FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO "EURÍPIDES DE MARÍLIA" – UNIVEM
PROGRAMA DE MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

WENDEL BRUSTOLIN DA SILVA

TÉCNICAS DE RECUPERAÇÃO E CONTROLE DE TRANSAÇÃO UTILIZANDO BANCO DE DADOS STANDBY

MARÍLIA / SP
2006

WENDEL BRUSTOLIN DA SILVA

TÉCNICAS DE RECUPERAÇÃO E CONTROLE DE TRANSAÇÃO UTILIZANDO BANCO DE DADOS STANDBY

Dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, para obtenção do Título de Mestre em Ciência da Computação.
(Área de Concentração: Engenharia de Software).

Orientador:

Prof. Dr. Edmundo Sérgio Spoto

MARÍLIA / SP
2006

WENDEL BRUSTOLIN DA SILVA

TÉCNICAS DE RECUPERAÇÃO E CONTROLE DE TRANSAÇÃO UTILIZANDO BANCO DE DADOS STANDBY

Banca examinadora da dissertação apresentada ao Programa de Mestrado da UNIVEM/F.E.E.S.R., para obtenção do Título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software.

Resultado: _____

ORIENTADOR: Prof. Dr. _____

1º EXAMINADOR: _____

2º EXAMINADOR: _____

Marília, ____ de _____ de 2006.

Dedico esse trabalho à minha esposa, ao meu filho, aos meus pais, aos meus familiares e aos meus amigos, sem os quais essa caminhada jamais teria sido concluída

AGRADECIMENTOS

Agradeço ao meu orientador Prof. Dr. Edmundo Sérgio Spoto, pela orientação segura e a confiança que me passou ao longo da elaboração dessa tese.

À minha esposa e ao meu filho, pelo amor, carinho, compreensão e dedicação que têm me dado nesse período.

À Usina de Açúcar Santa Terezinha, que colaborou para que eu pudesse vencer mais essa etapa da minha vida.

Aos amigos e companheiros, que direta ou indiretamente, me incentivaram nessa minha caminhada.

SILVA, Wendel Brustolin da. **Técnicas de Recuperação e Controle de Transação Utilizando Banco de Dados Standby.** 2006. 104 f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares Rocha, Marília, 2006.

RESUMO

Com o crescimento empresarial e institucional, o Banco de Dados é uma ferramenta que não pode ficar fora do ar por motivo de falhas. Com o objetivo de minimizar esse tempo é proposta uma técnica de recuperação de transação utilizando Banco de Dados *Standby*, onde as transações executadas no Banco de Dados de Produção são enviadas para um Servidor de *Backup* que aplica esses *logs* e mantém uma cópia atualizada do Banco de Dados de Produção. Após a identificação da falha, esse *backup* se torna operante em um curto espaço de tempo. A técnica proposta nesta dissertação para que haja a recuperação de transação de um Banco de Dados Principal fazendo uso de um Banco de Dados *Standby* é a ferramenta SABaDO, que foi desenvolvida em Java com o intuito de atender a diversos requisitos de *backup*, como: *backup* lógico do Banco de Dados, *backup full* do Banco de Dados, *backup* e aplicação dos arquivos de transações.

Palavras-Chave: Backup. Recovery. Transação, BD Standby, Ferramenta SABaDO.

SILVA, Wendel Brustolin da. **Técnicas de Recuperação e Controle de Transação Utilizando Banco de Dados Standby.** 2006. 104 f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares Rocha, Marília, 2006.

ABSTRACT

Due to the business and institutional development, Database is a tool which cannot be out the air for failure reason. With the objective of minimizing this time, it is proposed a technique of transaction recovery using Standby Database where the performed transactions in the Production Database are sent to a Backup Server that applies these logs and keep an updated copy of Production Database. After a failure identification, this backup becomes operative at a short time term. The technique proposal in this dissertation so that it has the recovery of transaction of a Main Database making use of a Standby Database is the SABaDO tool, that was developed in Java with intention to take care of the diverse requirements of backup, as: logical backup logical of Database, full backup of Database, backup and application them archives of transactions

Keywords: Backup. Recovery. Transaction.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS

API – Applications Programming Interface, Aplicações de Programas de Interface.
BD - Banco de Dados.
BF – Backup Full.
DCL – Data Control Language, Linguagem de Controle de Dados.
DDL – Data Definition Languagem, Linguagem de Definição de Dados.
DEPT – Departamento.
DEPTO – Departamento.
DML – Data Manipulation Language, Linguagem de Manipulação de Dados.
E-R – Entidade – Relacionamento.
FK – Foreign Key, Chave Estrangeira.
FTP – File Transfer Protocol, Protocolo de Transferência de Arquivo.
FUNC – Funcionários.
GB – Gigabyte.
HBA – Host – based authentication, Autenticação baseada no hospedeiro.
JVM – Java Virtual Machine.
MB – Megabytes.
MER – Modelo de Entidade e Relacionamento.
NM – Nome.
NM_FUNC – Nome do Funcionário.
PK – Primary Key, Chave Primária.
PROJ – Projeto.
PROJATV – Projetos – Atividades.
R – W – Read – Write.
SGBD – Sistema de Gerenciamento de Banco de Dados.
SGBDOR – Sistema Gerenciador de Banco de Dados Objeto – Relacional.
S_A – Servidor de Aplicação.
Sal – Salário.
S_B – Sistema de Backup.
S_P – Sistema de Produção.
SQL – Structured Query Language, Linguagem de Consulta Estruturada.
T – Transações.
TB – Terabyte.
VL – Valor.
WAL – Write ahead log, Arquivo de Escrita Prévia.
W – R – Write – Read.
W- W – Write – Write.

SUMÁRIO

1 INTRODUÇÃO

Sistemas computacionais têm sido usados em diversas áreas, atendendo uma enorme gama de atividades. Esses sistemas precisam de um meio de armazenamento que, por sua vez, precisa ser seguro, atender algumas propriedades exigidas e propiciar mecanismos de recuperação quando houver falhas inesperadas.

Atualmente, a grande maioria dos sistemas são suportados por robustos Sistemas Gerenciadores de Banco de Dados para alimentar as principais aplicações de negócios da empresa. Para isso, são necessários vários requisitos básicos para o seu funcionamento, tais como: segurança, *performance*, integridade dos dados e um bom mecanismo de recuperação ou *backup*.

Alguns exemplos do nosso cotidiano podem ser citados, tais como os modernos sistemas de reservas de passagens aéreas, que em caso de falhas do Banco de Dados, dificilmente as empresas administradoras terão condições de controlar as reservas. Outro exemplo são os sistemas bancários, onde os Sistemas Gerenciadores de Banco de Dados são cruciais para o controle de suas movimentações.

Segundo Breton (1998), os negócios de uma organização são disponibilizados com a necessidade crítica de assegurar a disponibilidade da informação e operação contínua de seus sistemas de missões críticas em face a falhas potenciais que variam desde perda de *hardware,* tais como falhas em discos e CPU, até falhas de *software*.

Com o crescente aumento das organizações de negócios dependentes de processamento de transações *online* para gerenciar seus dados, e o alto custo provocado por falhas, surge a demanda por sistemas que possuam serviços contínuos para nunca deixar o sistema ou as organizações dependentes do sistema inoperantes (Mehrotra, 1997).

Uma das principais preocupações na área da tecnologia de informação aplicada em uma empresa é manter o Banco de Dados disponível para uso em tempo real de resposta. Para isso, é necessário que se crie um controle de redução de falhas do sistema, visando manter a integridade e a confiabilidade da informação em tempo de resposta. Apesar desses cuidados, é ingênuo imaginar que nunca ocorrerão falhas mecânicas ou físicas que podem levar à perda de dados (ou de transformação). Em caso de falhas mecânicas ou físicas que possam retardar o tempo de resposta, é preciso tornar novamente o Banco de Dados operacional de maneira rápida e segura. Para proteger os dados dos diversos tipos de falhas que podem ocorrer, é necessário criar um mecanismo de *backup* do Banco de Dados regularmente. Sem um *backup* atualizado, não será possível ativar e executar o Banco de Dados em caso de perda de arquivo, sem perder os dados. Com os *backups* criados, a recuperação após diferentes tipos de falhas, é obtida com técnicas de *standby*[1] que é o enfoque desse trabalho.

Com o aprimoramento das redes de computadores e dos Servidores aliados a diminuição de custos de equipamentos, a técnica de *backup* utilizando Banco de Dados *Standby* torna-se extremamente atraente, garantindo um *backup* válido e eficiente de um Banco de Dados.

1.1 Objetivos

Aplicações de Banco de Dados têm, tradicionalmente, constituindo um campo com necessidades de tolerâncias a falhas, principalmente visando a integridade de dados e disponibilidade dos mesmos. Aplicações que empregam Sistemas Gerenciadores de Banco de Dados requerem que a integridade dos dados seja preservada durante a operação normal, bem

[1] *Standby* é uma técnica de *backup* de Banco de Dados utilizando como recurso um Servidor de *Backup.* No momento de sua utilização, será necessário um auxílio externo para que ele fique disponível para uso;

como após a recuperação ocasionada por uma falha. Preservar a integridade e a disponibilidade implica em medidas extras de segurança na forma de verificação de consistência e redundância. Em caso de falhas, informações vitais podem ser perdidas. Desta forma, uma parte essencial no Sistema de Banco de Dados é um esquema de recuperação responsável pela detecção de erros e pela restauração do Banco de Dados para um estado consistente, que existia antes da ocorrência da falha.

Este trabalho tem como objetivo estudar técnicas de *backup* e *recovery* de Banco de Dados utilizando Banco de Dados com tecnologia *open source*, focando no método de *backup* chamado Banco de Dados S*tandby*. O Banco de Dados escolhido foi o PostgreSQL 8.1, sendo a última versão disponível no mercado.

Quando se pretende trabalhar com técnicas de *backup* utilizando Banco de Dados S*tandby*, algumas etapas devem ser cumpridas. Primeiramente, será preciso configurar o SGBD para gerar arquivos de *logs* para que seja feita a replicação de dados de um Servidor de Banco de Dados de Produção, para um Servidor de Banco de Dados S*tandby* ou de *Backup*.

