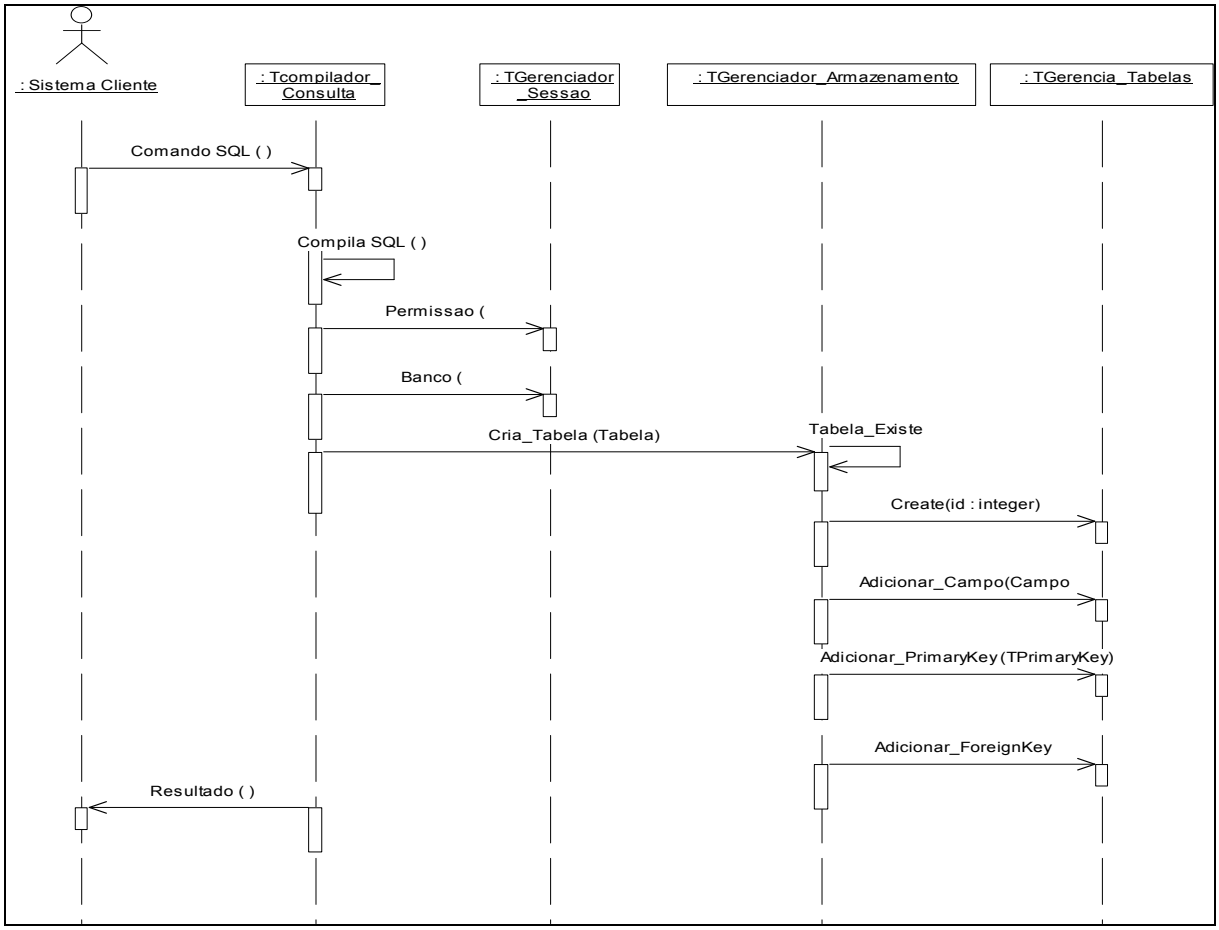


Figura 7 – Diagrama de seqüência para criação de uma nova tabela dentro do SGBD.

A seguir (figura 8) é apresentado o diagrama de seqüência para exclusão de tabela.

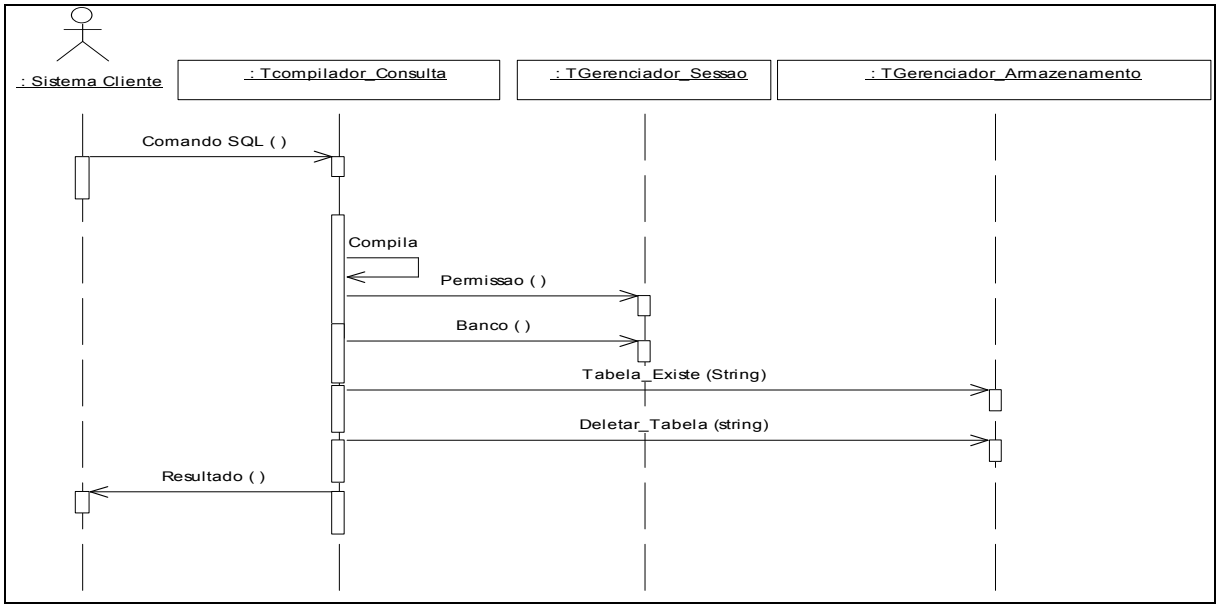


Figura 8 – Diagrama de seqüência para exclusão tabela dentro do SGBD.

A seguir (figura 9) é apresentado o diagrama de seqüência para adicionar dados dentro da tabela.

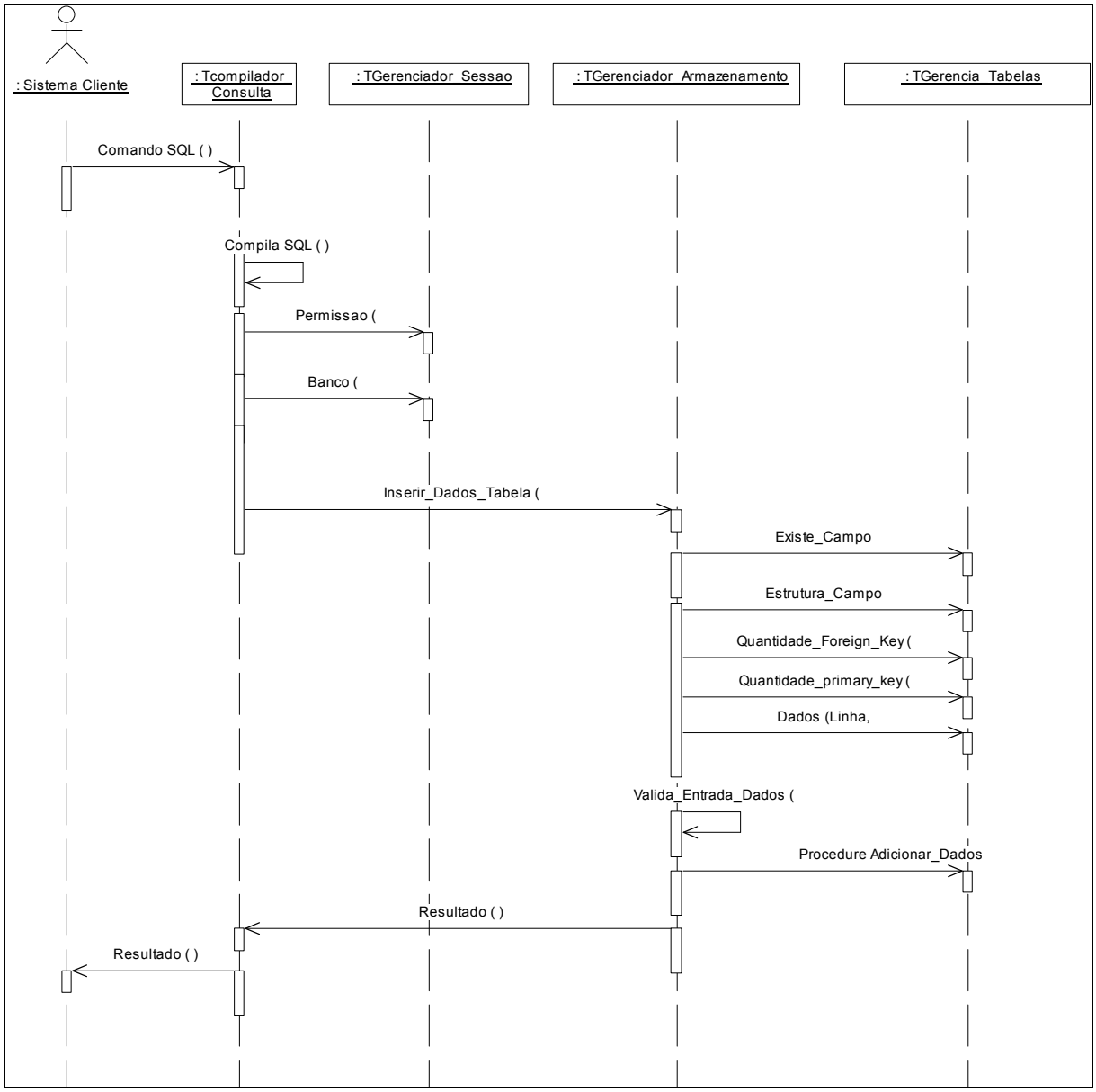


Figura 9 – Diagrama de seqüência para adicionar dados dentro da tabela.

A seguir (figura 10) é apresentado o diagrama de seqüência para seleção de dados dentro da tabela.