A partir desse estudo visa-se a estabelecer uma abordagem para a técnica de Banco de Dados S*tandby*, que deve incorporar as falhas que podem ocorrer em um Banco de Dados de Produção, os métodos de transferência de transações de um Banco de Dados de Produção para um *backup*, o marco inicial do uso do Servidor *Standby* e a replicação de dados.

A técnica de *backup* de Banco de Dados S*tandby* para aplicação de Banco de Dados Relacional é uma importante contribuição para administradores de Servidores de Banco de Dados que necessitam estar em operação 24 horas por dia e 7 dias por semana.

1.2 Motivação

Com o crescimento da utilização de um Sistema Gerenciador de Banco de Dados *Open Source*[2] nas empresas e instituições de ensino, há uma necessidade de realizar estratégias de *backups* válidas. A técnica de *backup* através de um Banco de Dados *Standby* ajuda a minimizar o tempo em que o Banco de Dados ficará fora do ar.

Courtright (1994) trabalha com redundância de informação baseada em *hardware*, utilizando redundância de Discos. Quando um erro é encontrado, o sistema entra em um estado *errôneo,* significando que há uma falha física de discos, essa falha é inconsistente com o estado do sistema como esperado. Um processo de recuperação de erros é executado para que seja restaurado para um estado consistente, ou seja, um estado livre de erros.

Mehrotra (1997) trabalha em seu artigo com a criação de um sistema de *backup* remoto, onde é feita uma cópia do Banco de Dados em um outro Servidor. Transações são executadas no Servidor de Banco de Dados Primário e arquivos de transações são gerados e propagados para o Servidor de *Backup*, o qual utiliza esses arquivos de transações para reconstruir o Banco de Dados para um estado atual.

O processo de redundância em um Sistema Gerenciador de Banco de Dados de alta disponibilidade permite a continuação de operações em caso de falhas e pode ser dividida em redundância de dados física e redundância de dados lógica (Drake, 2004).

Redundância de dados física refere-se à utilização de outros equipamentos ou *hardware* para manter múltiplas cópias físicas dos dados. Como exemplo, podemos ter espelhamento de discos, RAID, espelhamento remoto de discos e replicação de arquivos de sistema. Essa redundância é frequentemente combinada com processo de redundância usando

[2] *Open Source* é todo e qualquer software que permita a sua utilização para qualquer fim e sem restrições de custos;

uma área de armazenamento na rede. Isso permite múltiplos nós de acesso em um mesmo Banco de Dados físico. Caso um Servidor de Banco de Dados falhe, por motivo de *hardware* ou *software*, o Banco de Dados continua acessível por outros nós de acesso (Drake, 2004).

Redundância de dados lógica refere-se à situação onde o Banco de Dados explicitamente mantém múltiplas cópias dos dados. Transações aplicadas em um Banco de Dados primário são replicadas para um Banco de Dados secundário (Drake, 2004).

Várias razões levaram à motivação desse estudo, como:

a) proteção contra interrupção no Servidor de Produção, ou seja, o Servidor de Banco de Dados *Standby* pode ser usado como uma opção rápida de volta enquanto o Servidor Principal está sendo consertado;

b) proteção contra perda de arquivos de dados, no caso dos arquivos de dados do Banco de Dados de Produção se tornarem indisponíveis devido a uma pane de disco, exclusão acidental ou corrupção, o Servidor de Banco de Dados *Standby* pode ser ativado e usado como um Banco de Dados de Produção, enquanto o Banco de Dados de Produção Principal é recuperado;

c) proteção contra desastres, o Servidor de Banco de Dados *Standby* pode proteger os dados contra os desastres que possam ocorrer no Servidor Principal, tais como inundações, terremotos, incêndios ou outro fenômeno natural, os quais podem tornar o Banco de Dados de Produção inoperante;

d) simplicidade e baixo custo, a configuração e atualização de um Servidor de Banco de Dados *Standby* pode ser mais simples do que outras opções, tais como em um Servidor Paralelo ou um Servidor Replicado. Na comunidade de Banco de Dados, replicação é utilizada para melhorar *performance* e tolerância a falhas, mas, em atividades comerciais de Banco de Dados, essa replicação afeta diretamente o tempo gasto para realizar a replicação, diminuindo *performance* da aplicação (Wiesmann, 2000);

e) ganho de desempenho no Banco de Dados de Produção, como o Banco de Dados *Standby* está localizado em um Servidor separado, suas operações não afetam a operação do Servidor de Produção, o qual abriga o Banco de Dados de Produção. A CPU e a memória exigida para executar o Banco de Dados *Standby* e para receber e aplicar os *logs*[3] são consumidas totalmente dos recursos do Servidor de Banco de Dados S*tandby*. O único efeito é sobre os recursos de rede, os quais são consumidos na medida em que os arquivos de *logs* do Banco de Dados de Produção devem ser transmitidos para o Servidor de Banco de Dados S*tandby*;

f) utilização do Sistema Gerenciador de Banco de Dados (SGBD) open source, devido a sua alta aceitação no mercado e de ser de código livre e sem custos para utilização, esse tipo de Banco de Dados vem crescendo bastante nas empresas e instituições de ensino.

1.3 Organização do Trabalho

Neste primeiro capítulo foram apresentados o contexto, a motivação e os objetivos.

No Capítulo 2 são apresentados as terminologias e os conceitos básicos utilizados na recuperação e controle de transação do Banco de Dados.

No Capítulo 3 é apresentado o Sistema Gerenciador de Banco de Dados PostgreSQL e da linguagem de programação Java. Também é mostrado o histórico de utilização desse Banco de Dados, bem como suas principais funcionalidades e aplicações. O Capítulo é finalizado mostrando o surgimento da linguagem de programação Java e como seus programas são estruturados.

[3] *Logs* são arquivos que contêm as transações realizadas no Banco de Dados de Produção, será mais detalhada no Capítulo 2.7;

No capítulo 4 é feita uma revisão bibliográfica mostrando a estrutura funcional do SGBD e de seus arquivos de *logs*. Também são apresentadas as aplicações de Banco de Dados S*tandby*, fazendo definições do conceito de Banco de Dados S*tandby*, de *backup* e recuperação contra falhas. São ainda apresentados conceitos sobre a replicação de dados entre o Banco de Dados de Produção e o Banco de Dados de *Backup.* É trabalhado também o tratamento de falhas, onde é sugerido o acompanhamento das transações que estão ocorrendo em um Banco de Dados. Esse Capítulo trabalha ainda com a alteração do fluxo de transação onde é realizado o desvio das aplicações do Servidor de Produção para o Servidor de *Backup* e é finalizado com um caso de uso desse projeto.

No Capítulo 5 é apresentada a Ferramenta SABaDO, que foi desenvolvida neste trabalho e segue mostrando a arquitetura dessa ferramenta.

A Conclusão encerra o trabalho dando uma visão geral do projeto a ser desenvolvido para a obtenção do título de mestre em Ciência da Computação.

Alguns dos comandos SQL suportados pelo PostgreSQL são apresentados no Apêndice A, com definições e sintaxes.

Os comandos e arquivos utilizados pela Ferramenta SABaDO são documentados e detalhados no Apêndice B, onde é possível verificar suas variações e sintaxes, para aproveitar melhor o uso dessa Ferramenta.

2 CONCEITOS E TERMINOLOGIAS DE BANCO DE DADOS

Nesse capítulo, são discutidos alguns conceitos essenciais para o entendimento e o correto funcionamento de um Banco de Dados Relacional *Standby*.

2.1 Banco de Dados

O uso da Informática em empresas acontece de forma gradativa; primeiramente, algumas funções são automatizadas, e aos poucos, de maneira evolutiva e de acordo com as necessidades dessas empresas, todo o seu funcionamento torna-se informatizado, facilitando cada vez mais o dia-a-dia e o futuro de tais organizações comerciais.

O Banco de Dados é um arquivo físico, em dispositivos periféricos, onde ficam armazenados dados de vários sistemas, para que o usuário possa consultá-lo e atualizá-lo quando achar necessário, de acordo com Heuser (2001).

Para que isso seja compreendido de uma forma mais simples, considere como exemplo uma determinada empresa, de ordem fictícia, onde são executadas três funções básicas: Vendas, Produção e Compras.

No setor de Produção, concentra-se toda a parte relacionada à produção e controle do que foi produzido pela tal empresa.

No setor de Compras, fica toda a parte da empresa relacionada à aquisição de matéria-prima e de outros produtos necessários à produção, bem como os preços junto aos fornecedores, realização de compras e acompanhamento da entrega do que foi adquirido.

A empresa precisa que todos os setores funcionem em completa harmonia para que haja um desenvolvimento produtivo e satisfatório. Para isso, é necessário que sejam todos

informatizados de forma conjunta, e não arquivos independentes para cada função, pois todos os setores estão interligados e precisam de comunicação para que o trabalho se apresente de maneira proveitosa e com êxito.

Ainda, segundo Heuser (2001), dado é o valor do campo quando é armazenado no Banco de Dados. Um exemplo disso é o valor do campo "nome do produto" para quem está fazendo a entrada de dados, no setor de Compras, por exemplo, da empresa fictícia citada anteriormente.

Caso as funções do Banco de Dados sejam informatizadas separadamente, tal informação é representada no sistema em computadores por diversas vezes, ou seja, as informações relacionadas aos produtos, aparecem nos arquivos dos produtos de cada um dos três sistemas, sendo que isso não é necessário. Esse problema recebe o nome de *redundância de dados*, que se divide em *redundância controlada de dados* e *redundância não controlada de dados* (Heuser, 2001; Date, 2000).

Segundo Heuser (2001), a primeira acontece quando o *software* conhece essa múltipla representação de informações e garante essa sincronia entre elas, sendo usada em casos que se queira melhorar a confiança e/ou a velocidade geral do sistema. Como exemplo disso, tome um sistema distribuído, observe que apenas uma informação é armazenada em diversos computadores, permitindo acesso rápido, tendo como base qualquer um deles.

A *redundância não controlada de dados* surge quando o usuário e não o *software* se torna responsável pela manutenção da sincronia entre as várias representações de uma informação. Essa redundância deve ser evitada porque acarreta vários problemas, tais como: a) *entrada repetida da mesma informação,* os dados de um produto da empresa fictícia são cadastrados nos três setores, podendo resultar em erros de transcrição de dados, além de ser um trabalho totalmente desnecessário; e, b) *inconsistência de dados,* quando há alguma modificação na estrutura de um produto A, ela é informada através do sistema de produção,

mas pode deixar de ser passada essa mudança para os demais setores e isso fará com que a estrutura do produto apareça diferenciada nos demais sistemas, passando o Banco de Dados a ter informações inconsistentes (Heuser, 2001).

Para que esses problemas não aconteçam, é necessário que haja o compartilhamento de dados, sendo cada informação armazenada uma única vez e podendo ser acessada por diversos sistemas que precisem dela.

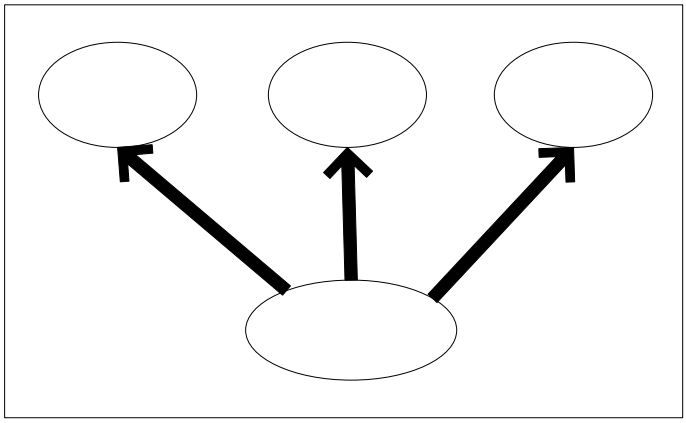

Figura 2.1 – Exemplo de Tabela de Compartilhamento de Dados.

Na Figura 2.1 é mostrado um exemplo onde o campo "produto" é compartilhado pelos três setores: Compras, Produção e Vendas. O compartilhamento desses dados reflete-se na estrutura do *software*, passando a estrutura interna dos arquivos a ser formada de acordo com a necessidade dos diferentes sistemas.

A esse conjunto de arquivos integrados, Heuser (2001) dá o nome de Banco de Dados, que para ele nada mais é do que "um conjunto de dados integrados que tem por objetivo atender a uma comunidade de usuários".

Já Young (1990) define Banco de Dados como:

> Banco de Dados corresponde a uma reunião de arquivos de dados de toda a organização em algum tipo de armazenamento magnético, sendo manipulado por um conjunto de programas. Tais programas efetuam operações de manutenção do Banco de Dados, como adições, exclusões e atualizações de dados, assim como processos de cálculo e regravação de informações no Banco de Dados e também operações de pesquisas de informações mais complexas.

Os sistemas de Banco de Dados estão disponíveis em máquinas que vão de pequeno porte até computadores de grande porte. Os recursos proporcionados por um determinado Banco de Dados são definidos pelo tamanho e pela potência da máquina onde ele se encontra.

Os Bancos de Dados se modificam com o passar do tempo, através da inserção e exclusão de informações, que são armazenadas no Banco de Dados em algum momento, sendo chamado de instância do Banco de Dados.

De acordo com Korth (2005), os primeiros Bancos de Dados se desenvolveram a partir de sistemas de gerenciamento de arquivos, que evoluíram primeiramente para Bancos de Dados de Rede ou Bancos de Dados Hierárquicos e, posteriormente, para Bancos de Dados Relacionais.

As características mais comuns dessas "velhas" aplicações, segundo Korth (2005) e Moraes (2003) são as seguintes:

a) Uniformidade, tendo muitos itens de dados estruturados de forma semelhante, todos apresentando o mesmo tamanho em *bytes*;

b) Orientação a Registro, os itens de dados básicos são compostos por registros de comprimento fixo;

c) *Itens de Dados Pequenos,* apresentando todos os registros curtos; com exceção de apenas alguns que são mais extensos do que algumas centenas de *bytes*; e,

d) *Campos Atômicos,* os campos existentes dentro de um registro são curtos e de comprimento fixo.

Para Chen (1990), Banco de Dados consiste em:

> Um Banco de Dados é uma coleção de registros de tipos diferentes. Os registros em um Banco de Dados são interligados, de forma que itens de dados relevantes em registros diferentes possam ser recuperados sem dificuldade.

Com o passar do tempo, novas aplicações de Bancos de Dados foram surgindo, principalmente por causa do aumento de memória principal que foi disponibilizada e pelo

tamanho do disco. Outra razão, também, é que o *hardware* se tornou mais barato e, assim, houve um crescimento no aprimoramento de gerenciamento de Banco de Dados, sendo o que ainda está ocorrendo atualmente.

2.2 Sistema Gerenciador de Banco de Dados (SGBD)

A programação de aplicações em computadores modificou-se expressivamente desde a sua invenção. No princípio, usavam-se linguagens como COBOL, Basic, C e outras, para que se incorporasse em um determinado programa toda a funcionalidade desejada. Nesses programas se continham todas as operações da *interface* de usuário, as transformações de dados e cálculos, as operações de armazenamento de dados e as tarefas de comunicação com outros sistemas e programas.

Com o passar do tempo, várias funcionalidades comuns a diversos programas foram aparecendo. No início do segundo milênio, muitos programas comunicam-se com os seus usuários através de *interfaces* gráficas de janelas, porém, na maioria das vezes esses programas não possuem todo o código relacionado com a exibição dos dados na *interface*, e utilizam para isso, gerenciadores de *interface* de usuário, que Heuser (2001) define como: "conjuntos de rotinas que incluem as funcionalidades que um programador vai necessitar freqüentemente, ao construir uma interface de usuário".

O Banco de Dados armazena registros dos clientes, tarefas e fontes, em forma de tabelas, sendo esta a finalidade de um programa chamado Sistema Gerenciador de Banco de Dados (SGBD) para armazenar e restaurar os dados nestas tabelas. É função da aplicação de Banco de Dados e do SGBD processar o formulário e armazenar os dados fornecidos em tabelas.

O SGBD é um *software* que é responsável pelo gerenciamento, ou seja, armazenamento e recuperação dos dados contidos nos Bancos de Dados.

Para comunicar-se com programas mais remotos, são utilizados Gerenciadores de Comunicação e para manter Bancos de Dados, são utilizados Sistemas de Gerência de Banco de Dados (SGBD).

Segundo Date (2000), "o Sistema Gerenciador de Banco de Dados é o *software* que manipula todos os acessos ao Banco de Dados". O SGBD é um *software* que funciona como uma *interface* entre o usuário e o Banco de Dados, ou seja, todas as solicitações dos usuários, como criação de tabelas, inserção de dados, recuperação de dados, são manipuladas pelo SGBD.

Outra função do SGBD é isolar os usuários do Banco de Dados dos detalhes ao nível do *hardware*, como por exemplo, o armazenamento das informações em um disco rígido. Em outras palavras, o SGBD faz com que os usuários tenham uma visão do Banco de Dados acima do nível do *hardware*.

Korth (2005), define SGBD como:

> Um Sistema Gerenciador de Banco de Dados (SGBD) é constituído por um conjunto de dados associados a um conjunto de programas para acesso a esses dados. O conjunto de dados, comumente chamado Banco de Dados, contém informações sobre uma empresa em particular.

O SGBD tem como finalidade propiciar um ambiente conveniente e eficiente na recuperação e armazenamento de informações do Banco de Dados.

Um sistema de Banco de Dados deve assegurar a integridade das informações armazenadas caso haja problemas com o sistema, impedindo também tentativas de acesso ao programa que não sejam permitidos.

A tecnologia de Banco de Dados foi amplamente desenvolvida para superar as limitações dos sistemas de processamento de arquivos. Para melhor compreensão, faça a comparação entre um sistema de processamento de arquivos, sendo os arquivos de dados

armazenados acessados diretamente e os programas de processamento de Banco de Dados, que chamam o SGBD para acessar os dados que estão armazenados. Esta diferença é importante, pois torna a tarefa de programação da aplicação mais fácil; isto é, os programadores de aplicações não têm que estarem preocupados com a forma como os dados estão fisicamente armazenados. Na realidade, eles ficam liberados para se concentrarem em assuntos mais importantes para os usuários, ao invés de assuntos importantes para o computador.

Hoje, no mercado, há diversos tipos de SGBD, mas atualmente há um que tem dominado o mercado, *o Modelo de Dados Relacional*.

2.3 Modelos de Dados de Entidade e Relacionamento

O Modelo de Dados identifica a estrutura interna de recuperação e armazenamento dos dados no qual o SGBD foi projetado.

Segundo Korth (2005), existem três modelos de dados:

> Sob a estrutura do Banco de Dados está o *modelo de dados*: um conjunto de ferramentas conceituais usadas para a descrição de dados, relacionamentos entre dados, semântica de dados e regras de consistência. Os vários modelos que vêm sendo desenvolvidos são classificados em três diferentes grupos: modelos lógicos com base em objetos, modelos lógicos com base em registros e modelos físicos.

Os modelos lógicos com base em objetos são utilizados ao se descrever dados no nível lógico e de visões e se caracterizam por utilizar recursos de estruturação mais flexíveis e propiciar a especificação explícita das restrições de dados, sendo os mais conhecidos: Modelo Entidade-Relacionamento, Modelo Orientado a Objetos, Modelo Semântico de Dados e Modelo Funcional de Dados.

O Modelo de Dados Entidade-Relacionamento baseia-se na percepção da realidade como um conjunto de objetos básicos denominados de entidades e do relacionamento entre eles (Korth, 2005).

O Modelo de Dados Orientado a Objetos baseia-se num conjunto de objetos que apresentam valores armazenados em variáveis instâncias dentro do objeto, contendo, também, conjuntos de códigos que operam esse objeto, sendo esses conjuntos conhecidos como métodos.

Os Modelos Lógicos com base em Registros são usados na descrição de dados no nível lógico e de visão, bem como para especificar a estrutura lógica do Banco de Dados quanto para implementar uma descrição de alto nível. São três os Modelos de Dados com base em Registro: Modelo Relacional, Modelo de Rede e Modelo Hierárquico, sendo o primeiro o que mais tem se destacado atualmente.