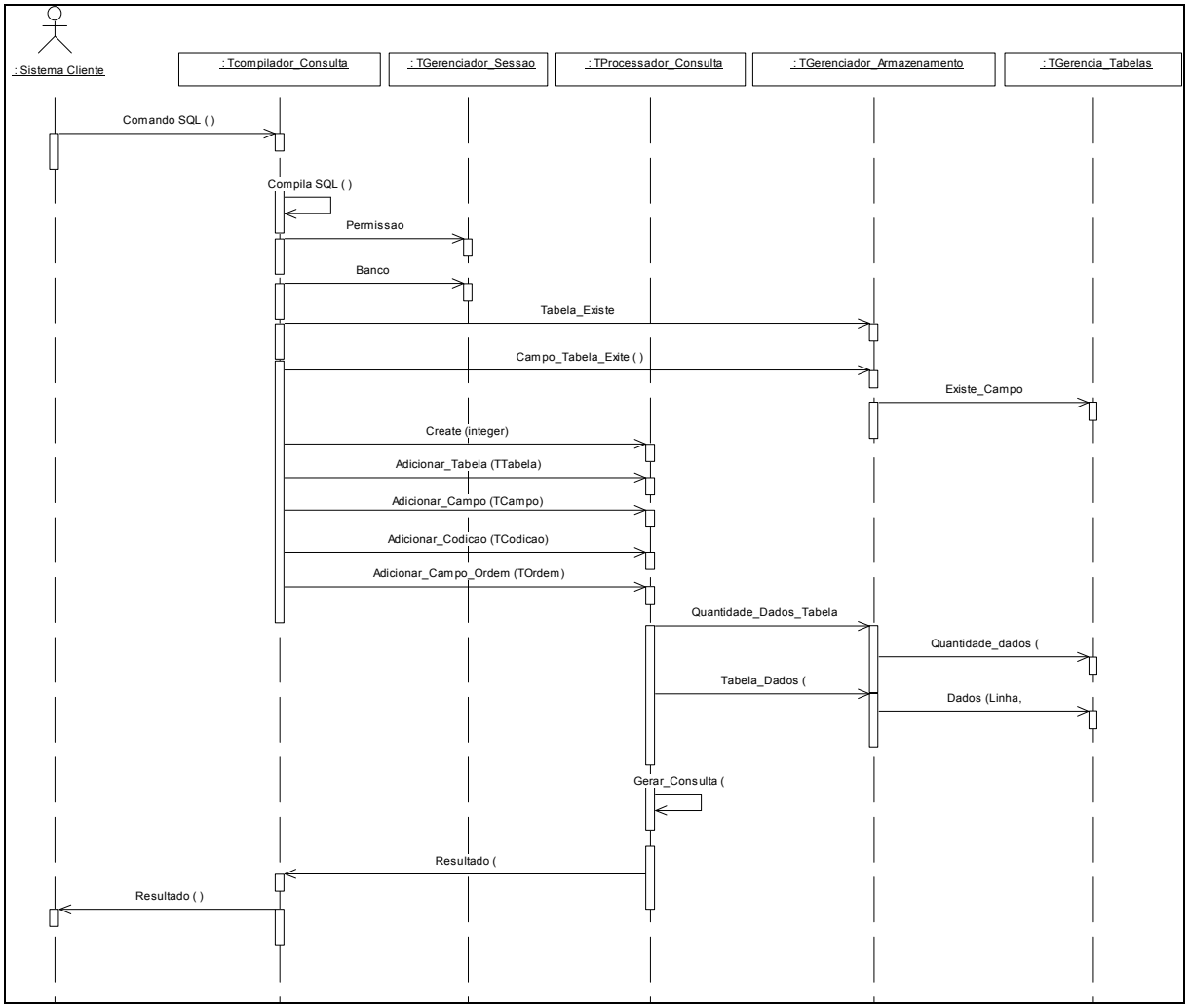


Figura 10 – Diagrama de seqüência para seleção de dados dentro do SGBD.

4.2 IMPLEMENTAÇÃO

A necessidade de desenvolver mecanismos para gerenciamento é fundamental para o banco de dados, pois define para cada gerenciador responsabilidades a serem tratadas dentro do banco de dados. Estes gerenciadores devem controlar e ter acesso às informações entre si, de forma independente. Para permitir esta independência de execução sem afetar os demais gerenciadores foram implementados recursos de *threads*, permitindo que vários processos possam ser executados concorrentemente, impedindo que algum serviço de gerência seja interrompido, o que é vital para um banco de dados relacional cliente / servidor, para distribuir informações pela rede sem a necessidade de bloquear a execução de outros gerenciadores. Cada controle que compõe o banco de dados é responsável por tratar de serviços bem distintos, como compilador, gerência de *buffer* (memória), gerência de armazenamento, gerência de seções e processador de consulta.

Reconhecimento e interpretação

Para desenvolver o protótipo foi necessário utilizar uma ferramenta que auxiliasse na construção léxica e sintática dos comandos em SQL. A ferramenta utilizada foi o *COCOR* que permite definir gramática para execução dos comandos. A ferramenta utiliza técnica de compiladores *LL(1)* que é um modelo baseando em tabelas. O *COCOR* é compatível com o *Delphi* que é um ambiente de programação com grandes recursos disponíveis para desenvolver este protótipo.

Com a definição da BNF a ferramenta gera um arquivo em linguagem pascal que contém todas as regras sintáticas a qual foi incorporada ao servidor do protótipo para validar os comandos solicitados pelo programa cliente. A tabela 1 apresenta as mensagens de erros definidas no protótipo.

Com a incorporação deste recurso, foram implementadas rotinas para capturar os *tokens* para identificar a ação semântica dentro do servidor de banco de dado. Para isto foram utilizados recursos de programação da ferramenta *Delphi*, utilizando matrizes dinâmicas. Com a construção da matriz dinâmica de forma estruturada, foi possível identificar todo os dados como tabelas, campos e tipos de dados utilizados dentro do gerenciador de banco de dados.

Tabela 1 – Erros tratados dentro do protótipo

Nº	Mensagens de erros
1	*Fim inesperado do comando*
2	*ON" ou "USING" a cláusula não podem ser usados com o "NATURA"*
3	*a palavra chave "NULL" não pode ser usado aqui - use "IS" ou "IS NOT"*
4	*Banco de Dados não existe*
5	*Banco de Dados já existe*
6	*O banco de dados não esta aberto*
7	*Tabela já existe*
8	*Tabela não existe*
9	*Campo não existe*
10	*Campo já foi utilizado*
11	*Campo não existem ou tipo incompatível com a tabela relacionada*
12	*Problema de validação*
13	*Tipo de Dado incompatível*
14	*Tamanho incompatível*
15	*Sem Permissão*
16	*Tabela esta sendo relacionada com uma ou mais tabela*
17	*Usuário não tem permissão para criar banco de dados*
18	*Usuário não tem permissão para criar tabela*
19	*Usuário não tem permissão para inserir dados*
20	*Usuário não tem permissão para selecionar dados*
21	*O comando SQL apresenta erro*

O interpretador é responsável por identificar o que o usuário solicita, permitido que o banco de dados possa executar corretamente o que lhe foi pedido. Conforme Silberschatz, Korth e Sudarschan (1999), a consulta é um dos processos que mais consome recursos do banco de dados, pois é necessário passar por várias etapas, antes de chegar ao resultado final. Estas etapas são: o analisador léxico, onde é coletado todos os *tokens*, o analisador sintático, onde é verificado a estrutura dos comandos e o analisador semântico onde é identificado o que o sistema deve fazer.

A execução do compilador dentro do banco de dados fornece todas as informações necessárias para o banco de dados poder compreender o que deve fazer. A análise vai desde formular consultas até permitir a manutenção de dados armazenados, pois o compilador permite ao banco de dados coletar nomes de tabelas e campos, que serão utilizados para gerar consultas de uma ou diversas tabelas relacionadas. Tais recursos oferecem ao usuário liberdade na formulação da consultas, desde que obedecendo às regras da BNF definido neste protótipo. É importante salientar que não esta sendo previstos todos os recursos dos SGBDs existentes no mercado.

Estrutura de armazenamento

O armazenamento é um dos principais objetivos do banco de dados. A integridade da informação é essencial sendo que o desenvolvimento da estrutura de armazenamento tem que corresponder às necessidades do desenvolvedor que utiliza o sistema de banco de dados, por

ter necessidade de armazenar diversos tipos de dados como texto, números e datas. Este tipos de dados possuem tamanho variado. Para o banco de dados conseguir armazenar as informações sem oferecer grandes limitação no tamanho do armazenamento de cada campo da tabela, foi desenvolvido um mecanismo de armazenamento em formato de arquivos binários. Utilizando este formato é possível melhorar os recursos de armazenamento onde permite que campos tenham tamanhos variados e que num mesmo arquivo de armazenamento tenha a estrutura das tabelas e do seus respectivos dados armazenados. O banco de dados deve se preocupar com outros itens como o relacionamento entre tabelas do banco e o controle das chaves primárias e estrangeiras. O banco de dados deve permitir ao usuário definir os relacionamentos necessários e este deve manter um controle completo, caracterizando a função de integridade referencial.

Para implementar a estrutura de armazenamento foi utilizado recursos da linguagem *Object Pascal*. Os dados são armazenados em arquivos binários, o que permite um melhor aproveitamento do espaço físico no disco, oferecendo recursos de armazenamento para diversos tipos de dados como número, caracteres e datas.

A tabela 2 apresenta os tipos de dados e as respectivas capacidades de armazenamento.

Tabela 2 – Capacidade de armazenamento para cada tipos de dados

TIPO DE DADOS	CAPACIDADE MÁXIMA
CHAR	1073741824 caracteres
CHARACTER	1073741824 caracteres
VARCHAR	1073741824 caracteres
VARCHAR2	1073741824 caracteres
INTEGER	9 números
INT	9 números
SMALLINT	9 números
NUMERIC	19 números
NUMBER	19 números
DATE	99/99/99 ou 99/99/9999
TIME	99:99:99
TIMESTAMP	99:99:99

Foi implementado um modelo de armazenamento muito similar aos utilizados nos demais bancos de dados conhecidos no mercado, pois o sistema de armazenamento é baseado em blocos, o que permite que os campos armazenem informações de tamanhos variados de 1 caractere até 2 gigabyte de caracteres. O conteúdo dos campos são armazenados em blocos, onde, dependendo do tamanho do valor a ser armazenado, podem ser utilizados vários blocos até que armazene todo o conteúdo. O quadro 1 apresenta a estrutura de armazenamento de um bloco.

Tipo de armazenamento	Identificador tamanho	Quantidade de Bloco	Bloco	Valor[Tamanho]

Quadro 1 – Estrutura de armazenamento físico no protótipo desenvolvido

No aspecto de armazenamento de dados foi implementando uma técnica de armazenamento semelhante a do trabalho de Hübner e Hugo (1992), que utiliza uma biblioteca do *Pascal TPVArray* (*turbo pascal virtual array*) que trata de uma estrutura de matriz dinâmica (tamanho variado), onde toda informação contida dentro da matriz é armazenada em disco. O quadro 2 apresenta a estrutura de armazenamento dos dados.

Campos(1)	Campo(2)	Campo(N)
Dados(1).Campo(1)	Dados(1).Campo(2)	Dados(1).Campos(N)
Dados(N).Campo(1)	Dados(N).Campo(1)	Dados(N).Campo(N)

Quadro 2 – Estrutura de acesso às informações armazenadas

Acesso aos dados

No sistema servidor do banco de dados foi desenvolvido um mecanismo que identifica todos os usuários conectados e que banco estão acessando. Isto se faz necessário pois um mesmo usuário pode estar conectado com o sistema cliente em diferentes computadores dentro da rede e em cada um deles acessando um banco diferente. Foi necessário criar um controle que permitisse identificar e controlar todos os acessos. Para criação deste gerenciador de sessão foi necessário identificar principalmente o código do cliente conectado. Através do componente *socket* do *Delphi,* é possível coletar do sistema operacional o código do sistema cliente que está sendo utilizando para fazer esta conexão. Isto permite a identificação do aplicativo independente do local onde esteja na rede, permitindo ao sistema servidor identificar para quem deve ser transmitido o resultado dos comandos enviados.

O sistema servidor identifica o cliente e solicita que o mesmo entre com o *login* e a senha, para o servidor constar que o usuário realmente esteja cadastrado dentro do servidor, e que a senha esteja correta para permitir que o acesso seja completado. Após autenticação da senha é criado um caminho virtual entre o servidor e o cliente que recebe um código identificador, para confirmar a conexão. Este procedimento foi adotado pois o servidor do banco de dados implementa um controle de permissões que determina o que cada usuário pode executar dentro do banco de dados, como criar um novo banco de dados, incluir e excluir tabelas, inserir dados e selecionar dados. O sistema servidor coleta quatro itens através do componente *socket* como é mostrado na tabela 3.