Chen (1990) afirma que há muitos sistemas de Bancos de Dados em uso no momento e classifica-os em três categorias: hierárquico, de rede e relacional, que se diferenciam um dos outros pelo tipo de estrutura lógica de dados que podem suportar.

Os Sistemas Hierárquicos de Bancos de Dados necessitam que os tipos de registros de dados estejam ordenados de forma hierárquica. Essa estrutura tem um bom funcionamento em alguns Bancos de Dados que apresentam uma hierarquia natural entre os tipos de registro.

A técnica de modelagem de dados mais conhecida e utilizada é o Modelo de Entidade e Relacionamento (MER). Foi criada em 1976 por Peter Chen e pode ser considerada como um padrão para modelagem conceitual.

De acordo com Munari (2002):

> Em 1976 foi lançada pelo Prof. Peter Chen a metodologia chamada de "Entidade-Relacionamento", muito utilizada atualmente para fins de representação de modelos de dados, que podem posteriormente ser implementados em SGBDs de várias filosofias diferentes tais como relacionais, objeto-relacional, objeto, etc. Posteriormente, com a continuidade dos estudos na área da modelagem, novos conceitos e técnicas foram acrescentados àqueles propostos inicialmente por Chen, originando-se

> aquilo que se convencionou chamar de modelo Entidade/Relacionamento estendido.

O Modelo Entidade-Relacionamento (E-R) baseia-se na percepção de que o mundo real é composto por um conjunto de objetos – "entidades" e pelo conjunto dos relacionamentos entre esses objetos.

Segundo Korth (2005), "o modelo E-R é um dos modelos com maior capacidade semântica; os aspectos semânticos do modelo se referem à tentativa de representar o significado dos dados".

Suas principais características são: a) *Representação gráfica de um modelo de dados*, *através de diagramas E-R*; b) *Preocupação com a semântica dos relacionamentos*; c) *Fácil entendimento para usuários leigos*; d) *Independência de dados e de sistemas gerenciados por Bancos de Dados*; e, e) *Permite a construção de modelos que apresentem maior estabilidade* (Korth, 2005).

São três as noções básicas utilizadas pelo modelo E-R: *conjunto de entidades, conjunto de relacionamentos e os atributos.*

a) *Conjunto de Entidades*: São coisas do mundo real do qual se deseja armazenar informações, tais como pessoas (físicas ou jurídicas), objetos (materiais ou abstratos) e eventos.

Korth (2005) define entidades como "Uma entidade é uma "*coisa*" ou um "*objeto*" no mundo real que pode ser identificada de forma unívoca em relação a todos os outros objetos".

Cada pessoa, em uma empresa, é uma entidade, que apresenta um conjunto de propriedades com valores únicos, como por exemplo, o número social 988-69-8700 identifica somente uma pessoa na empresa, não podendo o mesmo identificar mais do que uma.

Um conjunto de entidades que compartilham das mesmas propriedades é chamado de *atributos,* que são propriedades que descrevem cada parte de um conjunto de entidades. Esses conjuntos não são necessariamente separados.

A designação de atributos para um conjunto de entidades mostra que o Banco de Dados mantém informações semelhantes de cada uma das entidades desse conjunto, porém, cada entidade pode apresentar um valor próprio em cada atributo, ao qual há um conjunto de valores que são possíveis a ele, chamado de *domínio* ou *conjunto de valores* daquele atributo (Korth, 2005).

O atributo é uma função que relaciona o conjunto de entidades a seu domínio. Desde que um conjunto de entidades tenha alguns atributos, cada entidade pode ser exemplificada pelo conjunto formado pelos pares – atributos, valores de dados, um par para cada atributo do conjunto de entidades.

b) *Conjunto de Relacionamentos*: De acordo com Heuser (2001), "relacionamento é um conjunto de associações entre entidades".

Uma entidade desempenha uma função muito importante em um relacionamento, essa tal função é denominada de papel. Quando os conjuntos de entidades participantes em um conjunto de relacionamentos são distintos, papéis são implícitos e geralmente não são especificados, porém, são de grande utilidade quando o significado de um relacionamento necessita de esclarecimento.

Na Figura 2.2 é apresentado um esquema do modelo de entidades e relacionamentos, descrito logo a seguir:

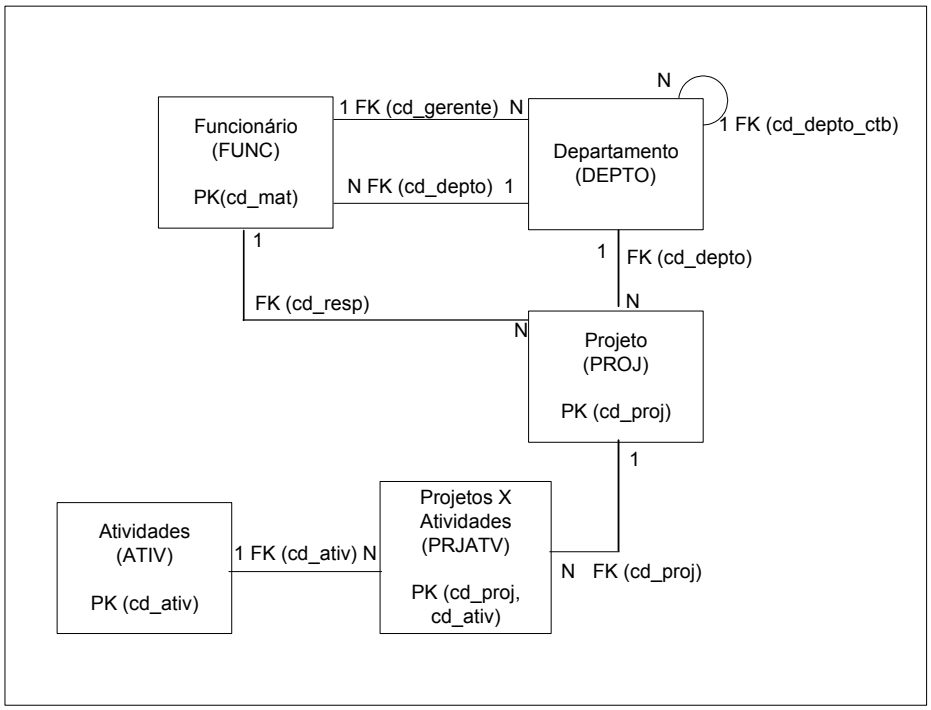

Figura 2.2: Exemplo de Diagrama Entidade-Relacionamento

A Tabela *FUNC (Funcionários)* tem como *primary key* (PK) a coluna *cd_mat* e como *foreign key* (FK) a coluna *cd_depto*, que estabelece relacionamento com a tabela *DEPTO*. A Tabela *DEPTO (Departamento)* tem como *primary key* a coluna *cd_depto* e como *foreign key* a coluna *cd_gerente*, que estabelece relacionamento com a Tabela *FUNC*, e a coluna *cd_depto_ctb*, que estabelece um auto-relacionamento. A Tabela *PROJ (Projeto)* tem como *primary key* a coluna *cd_proj* e como *foreign key* a coluna *cd_resp*, que estabelece relacionamento com a Tabela *FUNC*, e a coluna *cd_depto*, que estabelece relacionamento com a Tabela *DEPTO*. A Tabela *ATIV (Atividades)* tem como *primary key* a coluna *cd_ativ* e não possui relacionamentos com outras tabelas. E a Tabela *PRJATV* (*Projetos-Atividades)* tem como *primary key* as colunas *cd_proj* e *cd_ativ*, que simultaneamente agem como *foreign key* e estabelecem relacionamento com as tabelas *PROJ* e *ATIV*, respectivamente.

Segundo Elmasri e Navathe (2005), cada *linha* na tabela representa uma coleção de valores de dados relacionados. Esses valores podem ser interpretados como *fatos* que representam uma *entidade* do mundo real ou um relacionamento. O nome da tabela e os

nomes das colunas são usados para interpretar os valores em cada linha da tabela. Por exemplo, uma tabela chamada FORNECEDOR contém informações sobre os fornecedores de uma determinada organização, onde cada linha ou registro contêm os dados sobre um determinado fornecedor. Os nomes das colunas *id, razao_social, nome_fantasia, nr_cgc* interpretam os valores específicos para cada linha, obedecendo aos tipos de valores das colunas.

2.4 Modelo Relacional

O Modelo de Dados Relacional foi introduzido por E. F. Codd em 1970. O modelo é baseado em uma estrutura de dados simples denominada *relação* e tem uma fundamentação teórica apoiada na álgebra relacional (Spoto, 2000). Cada *relação* pode ser considerada como uma *tabela*, cada qual designada por um nome único.

Fernandes (2000), define o Modelo de Dados Relacional da seguinte maneira:

> o Banco de Dados relacional tem como objetivo implementar o modelo de Banco de Dados relacional, com todas as suas características básicas, ou seja, entidades, atributos e relacionamentos.

O Modelo de Dados Relacional é o Modelo Lógico com Base em Registros mais utilizados atualmente, dividindo espaço com o Modelo de Rede e o Modelo Hierárquico, sendo que ambos são utilizados somente em Bancos de Dados mais antigos.

De acordo com Korth (2005), o Modelo de Dados Relacional foi o primeiro modelo de dados utilizado para aplicações comerciais.

O Modelo de Dados Relacional representa os dados contidos em um Banco de Dados através de relações, que possuem informações sobre as entidades representadas e seus relacionamentos. O Modelo Relacional baseia-se no conceito de matrizes, onde as chamadas linhas (das matrizes) seriam os “registros” e as colunas (das matrizes) seriam os “campos”. Os

nomes das tabelas e dos campos são de fundamental importância para a compreensão entre o que está sendo armazenando, onde está sendo armazenado e qual a relação existente entre os dados armazenados (Korth, 2005).

Um Banco de Dados Relacional é formado por *tabelas* ou *relações.*

2.4.1 Tabelas

Todos os dados de um Banco de Dados Relacional são armazenados em tabelas. Uma tabela é uma simples estrutura de linhas e colunas. Cada linha contém um mesmo conjunto de colunas, mas as linhas não seguem qualquer tipo de ordem. Em um Banco de Dados podem existir uma ou centenas de tabelas. O limite é imposto unicamente pela ferramenta de *software* utilizada. Um esquema de Banco de Dados Relacional S é um conjunto de esquemas de relação S={$R_1, R_2, .., R_m$} e um conjunto de restrições de integridade denominado de instâncias de relação DB = {$r_1, r_2, .., r_m$} tal que cada r_i é uma instância de R_i (Elmasri, 2005). Cada esquema de Banco de Dados de Relação também é conhecido como Tabela do Banco de Dados.

Quando uma tabela é lida, as linhas são retornadas randomicamente, ou seja, não há uma ordenação pré-definida, a não ser que seja exposto por ordenação no Comando SQL. Cada coluna possui um tipo, *constraint* e possível valor *default* que possa ser atribuído a ela.

Cada campo recebe um nome de campo (atributo). O conjunto de campos homônimos de todas as linhas de uma tabela forma uma coluna, como pode ser observado no exemplo da Figura 2.3:

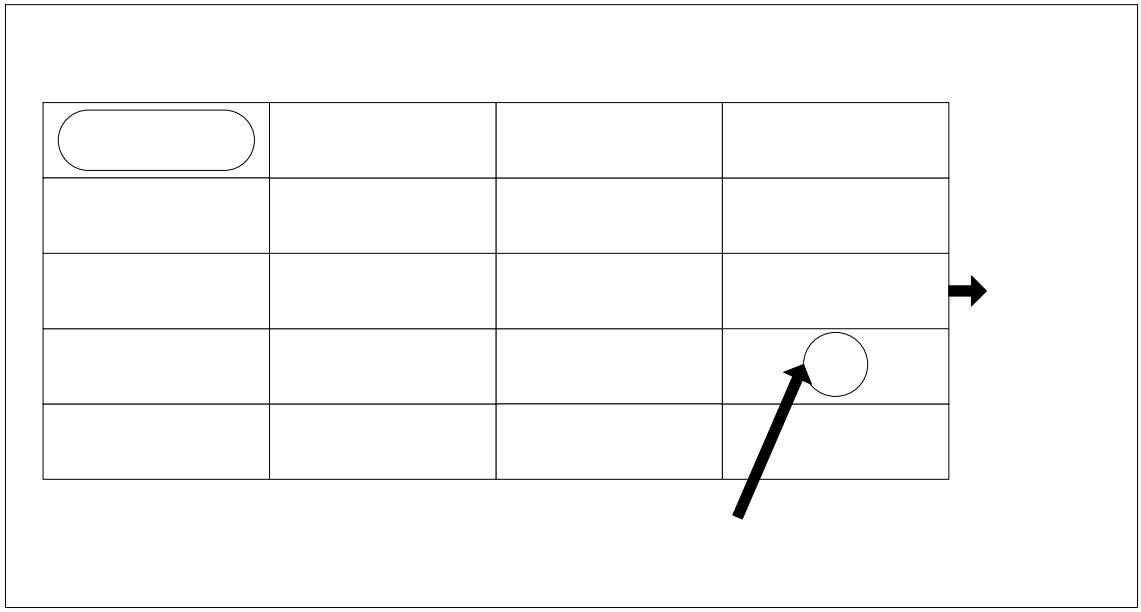

Figura 2.3: Exemplo de Tabela

Toda tabela apresenta linhas e colunas, onde cada linha especifica algo sobre um único funcionário em questão; onde cada coluna especifica um tipo de campo a ser preenchido com a descrição de todos os funcionários da tal empresa fictícia, como foi citado anteriormente.

Cada coluna recebeu um nome especial nessa tabela, como: “Código de Funcionário”, “Nome”, “Código de Departamento” e “Categoria Funcional”.

Cada linha da tabela, por exemplo, refere-se a um único funcionário F5, que foi cadastrado no sistema como “Novaes”, apresentando o Código de Departamento D1 e pertencendo à Categoria Funcional C5.

Korth (2005), descreve tabela de tal forma:

> Um Banco de Dados em conformidade com o esquema de Banco de Dados E-R pode ser representado por uma coleção de tabelas. Para cada conjunto de entidades e para cada conjunto de relacionamentos, dentro de um Banco de Dados, existe uma tabela única registrando o nome do conjunto de entidades ou relacionamentos correspondente. Cada tabela possui várias colunas, cada uma delas com um único nome.

2.4.2 Chaves

São utilizadas para identificar linhas e estabelecer relações entre linhas de tabelas de um Banco de Dados Relacional.

Existem três tipos de chaves em um Banco de Dados Relacional: *chave primária*, *chave alternativa* e *chave estrangeira*.

Uma chave primária é uma coluna ou a combinação de colunas cujos valores diferenciam uma linha das outras dentro de uma tabela.

A chave primária (*PK- Primary Key)* é definida como "a chave que identifica cada registro dando-lhe unicidade, a chave primária nunca se repetirá", (Heuser, 2001).

Segundo Elmasri e Navathe (1994), o atributo na chave estrangeira *Foreign Key* (FK) de um esquema de relação R_1, tem o mesmo domínio que o da chave primária PK de outro esquema de relação R_2. É o mecanismo que permite a implementação de relacionamentos em um Banco de Dados Relacional.

Quando se faz uso de chave estrangeira, algumas restrições devem ser seguidas ao se executar variadas operações de alteração do Banco de Dados, tais como:

a) *Inclusão de uma linha na tabela que contém a chave estrangeira*, garantindo que o valor da chave estrangeira apareça na coluna da chave primária referenciada;

b) *Alteração do valor da chave estrangeira*, deve-se ter a garantia de que um novo valor de uma chave estrangeira apareça na coluna da chave primária referenciada;

c) *Exclusão de uma linha da tabela que contém a chave primária referenciada pela chave estrangeira*, deve-se garantir que não apareça na coluna da chave estrangeira o valor da chave primária que está sendo excluída; e,

d) *Alteração do valor da chave primária referenciada pela chave estrangeira*, deve-se ter a garantia que na coluna chave estrangeira não apareça o valor antigo da chave primária que está sendo modificada.

2.5 Linguagem de Banco de Dados Relacional – SQL

Um Banco de Dados representará sempre os aspectos do mundo real, sendo assim, uma Base de Dados (BD) é uma fonte de onde pode-se retirar uma imensa gama de informações derivadas, que possui um nível de interação com eventos com o mundo real que representa. A forma mais comum de interação *Usuário* e *Banco de Dados* se dá através de sistemas específicos que por sua vez acessam o volume de informações, geralmente através da linguagem SQL.

Existem inúmeras versões de SQL (*Structured Query Language,* Linguagem de Consulta Estruturada). A versão original foi desenvolvida no Laboratório de Pesquisa da IBM - Laboratório de Pesquisa de San José – hoje o Centro de Pesquisa Almaden (Korth, 2005). Esta linguagem, primeiramente chamada de *Sequel*, foi projetada e implementada como uma *interface* para um Sistema de Banco de Dados Relacional chamado sistema R, no início de 1970. A Linguagem *Sequel* evoluiu e seu nome foi mudado para SQL.

Atualmente a Linguagem SQL é a mais usada pelos SGBDs comerciais e tem sido bastante modificada pelos fornecedores, o que resultou em diferentes tipos de comandos, algumas vezes incompatíveis com outros Sistemas Gerenciadores de Banco de Dados. Houve, então, a necessidade de uma padronização, que aconteceu em 1986 com o *American National Standard Institute* (ANSI) publicando um padrão SQL (Korth, 2005).

A SQL contém comandos que realizam uma variedade de tarefas, tais como: consulta aos dados; inserir, atualizar e remover linhas de uma tabela; criar, modificar e remover

objetos do Banco de Dados; controlar o acesso ao Banco de Dados e seus objetos; e garantir a sua consistência.

Existem três linguagens de operação, que são: DML (*Data Manipulation Language*, Linguagem de Manipulação de Dados), a DDL (*Data Definition Language*, Linguagem de Definição de Dados) e a DCL (*Data Control Language*, Linguagem de Controle de Dados), (Elmasri e Navathe, 2005).

A DML abrange os comandos que auxiliam os usuários a terem acesso e a armazenar os seus dados, sendo eles: INSERT, UPDATE e DELETE, usados para manipular os dados das relações. O comando INSERT é utilizado para inserir novas linhas nas tabelas em um específico Banco de Dados. UPDATE atualiza as linhas já existentes de uma tabela em determinado Banco de Dados. E o comando DELETE remove as linhas das tabelas de um Banco de Dados específico.

Os comandos existentes na DDL descrevem como as tabelas e outros objetos podem ser definidos, alterados e removidos, e se classificam em: CREATE, ALTER e DROP. O comando CREATE é utilizado dinamicamente para fazer a definição de relações. ALTER é o comando que altera a estrutura de dados das relações. E o comando DROP é usado na remoção de relações.

Os comandos que compõem a Linguagem de Controle dos Dados (DCL) dão uma visão mais ampla dos mecanismos de controle, e dividem-se em três grupos de comandos: *comandos para controle de transação, comandos para controle de sessão* e *comandos para controle de sistema* (Fernandes, 2002).

Os *comandos para controle da transação* são necessários para que se possa controlar a efetivação ou não das alterações realizadas no Banco de Dados, como afirma Fernandes (2002):

> um software de Banco de Dados deve ser capaz de garantir a integridade física e lógica da Base de Dados, independente do número de usuários

> simultâneos que estejam atualizando ou consultando dados, para isso, o software utiliza mecanismos que permitam a concorrência.

Os comandos para controle de transação são: COMMIT, SAVEPOINT e ROLLBACK. O comando COMMIT efetua as alterações de transação corrente de maneira permanente, elimina todos os SAVEPOINTS da transação, termina a transação e libera os *locks* causados pela transação. Se uma transação termina de forma anormal, as atualizações da transação corrente não serão atualizadas. COMMITs automáticos ocorrem depois de um comando DDL e ao término normal de uma sessão de trabalho no Banco de Dados. O comando SAVEPOINT pode ser usado para dividir uma transação e também possibilita o retorno a um ponto, desprezando as alterações ocorridas após aquele ponto. Se criar um SAVEPOINT com o mesmo nome de um anterior, o anterior será eliminado (Fernandes, 2002).