Tabela 3 – Dados coletados do componente *Socket*

Componente TServerSocket	**Descrição**
socket.SocketHandle	Código de identificação do programa cliente
socket.ReceiveText	Comandos enviados pelo sistema cliente
socket.RemoteHost	Endereço *Host* do Cliente
socket.RemoteAddress	Endereço Ip do Cliente

Após o sistema servidor identificar o endereço virtual com o cliente, o servidor aguarda que o sistema cliente solicite um banco a ser aberto. Com a solicitação de abertura do banco, o sistema servidor verifica se o mesmo já se encontra aberto para integrar ao usuário. Caso o banco não esteja aberto o servidor cria uma nova instância e associa ao cliente, permitindo que acesse os dados contidos dentro do banco. Com este procedimento o

servidor evita criar muitas instâncias do mesmo banco de dados, pois dificultaria o controle dos dados sobre a mesma e faria o computador ocupar mais memória sem necessidade. Conforme pode-se observar na figura 11 alguns sistemas clientes acessando bases no servidor de banco de dados.

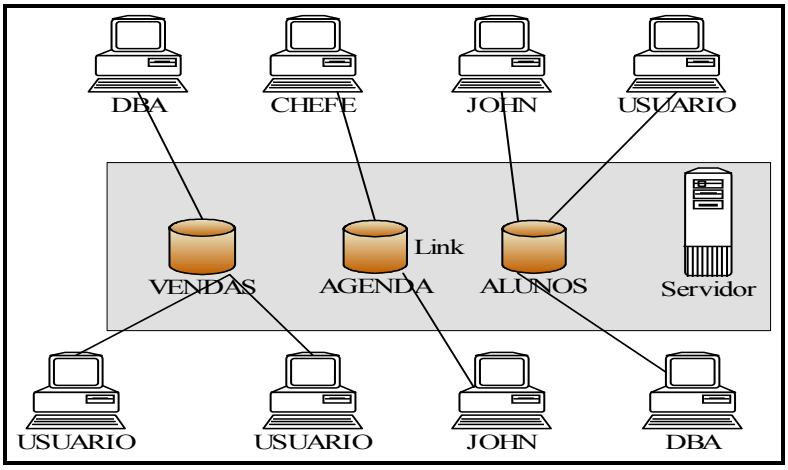


Figura 11 – O servidor de banco de dados sendo acessado por vários clientes.

Após executar todas as rotinas de identificação e conexão o sistema cliente terá condições de enviar comandos para o sistema servidor solicitando pedidos através da linguagem *SQL*. O acesso aos dados se dá através do comando *SELECT* como mostra o quadro 3, onde foi atribuído algum dos vários recursos de um SGBD como:

a) recurso para selecionar todas ou partes dos campos das tabelas;
b) recurso para definir nome de alias para os campos solicitados;
c) recurso para selecionar várias tabelas dentro do banco;
d) recurso para definir nome de alias das tabelas selecionadas;
e) recurso para inserir várias condições entre campos e valores constantes;
f) recurso para ordenar tabelas por ordem de campos de forma crescente e decrescente.

```
"SELECT"        ( "*"
                      | [(tabela | alias) "."] [Campo] ["as" alias]
                      | [lista Campos]
                )
"FROM"          (     [tabela] [alias]
                      | [ "," [lista tabelas] ]
                )
"WHERE"         (     [ (tabela | alias) "."] ( [campo] | [constante] )
                      Condicao
                      [ (tabela | alias) "."] ( [campo] | [constante] )
                      (
                            [condicao] [Lista Condicao]
                      )
                )
"ORDER BY"      (     [ (tabela | alias) "."] ( [campo] )
                      | [Lista de campos]
                )
```

Quadro 3 – Estrutura do comando *SELECT*.

O gerador de consulta é o mecanismo responsável por gerar a consulta a ser enviada para o cliente. O gerador de consulta necessita que o compilador envie os dados coletados na seguinte ordem:

a) tabelas selecionadas;

b) campos selecionados;

c) condições definidas;

d) campos a serem ordenados.

Após o gerador de consulta receber estes dados ele compara se os campos declarados existem dentro das tabelas declaradas. As tabelas são verificadas no próprio compilador, para evitar que envie para gerador de consulta quando tiver qualquer tipo de erro envolvendo tabelas, estrutura de comandos ou valores.

Após a coleta dos campos e tabelas são coletados as restrições e os campos a serem ordenados (caso haja). A relação de várias tabelas torna a consulta de forma cartesiana, quer dizer que para cada tabela declarada é criada uma multiplicação entre todas as tabelas envolvidas. Esta multiplicação tem um objetivo de criar um recurso chamado relacionamento entre tabelas, que é feita pelo comando *where* como pode-se observar o resultado no quadro 4.

Alunos

Cod	Nome	Materia
1	John	1
2	André	2

Materias

Cod	Materia
1	BCC
2	Medicina
3	Engenharia

União entre as tabelas pelo comando Where.
Aluno.materia = materia.cod

Alunos	Matéria
John	BCC
André	Medicina

Alunos X Materias

Cod	Nome	Materia	Cod	Matéria
1	John	1	1	BCC
1	John	1	2	Medicina
1	John	1	3	Engenharia
2	André	2	1	BCC
2	André	2	2	Medicina
2	André	2	3	Engenharia

Quadro 4 – A estrutura para relacionamento entre várias tabelas.

Como não foi encontrado em nenhum material que apresentasse um algoritmo para fazer o relacionamento entre várias tabelas, foi implementado no protótipo um algoritmo para fazer esta função. O algoritmo apresentou um bom desempenho e o resultado final demonstrou muita fidelidade dentro dos comandos especificados. O algoritmo foi implementado para executar o relacionamento através de dados armazenados em forma matriz, onde sua característica é de ser uma função recursiva, ou seja, chama a si própria. A cada chamada que é executada o algoritmo aponta para a próxima tabela relacionada até chegar à última onde processa os campos e preenche o valor de saída e retorna para a tabela anterior.

O algoritmo compara as condições com os campos coletados, para indicar se podem ser inseridos no valor de saída do *select*. Esta coleta é feita cada vez que avança o registro dentro de cada tabela, onde são atualizados os valores dos campos das condições pertencentes àquela tabela. No final da rotina, antes de inserir os dados, é verificada se a condição dos campos selecionados está dentro das regras da condição para incluir os campos no resultado final. Pode-se visualizar no quadro 5 o algoritmo implementado no protótipo.

```
Gerador_Consulta(Index_Table : Inteiro)
    Se  Index_Table < Quant_Tabelas
        Se Tabela[Index_Table].Quantidade_Dados = 0
            Preencha_Nulo
        Faça Index_Dados = 0 Até Tabela[Index_Table].Quantidade_Dados-1
            Monta_Condicao (Index_Table, Index_Dados)
            Se (Index_Table = Quantidade_Tabela-1) E (Quant_Tabela <> 1 )
                Se Não(Condicao)
                    Retorna Proximo_Dado
            Faça Index_Campo To Select_Quantidade_Campos-1
                Se Campo_Select[Index_Campo].Index_Table = Index_Table
                    Valor_Select[Linha, Index_Campo]  := Dados_Tabela[Index_Table, Index_Campo]
                Senão
                Se Linha > 0
                    Valor_Select[Linha, Index_Campo] := Valor_Select[Linha -1, Index_Campo]

            Se Index_Table +1 < Quantidade de Tabelas then
                Gerador_Consulta(index_Table+1)

            Se condição = Verdadeira
                Se Index_Dados < Tabela[Index_Table].Quantidade_dados
                    Linha := Linha +1
                Condição = Falso
```

Quadro 5 – Função utilizada para o relacionamento entre várias tabelas.

Após tratar as condições (quando houver), o gerador de consulta verifica se há campos a serem ordenados. Este ordenamento é feito através do algoritmo *QuickSort*, que utiliza uma técnica de grande eficiência na ordenação dos campos selecionados. Foi feita adaptação no algoritmo para que aceita-se vários campos, já que o algoritmo original aceita somente um valor de cada vez.

No protótipo foi implementado o comando *CREATE TABLE*, que é responsável por criar novas tabelas dentro do banco de dados. Dentro deste comando foram implementadas algumas validações como:

a) checa se a tabela já existe dentro do banco de dados;

b) checar se os campos não foram duplicados;

c) checar se o tamanho dos campos é compatível com o tipo de dados;

d) checar se o campo do *primary key* existe;

e) checar se a tabela e campo do *foreign key* existe;

f) checar se a estrutura do comando é válida.

A estrutura do comando pode ser vista na quadro 6.

```
"CREATE TABLE"  [Nome da tabela]
                "("
                    [nome campo] ( [tipo dados]  ( [tamanho] [ tamanho decimal] ) )    ["NOT NULL"]
                                                                                       (  [ Primary Key]
                                                                                          | [Foreign Key]  )
                                                                                       )
                    | ( "," [ lista campos] )
                    | (
                        ","
                        | [Nome Constraint] [ Primary Key]
                        | ["," (Lista primary key) ]
                        | [Nome Constraint] [ Foreign Key]
```

Quadro 6 – Estrutura do comando *CREATE TABLE*.

No protótipo foi implementado o comando *INSERT INTO*, que é responsável por inserir novos dados dentro da tabela. Neste comando foi desenvolvido duas formas para os armazenar os dados:

a) indicar os campos a receber o valor;
b) sem indicar os campos, onde o valor será inserido pela ordem de cadastro dos campos.

Dentro do comando foi necessário fazer várias validações antes de inserir os valores dentro da tabela como:

a) checar se a tabela a ser inserido o valor existe;
b) checar se os campos existem dentro da tabela;
c) checar se o valor a ser inserido é compatível com o campo;
d) checar se o valor já existe dentro da chave primaria;
e) checar se o valor existe dentro da referencia do *foreign key*.

A estrutura do comando pode ser vista na quadro 7.

```
"INSERT INTO"   [Nome Tabela]
                                [    "("
                                        [Nome campo]
                                        | ( "," [Lista campos]
                                    ")"
                                ]
"VALUES"        "("
                    [Valor]
                    | ("," Lista Valores)
                ")"
```

Quadro 7 – Estrutura do comando *INSERT INTO*.

No protótipo foi implementado o comando *CREATE DATABASE*, que é responsável por criar um novo banco de dados para armazenamento de novas tabelas e dados. Neste comando somente se preocupa em verificar se existe um banco de dados com o mesmo nome. A estrutura do comando pode ser vista na quadro 8.

“CREATE DATABASE” [Nome do Bando]

Quadro 8 – Estrutura do comando CREATE DATABASE.

No protótipo foi implementado o comando *OPEN DATABASE*, que é responsável por abrir um banco de dados já existente. Neste comando somente se preocupa em verificar se existe um banco de dados criado. A estrutura do comando pode ser vista na quadro 9.

“OPEN DATABASE” [Nome do Banco]

Quadro 9 – Estrutura do comando OPEN DATABASE.

No protótipo foi implementado o comando *DROP TABLE*, que é responsável por excluir tabelas existentes dentro do banco de dados. Este comando faz 2 validações antes de excluir a tabela como:

a) checa se a tabela existe;
b) checa se a tabela não esta sendo referenciada por outra.

A estrutura do comando pode ser vista na quadro 10.

“DROP TABLE” [Nome da tabela]

Quadro 10 – Estrutura do comando *DROP TABLE*.

No protótipo foi implementado o comando *DESC*, que é responsável por exibir toda a estrutura da tabela, mostrando todos os campos e seus respectivos tipos de dados, como também as suas restrições como campo que não pode ter valores nulos, chave primária que impede a duplicação de valores e as chaves estrangeiras que somente aceitam inserir novos dados quando estes existem na tabela relacionada. A estrutura do comando pode ser vista na quadro 11.

“DESC” [Nome da tabela]

Quadro 11 – Estrutura do comando *DESC*.

No protótipo foi implementado o comando *SHOW TABLES*, que é responsável por listar todas as tabelas existentes dentro do banco de dados. A estrutura do comando pode ser vista na quadro 12.