O comando ROLLBACK é utilizado para desfazer o trabalho. A utilização desse comando sem a cláusula SAVEPOINT, irá finalizar a transação, não efetivará as alterações de transação, eliminará todos os SAVEPOINTS da transação e eliminará, também, os *locks* causados pela transação. Utilizando o comando ROLLBACK com a cláusula SAVEPOINT partes das alterações da transação serão desprezadas.

Os *comandos para controle de sessão* são utilizados para modificar as características de uma sessão do usuário, podendo ser REVOKE e GRANT. O comando REVOKE remove ou restringe a capacidade de um usuário de executar operações e o comando GRANT autoriza ao usuário executar ou setar operações.

Por fim, a Linguagem SQL possui um comando SELECT para recuperação de informações, sendo este o comando mais utilizado para recuperar os dados do Banco de Dados. Ele retorna linhas de uma ou mais tabelas ou de uma visão.

2.6 Transação

Uma transação é um conjunto de comandos SQL que realizam uma mesma unidade lógica de ação. Uma transação pode ter dois finais; ou será confirmada e gravada no Banco de Dados ou será descartada. Quando chega ao final, seja de qualquer uma das formas, uma nova transação é iniciada.

De acordo com Korth (2005), "uma transação é uma unidade de execução de programa que acessa e possivelmente atualiza vários itens de dados"**.**

Uma transação geralmente é o resultado de um programa de usuário escrito em uma linguagem de manipulação de dados de alto nível ou em uma linguagem de programação, por exemplo, C, Pascal, Cobol. A transação consiste em todas as operações ali executadas, entre o começo e o fim da transação. Por exemplo, para transferir uma quantia em dinheiro de uma conta A para uma conta B, devemos retirar o dinheiro da conta A e depositar o mesmo na conta B. Para que a operação acima seja segura, ou os dois passos são executados ou nenhum deles pode ocorrer.

O início de uma transação é determinado pelo comando BEGIN do PostgreSQL ou pelo comando START TRANSACTION. O fim de uma transação é determinado de uma das seguintes formas: a) execução do comando COMMIT, todas as modificações são efetivadas no Banco de Dados; b) execução do comando ROLLBACK, nenhuma modificação é efetivada.

Para assegurar a integridade dos dados, é necessário que o sistema de Banco de Dados mantenha as seguintes propriedades das transações:

a) atomicidade, ou seja, todas as operações da transação são refletidas corretamente no Banco de Dados ou nenhuma o será;

b) consistência, a execução de uma transação isolada, ou seja, sem a execução concorrente de outra transação, preserva a consistência do Banco de Dados;

c) isolamento, embora diversas transações possam ser executadas de forma concorrente, o sistema garante que, para todo par de transações T_1, T_2,..., T_n tem a sensação de que T_2 terminou sua execução antes de T_1 começar, ou que T_2 começou sua execução após T_1 terminar. Assim, cada transação não toma conhecimento de outras transações concorrentes no sistema;

d) durabilidade, ou seja, depois da transação completar-se com sucesso, as mudanças que ela faz no Banco de Dados persistem, até mesmo se houver falhas no sistema (Date, 2000).

Na ausência de falhas, todas as transações completam-se com êxito. Entretanto, nem sempre uma transação pode completar-se com sucesso. Nesse caso, a transação é abortada ou cancelada. Seguindo a propriedade de atomicidade, uma transação abortada não deve ter efeito algum sobre o estado do Banco de Dados. Assim, quaisquer atualizações que a transação abortada tiver realizado no Banco de Dados devem ser desfeitas.

Korth (2005) define ROLLBACK como:

> quando as mudanças causadas por uma transação abortada são desfeitas, diz-se que foi feito ROLLBACK. Toda transação é controlada por um segmento de *rollback* que nada mais é do que uma pilha de transações pendentes que aguardam a validação (COMMIT) ou não (ROLLBACK). Se acaso o ROLLBACK for acionado, as buscas de cancelamento das transações são realizadas na pilha de *rollback*.

No exemplo apresentado na Figura 2.4 é feita uma conexão do *Usuário 1* no Sistema Gerenciador de Banco de Dados e este faz uma seleção das informações da tabela de *DEPT (DEPARTAMENTO)*. Em seguida, é feita a inserção de dois novos registros e novamente é feita a seleção. Esse usuário consegue visualizar as informações inseridas, mesmo que não tenha sido finalizada a transação. Agora, se outro usuário tentar fazer a seleção dos dados

dessa tabela, será impossível visualizar os registros inseridos, a não ser que o *Usuário 1* finalize a transação com o comando COMMIT.

Usuário 1

```
SELECT deptno, dname FROM dept;

DEPTNO      DNAME
----------  --------------------
10          ACCOUNTING
20          RESEARCH
30          SALES
40          OPERATIONS

INSERT INTO dept values (50, 'MARKETING', 'SAO PAULO');

INSERT INTO dept values (60, 'INFORMATICA', 'CAMPINAS');

SELECT deptno, dname FROM dept;

DEPTNO      DNAME
----------  --------------------
10          ACCOUNTING
20          RESEARCH
30          SALES
40          OPERATIONS
50          MARKETING
60          INFORMÁTICA
   Commit;
```

Figura 2.4: Exemplo de um Cenário de Transação.

No exemplo apresentado na Figura 2.5 observa-se que o *Usuário 2* consegue visualizar as alterações somente depois que o *Usuário 1* finaliza a transação, caso contrário, é visto o valor antigo, para garantir a consistência de leitura. Isso ocorre devido ao controle do segmento de *rollback* criado pelo SGBD para preservar as informações ainda não liberadas na transação ou denominados de controle de *locks,* que geralmente são realizados nos registros das relações que estão em fase de transação e ainda não foram concretizadas.

USUÁRIO 1	**USUÁRIO 2**
SELECT sal FROM emp WHERE empno = 7369;	SELECT sal FROM emp WHERE empno = 7369;
SAL ------- 800	SAL ------- 800
UPDATE emp SET sal = 4000 WHERE empno = 7369;	
SELECT sal FROM emp WHERE empno = 7369;	SELECT sal FROM emp WHERE empno = 7369;
SAL ------- 4000	SAL ------- 800
Commit;	
SELECT sal FROM emp WHERE empno = 7369;	SELECT sal FROM emp WHERE empno = 7369;
SAL ------- 4000	SAL ------- 4000

2.5: Exemplo de Visão Geral de uma Transação

Transações são seqüências de operações de leitura ou gravação seguidas de um *commit* ou *rollback*. Transações que contêm somente operações de leitura são chamadas de *consultas*, e transações que contêm leitura e gravação são chamadas de *transações alteradas* (Wiesmann, 2000).

2.7 Arquivos de Transação (*Redo Logs*)

De acordo com Korth (2005), "*log* é a estrutura mais usada para gravar modificações no Banco de Dados". Os arquivos de *log* guardam as atualizações das atividades (transações) executadas no Banco de Dados.

Sempre que uma transação realiza uma alteração, inserção ou exclusão de informações no Banco de Dados, é essencial que seja criado um registro de *log* para aquela

alteração. Existindo esse registro de *log* será possível enviar a modificação existente no Banco de Dados de Produção para um outro Banco de Dados, ou seja, um Banco de Dados de *Backup*. Os registros de *logs* contêm a descrição completa de toda atividade do Banco de Dados, ou seja, ele contém informações como o início da transação; quais atributos sofreram alteração, armazenando o valor antigo e o novo valor; uma instrução de *commit*, ou seja, um indicativo de que a transação foi finalizada; ou uma instrução de cancelamento ou *rollback*, de acordo com Korth (2005).

2.8 Falhas

Segundo Verhofstad (1978), uma falha é "um evento em que o sistema não é executado de acordo com o especificado"**.** Algumas falhas são provocadas por falha de *hardware* (disco danificado); ou em caso de falha no sistema elétrico (ocasionado pela falta de Energia). Uma outra falha seria a falha de *software*, como exemplo, programas mal escritos ou dados inválidos, ocasionada por falhas humanas.

Vários tipos de falhas podem ocorrer em um sistema, cada um dos quais exigindo um tratamento diferente. O tipo de falha mais simples de tratar é aquele que não resulta na perda de informação no sistema. As falhas mais difíceis de tratar são aquelas que resultam em perda de informação.

Segundo Korth (2005), as falhas se classificam como:

a) falha de transação, que pode ser dividida em erro lógico, sendo que a transação não pode mais continuar com sua execução normal devido a alguma condição interna, como uma entrada inadequada, um dado não encontrado; e *overflow* ou limite de recurso excedido ou erro de sistema, em que o sistema entrou em um estado inadequado, por exemplo, *deadlock*, que é uma situação de bloqueio múltiplo em que as transações envolvidas esperam

umas pelas outras sem que seja possível a alguma delas continuar. Isto causa um bloqueio recíproco que só pode ser notado e solucionado pelo Banco de Dados. E com isso, uma transação não pode continuar com sua execução normal;

b) queda do sistema, quando há algum mau funcionamento de *hardware* ou um *bug*[4] no *software* de Banco de Dados ou no sistema operacional;

c) falha de disco, ou seja, um bloco de disco perde seu conteúdo em função de quebra do cabeçote ou da falha durante uma operação de transferência de dados;

d) *falhas ocasionadas por um dos seguintes casos: falta de energia elétrica; ou incêndio; ou inundação; ou qualquer outro fenômeno da natureza*, colocando todo o trabalho realizado até o momento em risco de perda total.

2.9 Processo *Standby*

Backup remoto é uma técnica usada em aplicações críticas para arquivamento contínuo de operações de Banco de Dados. Uma cópia do Banco de Dados Primário é transportada para um Servidor Remoto geograficamente situado em outro lugar que recebe o processamento das transações quando o Servidor Primário falhar (Molina, 1990).

O processo de Banco de Dados *Standby* consiste de um Banco de Dados Primário, que contém uma réplica dos Dados de Produção. É uma cópia de um Banco de Dados Primário utilizado essencialmente para a recuperação no caso de desastres do Banco de Dados Primário ou Principal. De acordo com a Figura 2.6, o Banco de Dados *Standby* está normalmente no estado de *Recovery*, aplicando as transações que ocorrem no Banco de Dados Primário.

[4] *Bug* é quando o sistema não funciona de forma adequada;

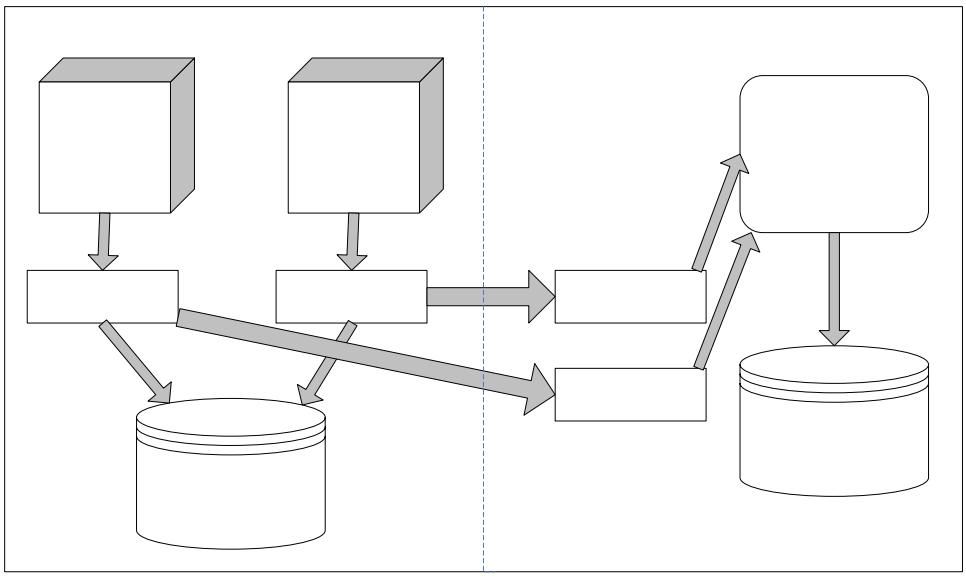

Figura 2.6: Exemplo de Representação do Banco de Dados Standby

Conforme pode-se observar na Figura 2.6, quando o Banco de Dados *Standby* é ativado, ele se torna o Banco de Dados de Produção e compatível com o Banco de Dados Primário que sofreu um desastre.

Essa seria uma representação do funcionamento de duas máquinas ou Servidores, um contendo o Banco de Dados Primário ou de Produção e outra contendo o Banco de Dados Secundário ou de *Backup*, aqui chamado de Servidor S*tandby*, (ver Figura 2.6). O que está acontecendo é que diversos usuários estão trabalhando normalmente no Banco de Dados Principal. As transações, na medida em que são finalizadas, são enviadas para um Banco de Dados *Standby*, que tem como tarefa receber essas transações e aplicá-las, fazendo com que esse Servidor sempre esteja atualizado.

2.10 Segmentos de *Rollback*

Cada Banco de Dados apresenta um ou mais segmentos de *rollback.* Esta área tem como finalidade registrar as alterações feitas para cada transação, sendo usada para

cancelamento das transações, consistência de leitura[5] e recuperação do Banco de Dados em caso de falha (Fernandes, 2002).

O segmento de *rollback* registra várias informações, tais como identificador da transação e descrição da informação como ela existia antes de ser modificada.

Cada modificação realizada em um dado de uma tabela é registrada no segmento de *rollback*. Mas o que realmente fica registrado não é a modificação e sim a imagem anterior do dado, ou seja, a cópia do dado antes da alteração. A modificação que deseja ser realizada é feita diretamente no Banco de Dados.

Quando se efetua uma consulta no Banco de Dados e a informação ainda não foi efetivada (processo de realização do *Commit)*, obtêm-se todas as linhas que não foram alteradas nos blocos de dados em memória e todas as informações modificadas diretamente do segmento de *rollback* em memória. O segmento de *rollback* registra os dados antes da alteração; por esta razão ele é usado para *rollback* e pelo mecanismo de consistência de leitura por apresentar a imagem original do dado (Fernandes, 2002).

O Banco de Dados não consegue iniciar se não tiver acesso a pelo menos um segmento de *rollback,* pois é esse segmento que auxilia no controle de todas as modificações.

Na Figura 2.7, vê-se o espaço reservado para um determinado *tablespace*[6] ser usado por quatro segmentos: um segmento *FUNC*, um segmento *PROJ,* um segmento de índice e um segmento de *rollback*.

[5] *Consistência de leitura* refere-se ao ato de consultar uma informação quando ela está sendo modificada por outro usuário, nessa situação, a informação é consultada do segmento de *rollback*;
[6] *Tablespace* é o local de armazenamento de tabelas;

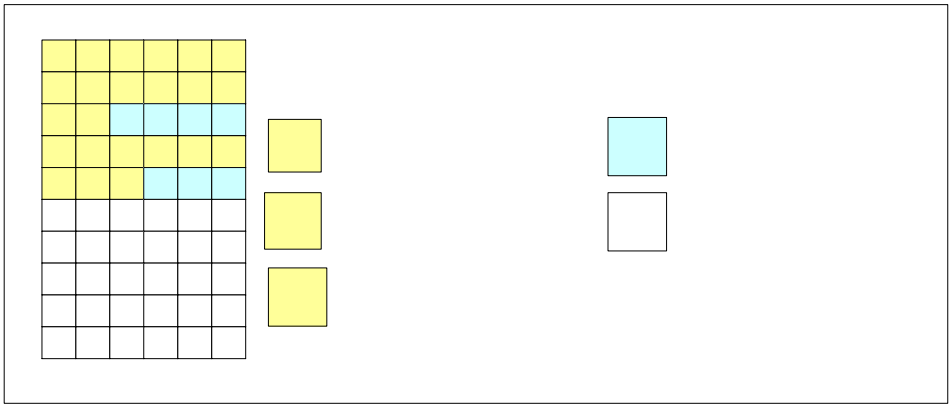

Figura 2.7: Exemplo de Alocação de Dados em um Segmento de Rollback.

Assim, pode-se dizer que um Banco de Dados contém tipos de segmentos dos quais são conhecidos três deles: o segmento para armazenamento de dados; o segmento para armazenamento de índices; e o segmento para informações de *rollback*.

Na Figura 2.7, os segmentos *FUNC* e *PROJ* são segmentos de dados, o segmento *S1* é de índice e o segmento *R* é de *rollback*, ficando claro que um segmento pode ser formado por várias partes de informação, isto é, não necessariamente é contínuo. Observe que o segmento *FUNC* possui duas partes (extensões), assim como o segmento *R* e o *P*. As duas partes que o segmento possui são chamadas de extensão, que vêm a ser uma área contínua para armazenamento de informações, um conjunto de blocos contíguos, podendo um segmento ser formado por uma ou mais extensões.

F F F F
S1S1 P P
S1S1 R R
F F F F
P P P R

3 Tecnologias Utilizadas

Nesse capítulo, são discutidos conceitos importantes do Sistema Gerenciador de Banco de Dados PostgreSQL e da linguagem de programação Java.

3.1 PostgreSQL

O PostgreSQL é um Sistema Gerenciador de Banco de Dados Objeto-Relacional (SGBDOR). Devido à sua licença aberta, o PostgreSQL pode ser utilizado, modificado e distribuído por qualquer pessoa para qualquer finalidade, seja privada, comercial ou acadêmica, livre de encargos.

O *site* oficial PostgreSQL (2004) se define como:

> o PostgreSQL é derivado do pacote Postgres escrito na Universidade da Califórnia em Berkeley. O projeto POSTGRES, liderado pelo professor Michael Stonebraker, foi patrocinado pela DARPA (*Defense Advanced Research Projects Agency*), pelo ARO (*Army Research Office*), pela NSF (*National Science Foundation*) e pela ESL. A implementação do Postgres começou em 1986.

O PostgresSQL passou por várias versões desde então. A primeira versão de demonstração do sistema se tornou operacional em 1987, e foi exibida em 1988 na Conferência ACM-SIGMOD. O *Postgres* tem sido usado para implementar muitas aplicações diferentes de pesquisa e de produção, incluindo sistema de análise de dados financeiros, pacote de monitoramento de desempenho de turbina a jato, Banco de Dados de acompanhamento de asteróides, Banco de Dados de informações médicas, além de vários sistemas de informações geográficas. O *Postgres* também tem sido usado como ferramenta educacional por várias universidades.

O tamanho da comunidade de usuários externos praticamente dobrou durante o ano de 1993. Começou a ficar cada vez mais óbvio que a manutenção do código do protótipo e suporte estava consumindo grande parte do tempo que deveria ser dedicado às pesquisas de Banco de Dados. Em um esforço para reduzir esta sobrecarga de suporte, o projeto do *Postgres* de Berkeley terminou oficialmente com a Versão 4.2, segundo o *site* oficial do PostgreSQL (2004).

Andrew Yu e Jolly Chen (1994), adicionaram um interpretador de linguagem SQL ao *Postgres*. Com um novo nome, *Postgres95*, foi liberado na *Web* para encontrar seu próprio caminho no mundo como descendente de código aberto do código original do *Postgres* de Berkeley.

Em 1996 o *Postgres* recebeu o nome de PostgreSQL, para refletir o relacionamento entre o *Postgres* original e as versões mais recentes com capacidade SQL. Ao mesmo tempo, foi mudado o número da versão para começar em 6.0. Com o PostgreSQL a ênfase foi o aumento das funcionalidades e recursos, embora o trabalho continuasse em todas as áreas.

As principais melhorias no PostgreSQL incluem: o bloqueio no nível de tabela foi substituído por um sistema de concorrência multi-versão, que permite aos que estão lendo continuarem a ler dados consistentes durante a atividade de escrita, e permite efetuar cópias de segurança através do *pg_dump*[7] enquanto o Banco de Dados se mantém disponível para consultas; a implementação de funcionalidades importantes no Servidor, incluindo subconsultas, padrões, restrições e gatilhos; a incorporação de funcionalidades adicionais compatíveis com a Linguagem SQL92, incluindo chaves primárias; a velocidade geral do código do Servidor foi melhorada em aproximadamente 20 a 40%, e o tempo de inicialização do Servidor foi reduzido em 80% desde que a Versão 6.0 foi liberada.