“SHOW TABLES”

Quadro 12 – Estrutura do comando *SHOW TABLES*.

4.3 APRESENTAÇÃO DO SOFTWARE

O protótipo é dividido em dois sistemas distintos. O primeiro é o servidor de banco de dados, cujo objetivo é tratar todas as requisições dos clientes conectados tanto localmente (na mesma máquina onde encontra-se o servidor) quanto dos clientes conectados em diversos pontos da rede. O segundo é o sistema cliente, que permitirá acessar recursos do servidor, através de uma comunicação remota, onde será possível executar comandos SQL para acesso e manipulação de dados no sistema servidor de banco de dados.

Para o sistema cliente ter condições de acessar o servidor é necessário informar o endereço da rede onde esteja executando o sistema servidor, que pode ser local ou remoto. Para descobrir o endereço basta executar o servidor de banco de dados e obter os endereço através da tela principal do sistema servidor. Existem duas possíveis formas para o sistema cliente conectar-se ao servidor de banco de dados, através dos seguintes meios:

a) utilizar o endereço de identificação da rede do microcomputador. Exemplo: 192.168.7.2;
b) utilizar o endereço host da rede do microcomputador. Exemplo: John.

4.3.1 Interface servidor

Ao ser iniciado o programa servidor fica à mostra na barra de *tray* do Windows como mostra a figura 12. O usuário pode exibir a tela principal do servidor com o *mouse* através de dois clique rápidos sobre o ícone. É possível também acessar um menu do servidor através de um clique com o botão direito (configuração padrão do Windows).

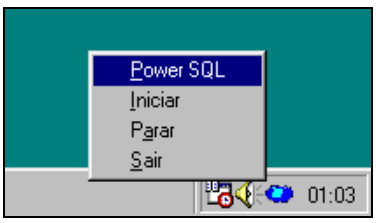


Figura 12 – Opções na barra de tarefa do windows

Pode-se verificar na figura 13 o sistema de banco de dados servidor que contém os endereços necessários para que o sistema cliente possa se conectar ao servidor de forma simples.

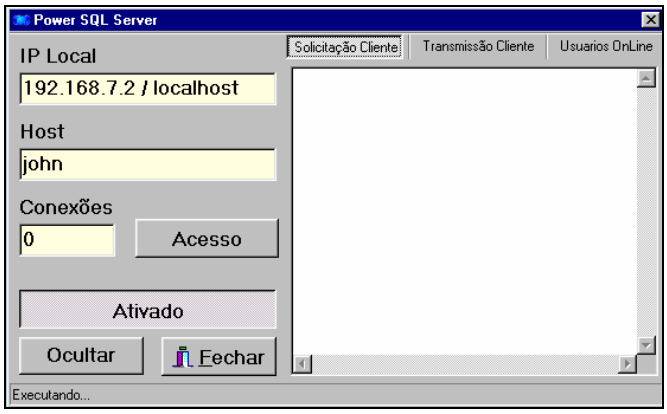


Figura 13 – Tela principal do comandos.

No momento em que o cliente solicita a conexão com o servidor, este registra o código de identificação da aplicação para o banco de dados retornar o resultado dos comandos para o sistema que solicitou. O servidor coleta o *login* do usuário que é autenticado para fornecer as permissões de ações de cada usuário tem dentro do banco de dados. Outros itens que são coletados é o HOST e o IP de origem do sistema do cliente. Conforme o sistema cliente acessa o servidor de banco dados é fornecido uma lista dos usuários conectados e os respectivos bancos de dados que estão acessando, conforme mostra a figura 14.

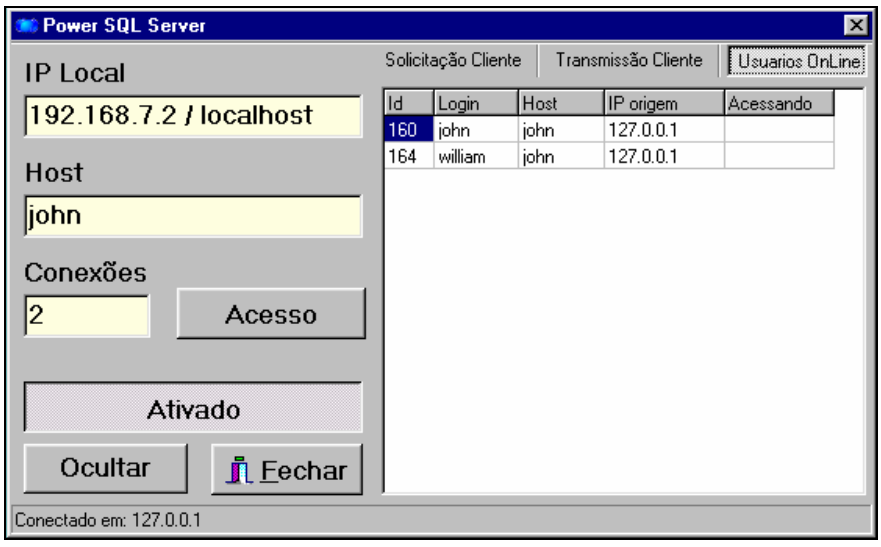


Figura 14 – Tela os clientes conectados.

É permitido ao administrador do banco de dados atribuir contas de usuários, para que estes possam acessar o banco de dados pelo sistema cliente, como mostra a figura 15.

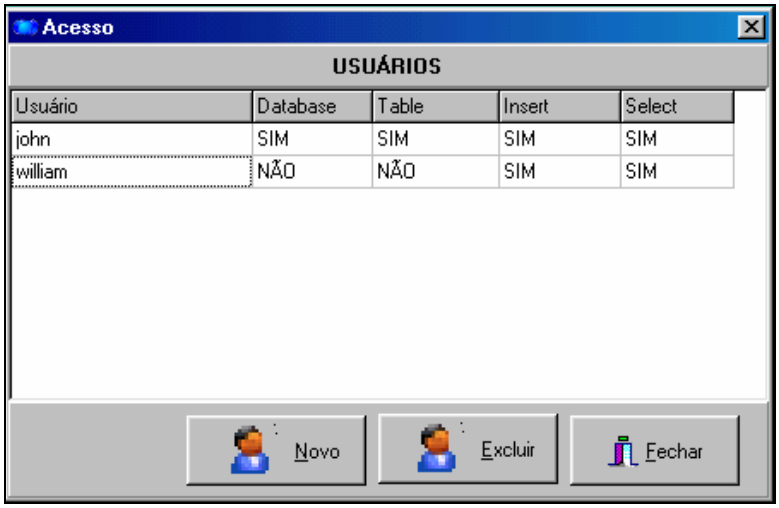


Figura 15 – Tela os clientes cadastrados e sua permissões.

O administrador do banco de dados poderá atribuir os seguintes dados ao usuário como mostra a figura 16:

a) login do usuário;
b) senha de acesso;
c) permissão para criar novos banco de dados;
d) permissão para incluir e excluir tabelas dentro do banco de dados;
e) permissão para incluir dados dentro das tabelas;
f) permissão para consultar dados das tabelas.

Figura 16 – Tela para cadastro e manutenção do usuário.

4.3.2 Interface cliente.

O sistema cliente tem como objetivo acessar os recursos do servidor. Para isto o sistema cliente solicita o *login* de acesso para permitir a autenticação de conexão com o servidor que solicita os seqüentes ítens:

a) endereço onde o servidor está sendo executado. Exemplo: 192.168.7.2;
b) login do usuário;
c) senha do usuário.

Pode-se verificar na figura 17 a janela solicitando os dados para conexão com o servidor de banco de dados.

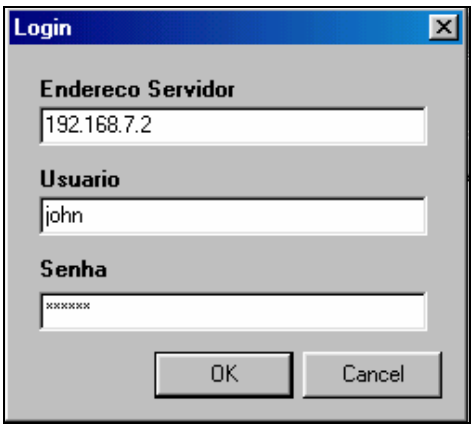


Figura 17 – Tela de login para acesso do cliente.

Após este processo o sistema cliente recebe um código de identificação, para confirmar que o servidor aceitou o acesso e que o sistema cliente tem permissão para executar os comandos SQL no servidor de banco de dados.

No sistema cliente é possível ver todas as tabelas e seus respectivos campos dentro do banco de dados conectado. Tais recursos permitem uma maior facilidade para criar e executar os comandos em SQL, como mostra a figura 18.

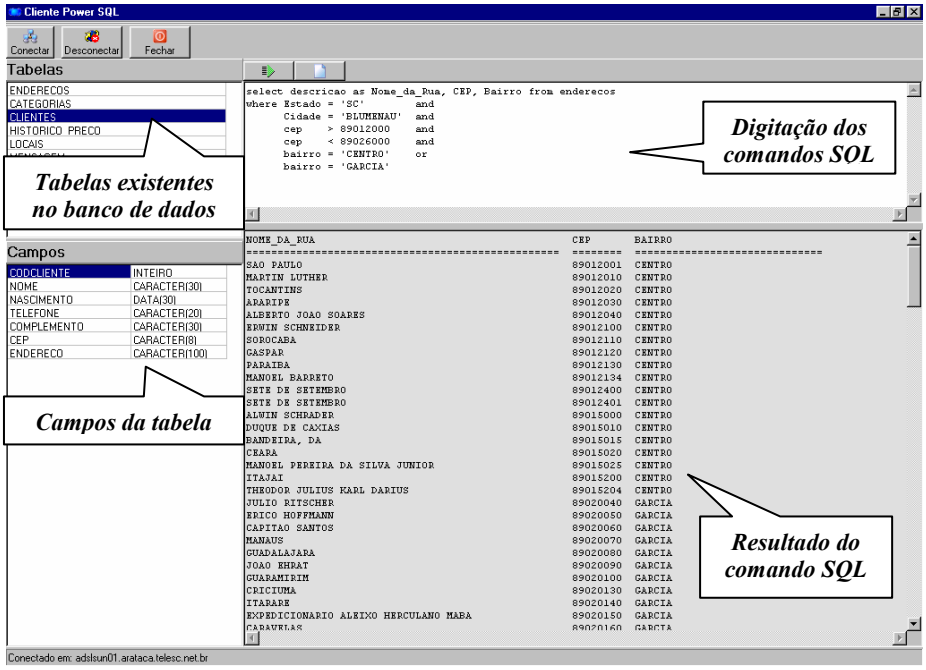


Figura 18 – Ambiente cliente para executar os comandos SQL.

O Sistema cliente possui um menu com todos os comandos possíveis de ser executado dentro do protótipo, este aparece toda vez que é pressionado o botão direto sobre a lista de tabelas, pois quando selecionado algum dos itens da tabela, o sistema já preencherá com nome da tabela selecionada. É possível adicionar nome de tabela e campo com um duplo clique do *mouse*, que inclui na área de edição o respectivo nome, o que permite ao usuário menos preocupação com erros de digitação e agilidade, como mostra a figura 19.

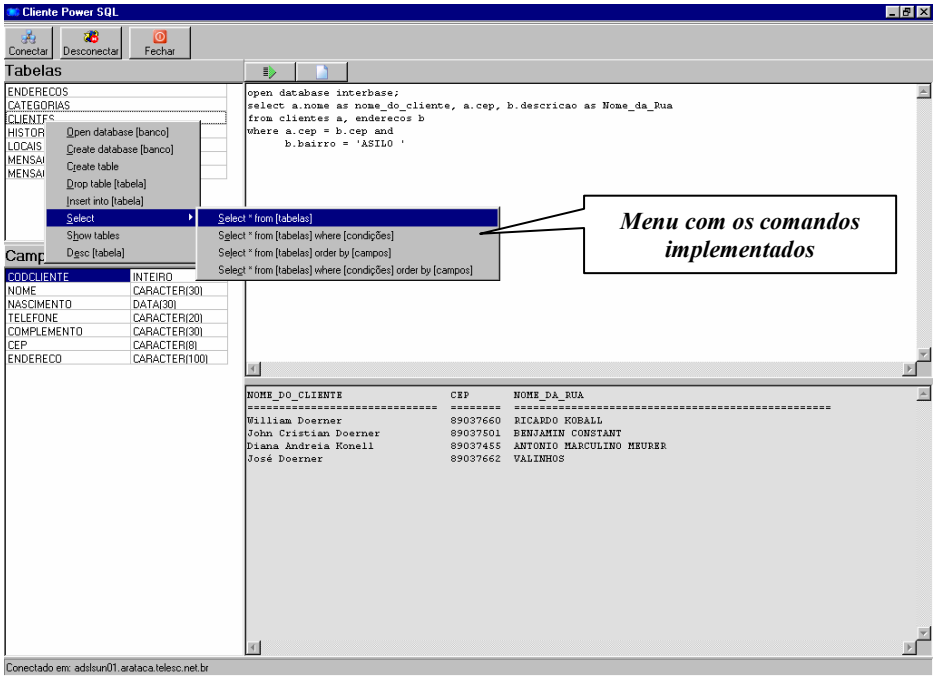


Figura 19 – Ambiente cliente com os comandos desenvolvidos.

4.4 RESULTADOS E DISCUSSÕES

Para iniciar o desenvolvimento do protótipo foram testado vários componentes do ambiente Delphi, o que mais correspondeu às necessidades foi componente *Socket* do Delphi, por apresentar uma maior simplicidade e desempenho para o desenvolvimento.

Na utilização da ferramenta COCOR, foi constatado que a mesma tem limitação de 255 caracteres por *token*, impedindo que o SGBD armazenasse campo com identificadores e dados maiores, como a ferramenta possui o código fonte disponível, foi possível fazer alteração nas bibliotecas tornado capaz de reconhecer *tokens* com tamanho máximo de 1073741824 caracteres.

Nos *tokens* tipo dados, foi necessário fazer validações, para impedir que dados inconsistentes fossem utilizados indevidamente pelo interpretador semântico do SGBD. Para isto foi verificada a estrutura dos números, caracteres no analisador léxico e data e tempo no analisador semântico.

Em testes realizados em uma rede corporativa observou-se uma perda de performance, pois o servidor não possuía condições de enviar os dados num sincronismo constante, uma vez que a rede encontrava variações. Foi necessário tratar o *buffer* recebido no sistema cliente para evitar que ocorresse perda de dados.

O gerador de consulta é o serviço do SGBD que ocupa recurso do sistema, pois muitas de suas consultas são referentes a relacionamento entre duas ou mais tabelas.

No desenvolvimento do protótipo do servidor foi necessário resolver diversos tipo de problemas, um dos maiores problemas foi à implementação de uma estrutura de armazenando eficiente o que permitisse armazenar a estrutura das tabelas e dados num mesmo arquivo. Além de armazenar os dados é necessário tratar estes dados dentro das regras definidas dentro do metadados do banco de dados, como também checar a estrutura dos valores, verificado se são válidos como por exemplo dadas, que podem ter variações com o ano bissexto. Para cada verificação de entrada de dados foi necessário fazer uma análise detalhada de todos os dados relacionados.

O segundo maior desafio foi à implementação do comando *Select*, pois nos livros pesquisados não havia nenhum exemplo de algoritmo para executar este comando. Foi

necessário implementar um algoritmo que permiti-se ao protótipo gerar consultas utilizando várias tabelas, e que permiti-se fazer o relacionamento entre elas através da cláusula *where*. O resultado do algoritmo foi eficiente na geração da consulta, correspondendo dentro do limite do comando *select* implementado.

Para teste de resultado do comando *select* foi utilizado o banco de dados *Access* da Microsoft, onde foi utilizada tabelas com estrutura e dados idênticos entre o *Access* e o protótipo, e foi possível constatar que dentro dos comando implementados o resultado do protótipo foi satisfatório.

O gerenciador de armazenamento desenvolvido no protótipo tem limitação no armazenado dos dados, pois esta limitação vem do ambiente Delphi que permite gerenciar arquivos com no máximo 4 GB.

No protótipo não foi desenvolvido um mecanismo de segurança do dado que transmitido pela rede, pois não foi implementado nenhum tipo algoritmos de criptografia e descriptografia.

O comando *SELECT* possui várias limitações, como:

a) conversão de valores;

b) concatenação de valores;

c) soma, subtração, multiplicação e divisão entre campos numéricos;

d) a execução de *select* dentro de *select*;

e) as funções *COUNT, SUM, ABS*;

f) o comando *GROUP BY*;

g) a condição *HAVING*.

Não foram implementados recursos de *script*, que permite ao banco de dados executar ações dentro do banco de dados quando é executado algum evento dentro do banco de dados.

No protótipo não foram implementados os comandos *UPDATE*, que serve para atualizar dados dentro das tabelas, e o comando *DELETE* que serve para excluir dados dentro da tabela.

Também não foram criados comandos que permitam a modificação das estruturas das tabelas definidas.

5 CONSIDERAÇÕES FINAIS

A seguir descreve-se a conclusão, limitações e as sugestões para trabalhos futuros.

5.1 CONCLUSÃO

Este trabalho permitiu mostrar as barreiras para desenvolver um banco de dados, pois os livros mostram pouco a parte de implementação, mas focam muito nas técnicas utilizadas nos bancos de dados. A maioria dos bancos tem arquitetura fechada e é difícil saber as técnicas que eles utilizam e como realmente funciona a arquitetura do banco de dados.

A base bibliográfica foi o livro de Garcia-Molina, que mostra todos os dispositivos necessários para o funcionamento de um SGBD. Os objetivos propostos foram todos alcançados e foram adicionadas outras funcionalidades como a implementação da álgebra relacional, que permitiu o relacionamento entre várias tabelas.

No protótipo desenvolvido foram realizadas pesquisas para identificar os itens necessários para o desenvolvimento e visualizou-se que existe uma quantidade de detalhes a serem controlados a fim de garantir a consistência das aplicações e dos dados mantidos pelas mesmas.

Com o desenvolvimento do protótipo do banco de dados, foi possível identificar o funcionamento de vários itens de um banco dados, como o aperfeiçoamento de programação, descobrindo os recursos necessários para armazenar os dados e distribuir estes dados em um ambiente cliente/servidor.

5.2 SUGESTÕES PARA TRABALHOS FUTUROS

Algumas sugestões para continuação do protótipo são:

a) implementar uma estrutura de armazenamento utilizado o algoritmo de *hash*, pois o protótipo desenvolvido não possui uma grande capacidade como os bancos relacionais existentes no mercado. Com a implementação deste algoritmo permitiria gerenciar uma capacidade de armazenamento superior a 4 GB;

b) implementação do comando GROUP BY e HAVING com as funções como COUNT, SUM, AVG entre outros, para adicionar mais recursos para tratamento das informações geradas pelo protótipo de banco de dados relacional;

c) Utilizar técnicas de criptografia e descriptografia, pois os dados transmitidos podem ser facilmente identificados por programas como sniffers;

d) criar um *driver* ODBC, para permitir que vários aplicativos que utilizam este recurso acesse o banco de dados. O que permitiria usar o banco de dados em diferentes plataforma de desenvolvimento;

e) implementar recursos de soma, subtração, multiplicação e divisão dentro do comando SELECT, para permitir gerar dados estáticos e geração de resultados que necessitem usar valores;

f) utilizar componentes *socket* mais recentes do Delphi, pois neste protótipo foi utilizado componente do Delphi 5;

g) criar mecanismos eficientes para as *threads* dos gerenciados, pois possibilitará uma maior performace para o servidor, um dos principais objetivos para todos os principais bancos de dados conhecidos.

REFERÊNCIAS BIBLIOGRÁFICAS

DATE, C. J. **Introdução a sistemas de bancos de dados**. Rio de Janeiro: Campus, 2000.

HARRINGTON, JAN L. **Projetos de banco de dados relacional:** teoria e prática. Rio de Janeiro: Campus, 2002.

HÜBNER, Jomi Fred; HUGO, Marcel. BCC-VI: **Uma experiência de implementação de banco de dados relacional**. Blumenau: Dynamis, v. 1, n. 1, p. 59-68, set./out. 1992.

MELO, Rubens N.; SILVA. **Bancos de dados não convencionais** : a tecnologia do BD e suas novas áreas de aplicação. Campinas: Infobook, 1988.

MOLINA, Hector Garcia; ULLMAN, Jeffrey D.; WIDOM, Jennifer. **Implementação de sistema de banco de dados**. Rio de Janeiro: Campus, 2001.

PRINCE, Ana Maria de Alencar; TOSCANI, Simão Sirineo; **Implementação de linguagens de programação:** compiladores. Porto Alegre: Sagra Luzzatto, 2001.

RENAUD, Paul E. **Introdução aos Sistemas Cliente / Servidor:** Guia Prático para Profissionais de Sistemas. Rio de Janeiro: Infobook, 1994.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Sistema de banco de dados.** São Paulo: Makron Books, 1999.

APÊNDICE A – Descrição das classes presentes no gerenciador de armazenamento.

Tusuário: esta classe é responsável por guardar o dados do usuário.

- **Login** : nome do usuário para acesso do usuário ao banco de dados;
- **Senha**: autenticação do usuário para permitir acesso ao banco de dados;
- **Select**: permitir o uso do comando *select*;
- **Create_DB**: permitir a criação de novos bancos de dados;
- **Create_Table**: permitir a criação e exclusão de tabelas;
- **Insert_Into**: permitir a inclusão de novos dados dentro da tabela;
- **Select_Table**: permitir a seleção de dados da tabela.

TControle_Acesso: esta classe é responsável por guardar o dados de todos os usuários conectados dentro do banco de dados.

- **Login**: nome do usuário conectado dentro do banco de dados;
- **Banco**: nome do banco que o usuário esta conectado;
- **IP_Programa**: código de acesso do programa conectado ao banco de dados;
- **Host**: nome da máquina cliente conectado dentro do servidor de banco de dados;
- **IP**: endereço de rede do cliente conectado dentro do servidor de banco de dados.

TGerenciador_Secao: esta classe é responsável em manter atualizados todos os dados de clientes e bancos conectados, o que permite ao servidor identificar o usuário e a que banco ele esta conectado e que permissão ele possui dentro do servidor para poder transmitir de forma íntegra os pedidos solicitados.

- **Usuarios_Arq** : armazenamento físico de todos os usuários do banco de dados;
- **Lista_Usuarios**: lista de todos os usuários cadastrados no banco de dados;
- **Usuarios_Conectados**: lista de todos os usuários conectados e que banco estão acessando;
- **Bancos_Server**: instancia de todos bancos carregados pelo SGBD;
- **Bancos_Cliente**: instancia local do banco de dados;
- **Quant_Banco**: quantidades de instancias carregadas do banco de dados;
- **Quant_Usuarios**: quantidades de usuários cadastrados no banco de dados;
- **Quant_Conectados**: quantidades de usuários conectados no banco de dados;
- **Controle**: identifica o controle do tipo de serviço entre cliente o servidor;

- **Id_Cliente**: identificação do usuário que esta solicitando um recurso do servidor;
- **Create**: cria a instancia de gerenciamento de sessão;
- **Carregar_Usuarios**: carrega todos os usuários cadastrados dentro do banco de dados;
- **Adicionar_Usuarios**: adicionar novo usuário dentro do banco de dados;
- **Atualizar_Usuarios**: atualiza dados dos usuários existentes dentro do banco de dados;
- **Excluir_Usuarios**: excluir usuário do banco de dados;
- **Atualizar_Lista_Usuario**: atualiza lista de usuários cadastrados no banco de dados;
- **Conectar:** conecta o usuário dentro do banco de dados;
- **Desconectar** : desconectar o usuário do banco de dados;
- **Abrir_Banco_dados** : abrir um acesso ao banco de dados não conectado;
- **Fechar_Banco_dados**: fechar acesso ao banco que não tiver sendo utilizado;
- **Existe_Banco_dados_aberto**: informa se existe o banco de dados aberto;
- **Indice_Banco_Dados**: retorna o índice de controle do banco de dados.;
- **Usuario_seleciona_banco**: atribui ao usuário acesso ao banco solicitado;
- **Permicao**: indica a permissão do usuário de criar, excluir e selecionar dados dentro do SGBD;
- **Banco**: indica o banco que o sistema cliente esta acessando.

TProcessador_Consultas: a classe é responsável por coletar todas as informações do comando SELECT, para permitir a geração da consulta usando todos os dados solicitados pelo cliente como campos, tabelas, condições e ordem.

- **Id_Cliente**: identifica o cliente que esta solicitando a consulta;
- **Lista_Campos**: lista de todos os campos solicitados pelo comando *SELECT*;
- **Lista_Tabelas**: lista de todas as tabelas solicitadas pelo comando *FROM*;
- **Lista_Condicao**: lista de todas as condições solicitadas pelo comando *WHERE*;
- **Lista_Ordem**: lista de todos os campos solicitados pelo comando *ORDER BY*;
- **Condição**: retorna se a condição dos campos relacionados é válida;
- **Valor_A**: valor a ser comparado com Valor_B;
- **Valor_B**: valor a ser comparado com Valor_A;
- **Valores_Campos**: valor dos campos selecionados;
- **Lista_Erros**: lista dos erros ocorridos pelo comando *SELECT*;
- **Quant_Campos**: quantidade de campos selecionados;
- **Quant_Tabelas**: quantidade de tabelas selecionadas;
- **Quant_Condicao**: quantidade de condições selecionadas;

- **Quant_Ordem**: quantidade de ordens selecionadas;
- **Quant_Erros**: quantidade de erros existe dentro do comando *SELECT*;
- **Linha_Valores**: linha de comparação dos valores;
- **Quant_Valores**: quantidade de valores resultante do comando *SELECT*;
- **Index_From_Alias**: retorna o índice de controle da tabela através do nome do alias;
- **Index_From_Tabela**: retorna o índice de controle da tabela através do nome da tabela;
- **Index_Tabela_campo**: retorna o índice de controle da tabela através do nome do campo;
- **Index_Campo_Tabela**: retorna o índice de controle do campo através do nome da tabela;
- **Codigo_Campo_Tabela**: retorna o código de identificação do campo através do nome da tabela;
- **Montar_Codicao**: monta a condição para validação para os campos e dados relacionados;
- **QuickSort**: utiliza o método do *QuickSort* para ordena a consulta;
- **Adicionar_Erro**: adiciona erro do comando *SELECT*;
- **Create**: cria o processador de consultas;
- **Adicionar_tabelas**: adiciona a tabela a ser utilizada na geração de consulta;
- **Adicionar_campo**: adiciona o campo a ser gerado a consulta;
- **Adicionar_condicao**: adiciona a condição para tratamento do resultado da consulta;
- **Adicionar_campo_ordem**: adiciona o campo a ser ordenado em forma crescente ou decrescente;
- **Gerar_Consulta**: rotina para gera a consulta com os dados coletados;
- **Dados_Titulo**: retorna o títulos dos campos selecionados;
- **Dados_Barra**: retorna a barra dos títulos selecionados;
- **Dados_Col**: retorna a quantidade de colunas dos dados selecionados;
- **Dados_Lin**: retorna a quantidades de linhas dos dados selecionados;
- **Dados**(Lin , Col): retorna o valores dos dados selecionados;
- **Numero_Erros**: retorna o numero de erros dentro do comando *SELECT*;
- **Erros**: retorna o tipo de erro existente dentro do comando *SELECT*.

TForeignKey : a classe é responsável em manter informação referente ao relacionamento entre as tabelas.

- **Id_ForenKey**: código identificador do *foreing key* dentro da tabela;
- **Nm_forenkey**: nome do *foreing key*;
- **Id_tabela**: código identificador da tabela que possui relacionamento;
- **Id_campo**: código identificador do campo da tabela que possui relacionamento;

- **Id_tab_Refer**: código identificador da tabela referenciada pela tabela principal;
- **Id_tab_campo**: código identificador do campo da tabela referenciada pela tabela principal.

TPrimaryKey: a classe é responsável em manter informação do(s) campo(s) que não poderão ter duplicação de valores.

- **Id_PrimaryKey**: código identificador da primary key dentro da tabela;
- **Nm_primakyKey**: nome da primary Key;
- **Id_campo**: código identificador do campo;
- **Id_tabela**: código identificador da tabela.

Tcampo: a classe é responsável em manter informação da estrutura de todo o campo existente dentro da tabela;

- **Id_campo**: código identificador do campo;
- **Id_Tabela**: código identificador da tabela;
- **Nome** : nome do campo;
- **Tipo**: é o tipo de dados a ser armazenado como inteiro, real, data, caracter e tempo;
- **Tamanho**: trata do tamanho do armazenamento de cada campo;
- **Tamanho_Dec**: trata do tamanho decimal a ser armazenado dentro de cada campo;
- **Não_vazio**: indica se é obrigatório o preenchimento do valor do campo.

TArmazenamento_Dados: a classe é responsável por armazenar os dados extraídos da base de dados.

- **Id_dados**: código identificador do dado armazenado;
- **Id_tabela**: código identificador da tabela;
- **Id_Campo**: código identificado do campo;
- **Dados**: valor armazenado dentro do campo;
- **Dados_Bloco**: valor(es) quebrado(s) que é coletado pelo gerenciador de armazenamento para ser montado.

TGerenciador_Armazenamento: a classe é responsável em manter controle de todas as tabelas e armazenamento de toda a estrutura das tabelas e dados nela armazenados.

- **Nome_Banco**: nome de identificação do banco carregado;
- **Banco_Arquivo**: responsável em acessa o arquivo físico do banco de dados;
- **Banco_Aberto**: indica se o banco de dados está aberto;

- **Quant_Tabela**: indica a quantidade de tabelas abertas;
- **Nomes_Tabelas**: lista com o nome e código identificador de todas as tabelas existentes dentro do banco de dados;
- **Ultimo_Id_Tabela**: identificador do ultimo código identificador utilizado pelas tabelas;
- **Final_Arquivo**: identifica a posição do final do arquivo do arquivo de banco de dados;
- **Estrutura**: Armazena o metadados de todas as tabelas do banco de dados;
- **Create**: cria o controle do gerenciador de armazenamento;
- **Abrir_BancoDados**: responsável por abrir o banco dados;
- **Cria_BancoDados**: responsável por criar um novo banco de dados;
- **Nome_do_Banco**: nome do banco em utilização;
- **Banco_carregado**: indica se o banco esta com metadados carregado;
- **Carregar_Tabelas**: executa a rotina para carregar todos dos dados das tabelas;
- **Cria_Tabela**: responsável por criar uma nova tabela;
- **Deletar_Tabela**: responsável por excluir uma tabela;
- **Valida_Entrada_Dados**: valida os dados a serem inseridos observando às regras das chaves primarias e relacionamento;
- **Inserir_dados_Tabela**: inseri os dados coletados dentro das tabelas;
- **Tabela_existe**: retorna se a tabela existe;
- **Campo_Tabela_existe**: retorna se o campo dentro da tabela existe;
- **Campo_Tabela_Tipo**: retorna o tipo de dados do campo dentro da tabela;
- **Quantidade_Tabelas**: retorna a quantidade de tabelas dentro do banco de dados;
- **Quantidade_Campos_Tabela**: retorna a quantidade de campos dentro da tabela;
- **Quantidade_Primary_Tabela** : retorna a quantidade de *primary key* dentro da tabela;
- **Quantidade_Foreign_Tabela**: retorna a quantidade de *foreign key* dentro da tabela;
- **Quantidade_Dados_Tabela**: retorna a quantidade de dados dentro da tabela;
- **Nome_Tabela**: retorna o nome da tabela selecionada;
- **Estrutura_Nome_Tabela**: retorna o metadados da tabela pelo nome da tabela;
- **Estrutura_Campo_Tabela**: retorna a estrutura do campo dentro da tabela;
- **Estrutura_Ordem_Campo_Tabela**: retorna a estrutura do campo pela ordem cadastrada;
- **Estrutura_Ordem_Primary**: retorna a estrutura do *primary key* pela ordem cadastrada;
- **Estrutura_Ordem_Foreign**: retorna a estrutura do *foreign key* pela ordem cadastrada;
- **Tabela_Dados**: retorna os dados existe dentro da tabela selecionada;
- **Codigo_Tabela**: retorna o código de controle da tabela selecionada;

- **Codigo_Campo_Tabela**: retorna o código de controle do campo selecionado da tabela.

TGerencia_Tabelas: a classe é responsável em manter a estrutura e os dados armazenados.

- **Id_Tabela**: código de identificação da tabela;
- **Estrutura**: contém a estrutura de todos os campos da tabela;
- **PrimaryKey**: contém a estrutura de todos os *primary key* da tabela;
- **ForeignKey**: contém a estrutura de todos os *foreign key* da tabela;
- **Armazenamento**: contém todos os dados armazenados dentro da tabela;
- **Quant_Campos**: quantidade de campos existentes na tabela;
- **Quant_Primary**: quantidade de *primary key* existente na tabela;
- **Quant_Foren**: quantidade de foreign key existente na tabela;
- **Quant_dados**: quantidade de dados existente na tabela;
- **Ultimo_id_Campo**: código de controle do ultimo campo utilizado;
- **Ultimo_id_PrimaryKey**: código de controle do ultimo *primary key* utilizado;
- **Ultimo_id_ForeignKey**: código de controle do ultimo *foreign key* utilizado;
- **Ultimo_id_dados**: código de controle do ultimo dados utilizado;
- **Adicionar_Campo**: adiciona campo dentro da estrutura da tabela;
- **Adicionar_PrimaryKey**: adiciona *primary key* dentro da estrutura da tabela;
- **Adicionar_ForeignKey**: adiciona *foreign key* dentro da estrutura da tabela;
- **Adicionar_Dados**: adiciona dados dentro da estrutura da tabela;
- **Indice_Campo**: retorna o controle de índice do campo;
- **Indice_Dados**: retorna o controle de índice do dado;
- **Existe_Campo**: retorna se existe o campo dentro da tabela;
- **Tipo_campo**: retorna o tipo de campo selecionado;
- **Estrutura_Campo**: retorna a estrutura do campo selecionado;
- **Codigo_Campo**: retorna o código de identificação do campo selecionado;
- **Quantidade_Campos**: retorna a quantidade campos existentes na tabela;
- **Quantidade_Primary_Key**: retorna a quantidade *primary key* existentes na tabela;
- **Quantidade_Foreign_Key**: retorna a quantidade *foreign key* existentes na tabela;
- **Quantidade_Dados**: retorna a quantidade dados existentes na tabela;
- **Dados**: retorna os dados armazenados na tabela.

TDados: a classe é responsável por remontar os dados armazenados em disco.

- **Id_Campo**: código de identificação do campo;
- **Id_Tabela**: código de identificação da tabela;

- **Id_dados**: código de identificação do dado;
- **Quant_Bloco**: quantidade de blocos gastos para armazenar o valor;
- **Id_bloco**: código de identificação do bloco armazenado;
- **Value**: valor do bloco armazenado.

TEstrutura_Armazenamento: a classe é responsável por identificar o tipo de estrutura a ser armazenado em disco.

- **TipoEstrutura**: identificador do tipo de estrutura a ser armazenado em disco;
- **Livre**: indicador se o espaço armazenado esta livre para ser utilizado por outro dado.

APÊNDICE B – Descrição das classes presentes no compilador de SQL.

TCompilador_Consulta: a classe é responsável por coletar os *tokens*, analisar e executar os comandos recebidos.

- **Nome_Constraint**: armazena o nome da *constraint* coletada pelo compilador;
- **Where_D_ou_E**: indica o lado que esta sendo coletado o valor de comparação;
- **Id_Cliente**: código de identificação do cliente;
- **Select_Tabela**: armazena todos os dados coletados pelo comando *SELECT*;
- **Create_Tabela**: armazena todos os dados coletados pelo comando *CREATE TABLE*;
- **Inserir_Tabela**: armazena todos os dados coletados pelo comando *INSERT INTO*;
- **Existe_Campo**: retorna de o campo já existe;
- **Cria_DataBase**: executa a rotina de criar novo banco de dados;
- **Abrir_DataBase**: executa a rotina para abrir um banco de dados;
- **Coleta_Nome_constraint**: coleta o nome da *constraint;*
- **Cria_Tabela**: inicia o valor do comando *CREATE TABLE*;
- **Cria_tabela_Novo_Campo**: indica que existe um novo campo;
- **Cria_tabela_Campo**: coleta o nome do campo;
- **Cria_tabela_Tipo**: coleta o tipo de dados do campo;
- **Cria_tabela_Tamanho**: coleta o tamanho de armazenamento do campo;
- **Cria_tabela_Tamanho_Dec**: coleta o tamanho decimal do campo;
- **Cria_tabela_NaoVazio**: coleta se o campo é do tipo não vazio;
- **Cria_tabela_Novo_Primary**: indica que existe um novo *primary key*;
- **Cria_tabela_Novo_Primary_Item**: indica que existe mais um campo para a mesma *primary key*;
- **Cria_tabela_Primary_Local**: indica que a *primary key* foi declarada depois do tipo do campo;
- **Cria_tabela_Primary_Campo**: coleta o nome do campo da *primary key;*
- **Cria_tabela_Novo_Foreign**: indica que existe um novo *foreign key*;
- **Cria_tabela_Foreign_campo**: coleta o nome do campo local;
- **Cria_tabela_Foreign_Tabela**: coleta o nome da tabela de referenciada;
- **Cria_tabela_Foreign_Tabela_campo**: coleta o nome do campo de referenciada;
- **Cria_tabela_Executar**: executa a rotina para criação da tabela com os itens coletados;
- **Deletar_Tabela_Nome**: coleta o nome da tabela a ser excluída;

- **Deletar_Tabela_Execute**: executa a rotina para exclusão da tabela;
- **Insert_Tabela**: inicia o valor do comando *INSERT INTO*;
- **Insert_Tabela_Novo_Campo**: indica que existe um novo campo;
- **Insert_Tabela_Campo** coleta o nome do campo;
- **Insert_Tabela_Novo_Value**: coleta o valor do campo;
- **Insert_Tabela_Value**coleta o tipo de valor a ser inserido no campo;
- **Insert_Tabela_Executar**: executa a rotina para inserir o dados dentro da tabela;
- **Seleciona_Tabela**: inicia o valor do comando *SELECT*;
- **Seleciona_Tabela_Novo_Campo**: indica que existe um novo campo a ser selecionado;
- **Seleciona_Tabela_Todos_Campo** indica que todos os campos serão selecionados;
- **Seleciona_Tabela_Campo_Tabela**: coleta o nome da tabela que o campo pertence;
- **Seleciona_Tabela_Campo_Nome**: coleta o nome do campo;
- **Seleciona_Tabela_Campo_Alias**: coleta o nome do alias que o campo pertence;
- **Seleciona_Tabela_Novo_From**: indica que existe uma nova tabela selecionada;
- **Seleciona_Tabela_From_Tabela**: coleta o nome da tabela selecionada;
- **Seleciona_Tabela_From_Alias**: coleta o nome do alias da tabela;
- **Seleciona_Tabela_Novo_Where**: indica que existe uma nova condição;
- **Seleciona_Tabela_Where_Tabela**: coleta o nome da tabela. da condição;
- **Seleciona_Tabela_Where_Campo**: coleta o nome do campo da condição;
- **Seleciona_Tabela_Where_Contante**: coleta o valor estático da condição;
- **Seleciona_Tabela_Where_Condicao**: coleta o tipo de condição a ser executada;
- **Seleciona_Tabela_Where_Relacionamento**: indica o tipo de relacionamento haverá com cada condição ("*and*" ou "*or*");
- **Seleciona_Tabela_Novo_OrderBy**: indica que existe um novo campo a ser ordenado;
- **Seleciona_Tabela_OrderBy_Tabela**: coleta o nome da tabela;
- **Seleciona_Tabela_OrderBy_Campo**: coleta o nome do campo a ser ordenado;
- **Seleciona_Tabela_OrderBy_Ordem**: indica o tipo de ordem a usado ("crescente" ou "decrescente");
- **Seleciona_tabela_Executar**: executa a rotina de seleção de dados da tabela.

TCamposTabela: a classe é responsável por manter os dados dos campos coletados pelo comando *CREATE TABLE*.

- **Campo**: armazena o nome do campo coletado;
- **Tipo**: coleta o tipo de dados do campo;

- **Tamanho**: coleta o tamanho do valor a ser armazenado pelo campo;
- **Tamanho_dec**: coleta o tamanho decimal do valor a ser armazenado;
- **Nao_Vazio**: colete se o tipo de campo é não vazio.

TPrimaryTabela: a classe é responsável por manter os dados das *primary key* existentes dentro comando *CREATE TABLE*.

- **Nome**: coleta o nome do identificador da *primary key*;
- **Campo**: coleta o nome do campo.

TForenTabela: a classe é responsável por manter os dados das *foreign key* existentes dentro comando *CREATE TABLE*.

- **Nome**: coleta o nome do identificador do *foreign key*;
- **Campo**: coleta o nome do campo;
- **Tabela**: coleta o nome da tabela a ser relacionado;
- **Tab_Campo**: coleta o nome do campo a ser relacionado.

TCreate_Tabela: a classe é responsável por manter a estrutura existente dentro comando *CREATE TABLE*.

- **Tabela**: coleta o nome da tabela a ser criada;
- **Campos**: armazena todos os campos declarados;
- **PrimaryKey**: armazena todos os *primary key* declarados;
- **ForenKey**: armazena todos os *foreign key* declarados;
- **Ncampo**: indica a quantidade de campos existentes;
- **NPrimary**: indica a quantidade de *primary key* existentes;
- **NForen**: indica a quantidade de *foreign key* existentes.

TCamposInto: a classe é responsável por manter os dados dos campos a serem inseridos pelo comando *INSERT INTO*.

- **Id_Campo**: código de identificação do campo a ser inserido;

TCamposValue: a classe é responsável por manter o valor dos dados dos campos a serem inseridos pelo comando *INSERT INTO*.

- **Value**: valor a ser armazenado pelo campo.

TInsert_Into: a classe é responsável por manter a estrutura existente dentro comando *INSERT INTO*.

- **Id_Tabela**: código de identificação da tabela a ter dados inseridos;
- **Nome_Tabela**: nome da tabela a ter dados inseridos;
- **Campos**: armazena todos os campos declarados;
- **Values**: armazena todos os valores declarados;
- **NCampo**: indica a quantidade de campos declarados;
- **NValue**: indica a quantidade de valores declarados.

Tidentificao: a classe é responsável por manter os dados dos campos selecionados pelo comando *SELECT*.

- **Tabela**: coleta o nome da tabela de referencia do campo;
- **Campo**: coleta o nome do campo a ser selecionado;
- **Alias**: coleta o nome do alias do campo.

TTabelas: a classe é responsável por manter os dados das tabelas selecionadas pelo comando *SELECT*.

- **Tabela**: coleta o nome da tabela a ser selecionado;
- **Alias**: coleta o nome do alias da tabela.

Tcondicao: a classe é responsável por manter os dados das condições selecionadas pelo comando *SELECT*.

- **TabelaA**: coleta o nome da tabela a ser selecionado;
- **CampoA**: coleta o nome do campo;
- **ConstantA** : coleta o valor estático do campo;
- **Condicao** : indica o tipo de condição entre os campo (“=”, “<>”, “>”, “<”,”>=”, “<=”);
- **TabelaB**: coleta o nome da tabela a ser selecionado;
- **CampoB**: coleta o nome do campo;
- **ConstantB** : coleta o valor estático do campo;
- **Relacao** : indica o tipo de relação com as outras condições (“and”, “or”).

TcondicaoOrdem: a classe é responsável por manter os dados das ordens dos campos selecionadas pelo comando *SELECT*.

- **Tabela**: coleta o nome da tabela selecionada;
- **Campo** : coleta o nome do campo a ser ordenado;
- **Ordem** : indica o tipo de ordem que o campo sofrerá (“crescente”, “decrecrescente”).

TSeleciona_tabela: a classe é responsável por manter a estrutura existente dentro comando *SELECT*.

- **SelectAll**: indica se todas tabelas serão selecionadas;
- **Select** : armazena todos os campos declarados;
- **From** : armazena todas as tabelas declaradas;
- **Orderby**: armazena todos os campos declarados;
- **NSelect** : indica a quantidade de campos existente;
- **NFrom**: indica a quantidade de tabelas existente;
- **NWhere**: indica a quantidade de condições existentes;
- **NOrderBy**: indica a quantidade de campos a serem ordenados.

APÊNDICE C – Estrutura da BNF utilizada no protótipo.

```
PowerSQL        =
                  ((OpenBaseSQL
                  | SelectSQL
                  | UpdateSQL
                  | InsertSQL
                  | DeleteSQL
                  | Transaction
                  | CreateExpr
                  | Drop
                  | AlterTable
                  | ShowObjeto
                  | DescObjeto
                  )
                   [";" [PowerSQL]
                   ]
                  )

                .
OpenBaseSQL     = "OPEN"
                  SubOpenBaseSQL
                .

SubOpenBaseSQL  = "DATABASE"
                  DataBase
                .

SelectSQL       = SelectStmt
                .

SubSelectSQL    = SelectSQL                                    .

UpdateSQL       = UpdateStmt .

InsertSQL       = InsertStmt .

DeleteSQL       = DeleteStmt .

UpdateStmt      = "UPDATE"
                   Table
                  "SET"
                  UpdateFieldList
                  [ WhereClause ] .

UpdateField     = ColumnName
                  "="
                  Expression .

UpdateFieldList = UpdateField
                  { ItemSeparator
                  UpdateField } .

InsertStmt      = "INSERT"
                  "INTO"
                  Table
                  [
                  OpenParens
                    InsertList
                  CloseParens
```

```
                  ]
                  ((
                  "VALUES"
                  OpenParens
                    ValueList
                  CloseParens
                  )| SelectSQL)
                  .

InsertList      = InsertName
                  { ItemSeparator
                  InsertName} .

InsertName      = ident
                  .

ValueList       = ValueField
                  { ItemSeparator
                  ValueField} .

ValueField      = Null
                  | float
                  | integer_
                  | date_
                  | time_
                  | SQLString
                  .

DeleteStmt      = "DELETE"
                  "FROM"
                  Table
                  [WhereClause]
                  .

SelectStmt      =
                  (SelectClause
                  FromClause
                  [WhereClause]
                  [GroupByClause]
                  [HavingClause]
                  [OrderByClause])
                  .

SelectClause    = "SELECT"
                  SelectFieldList
                  .

SelectFieldList = SelectField
                  { ItemSeparator
                  SelectField } .

SelectField     = (
                  FieldExpression
                  ["AS"
                  Alias
                  ])
                  | "*"
                  .

FieldExpression = (
                   FieldType
```

```
                    | ColumnFunction
                    | FunctionExpr
                    | (OpenParens
                          SubSelectSQL
                       CloseParens)
                    )
                    .

FieldType       =  ident
                   ( ["*"
                     | "."
                      ident
                     ]
                   )
                   .


ColumnFunction = ("COUNT"
                   | "SUM"
                   | "MAX"
                   | "MIN"
                   | "AVG")
                   OpenParens
                     ("*"
                     | (["DISTINCT"
                     ]
                     Expression))
                   CloseParens
                   .

FunctionExpr    = ( "TIMESTAMP"
                   | "UPPER"
                   | "MONTH"
                   | "YEAR"
                   )
                   OpenParens
                     FieldExpression
                     { ItemSeparator
                     FieldExpression }
                   CloseParens
                   .

FromClause      = SYNC "FROM"
                   FromTableList .

FromTableList   = QualifiedTable
                   {
                     ( ItemSeparator
                     QualifiedTable )
                   | JoinStmt
                   } .

QualifiedTable  = ident
                   [ Alias
                   ]
                   .

JoinStmt        =
                   (CrossJoin
                   | ([ JoinType ]
                   "JOIN"
```

```
                  QualifiedTable
                  [
                   JoinExpr ])) .
CrossJoin       = "CROSS"
                  "JOIN"
                  QualifiedTable .

JoinType        = ["NATURAL"
                  ]
                  ("INNER"
                  |(("FULL"
                  | "LEFT"
                  | "RIGHT")
                   [ "OUTER"
                    ])
                  )
                   .
JoinExpr        = ("ON"
                  Expression)
                  | ("USING"
                  OpenParens
                    ColumnList
                  CloseParens
                  ) .

WhereClause     = SYNC "WHERE"
                  WhereCondition
                  .

WhereCondition =   (
                    WhereExpression
                    {Relation
                    WhereExpression}
                    )
                    [(WhereOperator)
                     WhereCondition
                    ]
                  .

WhereExpression   = [ NotOperator ]
                     WhereTerm
                  .

WhereOperator    = ("AND"
                   |"OR"
                   )
                  .

WhereTerm       = (  ident
                       ["."
                        ident
                       ]
                     | integer_
                     | float
                     | date_
                     | time_
                     | SQLString
                     | (OpenParens
                          (WhereTerm
                           |
                           SubSelectSQL
```

```
                        )
                     CloseParens
                    )
                )
                .

HavingClause    = SYNC "HAVING"
                  HavingCondition .


HavingCondition =   (
                    HavingExpression
                    {Relation
                    HavingExpression}
                    )
                .


HavingExpression   = [ NotOperator ]
                     HavingTerm
                     {HavingOperator
                     [ NotOperator]

                     HavingTerm}
                .

HavingOperator    = ("AND"
                  | "OR"
                  )
                  .

HavingTerm       = (  ident
                      ["."
                       ident
                      ]
                    | integer_
                    | float
                    | date_
                    | time_
                    | SQLString
                    | (OpenParens
                         (HavingTerm
                          |
                          SubSelectSQL
                          )
                       CloseParens
                      )
                )
                .

OrderByClause   = SYNC "ORDER"

                  "BY"
                  OrderByFldList .

OrderByFldList  = OrderByField
                  { ( ItemSeparator
                  OrderByField ) } .


OrderByField    = (OrderByColumnName
```

```
                    )
                    [ ("DESC"
                       | "ASC"
                      )
                    ] .

OrderByColumnName = ident
                     [ "."
                       (ident
                        )
                     ] .


GroupByClause    = SYNC "GROUP"
                    "BY"
                    GroupFieldList .

GroupFieldList   = GroupColumnName
                    { ItemSeparator
                    GroupColumnName}
                    .

GroupColumnName = ident
                    [
                     "."
                     ident
                    ] .


ColumnList   =      ColumnName
                    { ItemSeparator
                    ColumnName} .

ColumnName       = ident
                    [
                    "."
                    (ident
                    | "*"
                    )] .

SimpleColumnName= ident
                    .

SimpleColumnList=  SimpleColumnName
                    { ItemSeparator
                     SimpleColumnName
                    } .


SimpleColumnParam= OpenParens
                      SimpleColumnList
                    CloseParens
                     .

FieldList        = Field
                    { ItemSeparator
                    Field}
                    .

Field            = ColumnName
                    | Null
```

```
                  | float
                  | integer_
                  | SQLString
                  | Param
                  .

Null              = "NULL"
                  .
DataBase          = ident
                  .
Table             = ident
                  .
Alias             = ident
                  .


Expression        = SimpleExpression
                  { Relation
                  SimpleExpression } .

SimpleExpression= [ NotOperator ]
                  Term
                  { Operator
                  [ NotOperator ]
                  Term } .

Term              = (
                    (Field
                      [ TestExpr ]
                      )
                  | ColumnFunction
                  | FunctionExpr
                  | (OpenParens
                      (Expression
                       |
                       SubSelectSQL
                      )
                    CloseParens
                    )
                  ) .


Param             = ":"
                     ident                                          .

NotOperator       = "NOT"
                  .

Operator          = MathOperator | WordOperator .

MathOperator      = ( "*"
                  | "/"
                  | "+"
                  | "-"
                  )
                  .

WordOperator      = ("AND"
                  | "OR"
                  )
                  .
```

```
LikeTest        = "LIKE"
                  (SQLString
                   | Param
                   )
                  [ "ESCAPE"
                  SQLString
                  ]
                  .

NullTest        = "IS"
                  [NotOperator]
                  Null
                  .

Relation        = ("="
                  | "<>"
                  | "<"
                  | "<="
                  | ">"
                  | ">="
                  )
                  .

TestExpr        = NullTest
                  | [NotOperator]
                  ( InSetExpr
                   | BetweenExpr
                   | LikeTest ).

BetweenExpr     = "BETWEEN"
                  Field
                  "AND"
                  Field
                  .

InSetExpr       = "IN"
                  OpenParens
                    (FieldList
                    | SubSelectSQL)
                  CloseParens
                  .
Transaction     = ('COMMIT' | 'ROLLBACK')
                  ['WORK'
                  ] .

len             = integer_
                  .

lenParam        = OpenParens
                    len
                  CloseParens
                  .

numPrecision    = OpenParens
                  ( integer_
                    [ ItemSeparator
                      integer_
                    ]
                  )
                  CloseParens
```

```
                .

DataType        = (("CHAR" | "CHARACTER")
                    lenParam
                    )
                  | ("VARCHAR"
                    lenParam
                    )
                  | ("VARCHAR2"
                    lenParam
                    )
                  | ("INTEGER" | "INT")
                    [ numPrecision ]
                  | "SMALLINT"
                    [ numPrecision ]
                  | ("NUMERIC"
                     [ numPrecision ]
                    )
                  | ("NUMBER"
                    [ numPrecision ]
                    )

                  | "DATE"
                  | ("TIME"
                    )
                  | ("TIMESTAMP"
                    )
                  .

ColumnDefault   =  "DEFAULT"
                     (SQLString
                     | integer_
                     | float
                     ).

ColumnDef       = SimpleColumnName
                  DataType
                  [ [ Constraint ]
                    ( ColPrimaryKey
                      | ColForeignKey ) ]
                  { ColumnDefault
                    |
                   (NotOperator
                    Null
                    )
                  }
                  .

SimplePrimaryName= ident
                  .

SimplePrimaryList=  SimplePrimaryName
                   { ItemSeparator
                    SimplePrimaryName
                   } .


SimplePrimaryParam= OpenParens
                     SimplePrimaryList
                    CloseParens
```

```
                    .

PrimaryKey      = "PRIMARY"
                  "KEY"
                  SimplePrimaryParam
                  .

ColPrimaryKey   = "PRIMARY"
                  "KEY"
                  .

RelationName    = ident
                  .

ConstraintName  = ident
                  .

SimpleForenName = ident
                  .

SimpleForenParam= OpenParens
                     SimpleForenName
                  CloseParens
                  .

SimpleReferName = ident
                  .

SimpleReferParam= OpenParens
                     SimpleReferName
                  CloseParens
                  .


ForeignKey      = "FOREIGN"
                  "KEY"
                  SimpleForenParam
                  "REFERENCES"
                  Table
                  SimpleReferParam
                  ["MATCH"
                    ("FULL" | "PARTIAL")
                    ]
                  {"ON"
                     ("DELETE"
                       | "UPDATE")
                     ("CASCADE"
                       | ("SET"
                       (Null
                         | "DEFAULT")
                       ))
                     | ("NO"
                        "ACTION"
                        )
                     }
                  .

ColForeignKey   = "REFERENCES"
                  Table
                  ( OpenParens
```

```
                    SimpleColumnName
                  CloseParens )
                .

Unique          = "UNIQUE"
                  SimpleColumnParam
                .

CheckConstraint = "CHECK"
                  OpenParens
                    Expression
                  CloseParens
                .

CreatePart      = ColumnDef
                       | PrimaryKey
                       | ForeignKey
                       | ConstraintKey
                       | Unique
                       | CheckConstraint
                .

CreateExpr      = "CREATE"
                  ( CreateDataBase
                    | CreateTable
                    | CreateIndex
                  )
                  .

CreateDataBase  = "DATABASE"
                   DataBase
                  .

CreateTable     = "TABLE"
                  Table
                  OpenParens
                    CreatePart
                    { ItemSeparator
                    CreatePart
                    }
                  CloseParens
                  .

ConstraintKey    = "CONSTRAINT"
                   ConstraintName
                   ( PrimaryKey
                     | ForeignKey
                   )
                   .

Constraint     = "CONSTRAINT"
                  ConstraintName
                  .

CascadeRestrict= ("CASCADE" | "RESTRICT")
                 .

Drop           = "DROP"
                 (DropTable
                  | IndexAndName)
                 .
```

```
DropTable      = "TABLE"
                 ident
                 [CascadeRestrict]
                 .

Add            = "ADD"
                  ( SimpleColumnName
                   DataType
                   [ [ Constraint ]
                     ( ColPrimaryKey
                       | ColForeignKey ) ] )
                   | Constraint
                     ( PrimaryKey
                       | ForeignKey
                       | Unique
                       | CheckConstraint
                      )
                  .

Alter          = "ALTER"
                 SimpleColumnName
                 (("DROP"
                  "DEFAULT"
                  )
                  | ("SET"
                     ColumnDefault
                    )
                 )
                 .

DropPart       = "DROP"
                 (
                 (SimpleColumnName
                 CascadeRestrict)
                 | ("PRIMARY"
                    "KEY"
                    )
                 | ("FOREIGN"
                    "KEY"
                    RelationName
                   )
                 | ("CONSTRAINT"
                    ConstraintName
                    CascadeRestrict
                   )
                 )
                 .

AlterTable     = "ALTER"
                 "TABLE"
                 QualifiedTable
                 (Add
                 | Alter
                 | DropPart
                 )
                 .

IndexColumn    = SimpleColumnName
                 [("ASC" | "DESC")
                 ]
```

```
                  .

  IndexColumnList= IndexColumn
                   { ItemSeparator
                     IndexColumn
                   }
                   .

  IndexName      = ident
                   .

  CreateIndex    = ["UNIQUE"
                   ]
                   IndexAndName
                   "ON"
                   Table
                   OpenParens
                     IndexColumnList
                   CloseParens
                   .

  IndexAndName   = "INDEX"
                   IndexName
                   .

  ItemSeparator  = WEAK ","
                   .

  OpenParens     = "("
                   .

  CloseParens    = WEAK ")"
                   .

  ShowObjeto     = "SHOW"
                   "TABLES"
                   .

  DescObjeto     = ("DESCRIBE" | "DESC")
                   Table
                   .

END PowerSQL.
```