[7] O *pg_dump* é um utilitário para salvar um Banco de Dados do PostgreSQL em um arquivo de script ou de exportação, que está mais detalhado no Apêndice B;

O PostgreSQL foi criado para a plataforma LINUX e uma empresa DBExpert gerou uma versão do PostgreSQL para a plataforma Windows; porém atualmente o PostgreSQL desenvolveu a Versão 8.0 para a Plataforma Windows.

O PostgreSQL é um sistema distribuído sob a licença BSD, o que torna o seu código fonte disponível e o seu uso livre para aplicações comerciais ou não (Almeida e Azevedo, (2003).

O SGBD PostgreSQL tem como grande vantagem possuir recursos comuns a Banco de Dados de grande porte, o que o deixa apto a trabalhar com operações de missão crítica. Além disso, trata-se de um Banco de Dados versátil, seguro e gratuito.

É possível utilizar o PostgreSQL em vários Sistemas Operacionais, dentre os quais o Windows, Linux. O PostgreSQL permite a criação de uma base de dados de tamanho infinito. Cada tabela pode ter até 16 TB (1 *terabyte* = 1024 *gigabytes*), sendo que cada linha pode ter até 1,6 TB e cada campo 1 GB. O Banco de Dados ainda conta com *triggers*, integridade referencial, entre outros recursos, além de ser compatível com uma série de linguagens, tais como PHP, C, Java, Perl.

Aliado a essa crescente procura, surge a necessidade de implementar técnicas de gerenciamento de *backup* e *recovery* mais elaboradas e seguras para o administrador do Banco de Dados.

3.2 Linguagem Java

A linguagem Java foi anunciada formalmente em maio de 1995, pela *Sun* em uma importante conferência, gerando muito interesse na comunidade comercial por causa do estrondoso interesse da *World Wide Web*, que passou a fazer uso de Java para criar páginas da

Web, com conteúdo interativo e dinâmico, para desenvolver aplicativos corporativos de grande escala, para aprimorar a funcionalidade de Servidores da *World Wide Web*, e fornecer aplicativos para dispositivos destinados ao consumidor final.

A Linguagem Java veio para facilitar uma abordagem disciplinada para o projeto de programas de computador. Cirne (1998), define a linguagem e a plataforma Java da seguinte maneira:

> A linguagem e plataforma Java vem adquirindo grande importância nos últimos anos, principalmente por ser interpretada e prover código móvel. Cada programa Java é compilado, gerando uma linguagem intermediária chamada *Java bytecodes*. As instruções em *bytecodes* podem rodar em qualquer máquina que possua o interpretador Java. Com isso, Java permite a comunicação de objetos em sistemas heterogêneos. A linguagem Java também é distribuída, multitarefa e possui uma ampla biblioteca de rotinas para lidar facilmente com protocolos TCP/IP como HTTP e FTP.

Geralmente, são formados por diversas partes: ambiente, linguagem, *interface* de programas aplicativos Java (*Applications Programming Interface* – API) e diversas bibliotecas de classes e apresentam cinco etapas para a sua execução, sendo eles: *edição*, *compilação, carga, verificação* e *execução*.

A primeira fase, ou seja, a *edição*, consiste em editar um arquivo, sendo realizado com um programa editor. Um programador digita um programa Java usando o editor e corrige o mesmo, caso seja necessário. Quando é especificado que o arquivo no editor deve ser salvo, esse programa fica armazenado em um dispositivo de armazenamento secundário, como um disco.

Na segunda fase, a *compilação*, o programador digita o comando *javac* para compilar o programa. Esse compilador traduz o programa Java para *bytecodes* (a linguagem entendida pelo interpretador Java).

Na terceira fase da execução de Java, *carga*, o programa deve ser colocado primeiramente na memória antes de ser executado, isso é realizado pelo carregador de classe, que pega arquivos com extensão *class* que contém os *bytecodes* e o transfere para a memória.

Esses arquivos *class* podem ser carregados a partir de um disco em seu sistema ou através de uma rede.

Existem dois tipos de programas onde o carregador de classe carrega arquivos com extensão *class*: aplicativos (programa parecido com um processador de texto, planilha, desenho, correio eletrônico, que é normalmente armazenado e executado a partir do computador local do usuário) e *applet* (programa pequeno que é armazenado em um computador remoto que usuários conectam via um navegador da *World Wide Web*. Eles são carregados no navegador a partir de um computador remoto, são executados no navegador e descartados quando termina a execução).

Na quarta fase, *verificação*, todos os dados (*bytecodes*) são checados pelo verificador de *bytecodes*, isso assegura que os *bytecodes* para classes que são carregadas a partir da *Internet* são válidos e não violam as restrições de segurança de Java (Deitel, 2001).

E na última fase, *execução*, o computador controlado pela sua CPU, interpreta o programa, um *bytecode* por vez, realizando assim a ação especificada pelo programa. Na Figura 3.1, é representado um ambiente típico de Java, mostrando essas cinco fases de execução vistas acima (Deitel, 2001).

A linguagem de programação Java é projetada para resolver vários problemas nas modernas práticas de programação. Ela provê excelentes ferramentas para programadores, facilitando a programação, pois é orientada a objetos. Devido a sua arquitetura neutra, aplicações em Java são ideais para ambientes como a *Internet* (Sun Microsystems, 1995).

Java apresenta duas formas básicas de ser utilizada numa aplicação: *Applet* e *Aplicação*.

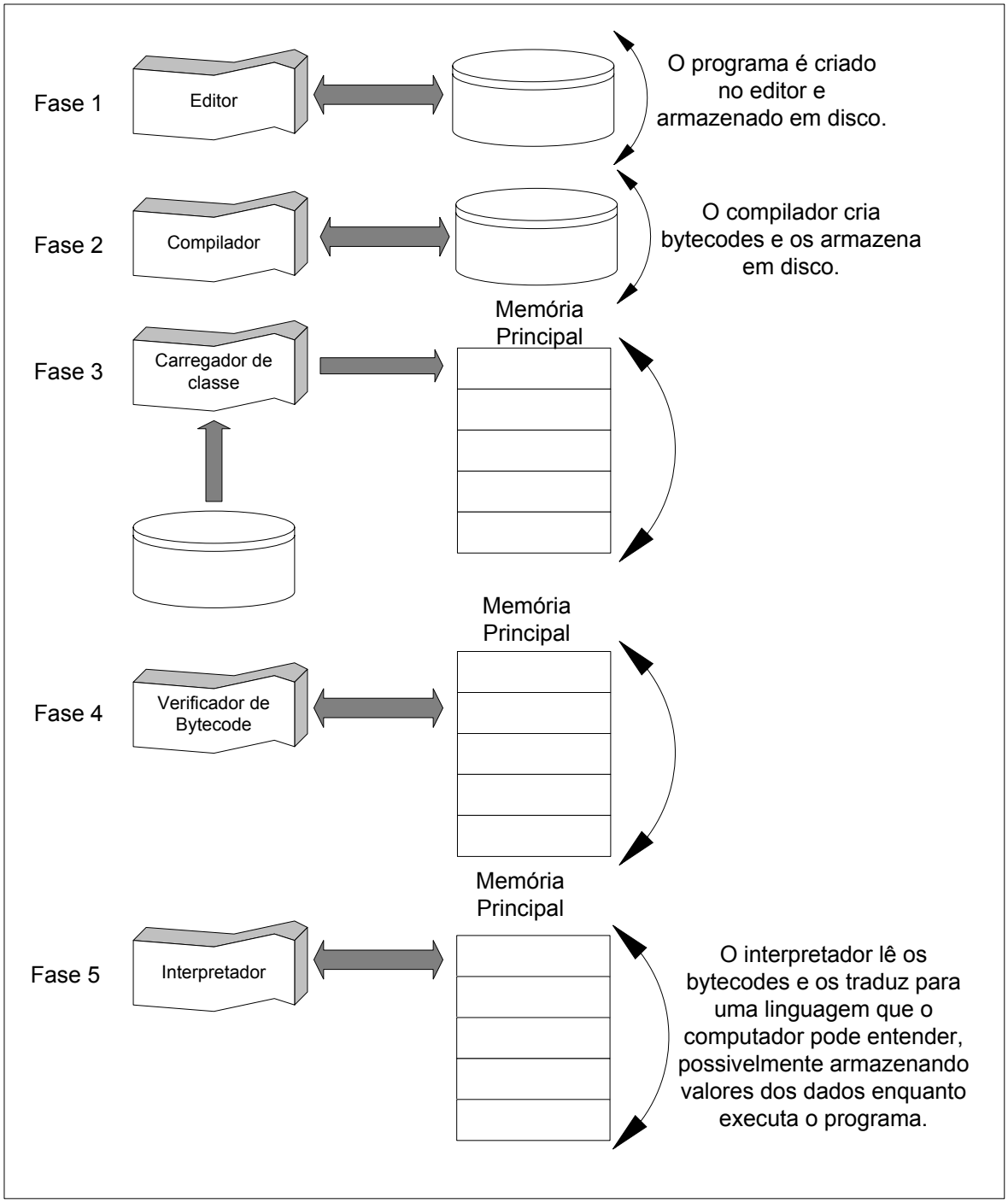

Figura 3.1: Exemplo de um Ambiente Java Típico,(Deitel, 2001).

Applets são como programas, mas são realizados por um *browser* que interpreta o código Java, ou seja, o código binário que está sendo transportado pela rede e será interpretado por um *browser* tal como *HotJava* ou *Netscape Navigator*.

Os *applets* são carregados a partir de um computador remoto no navegador, realizados no navegador e descartados ao ser finalizada a operação.

A Aplicação diferencia-se de um *Applet* por apresentar a função *main()*. A aplicação executa como qualquer outro programa tradicional podendo estar na máquina Servidora ou em uma máquina *stand-alone*.

Os aplicativos são carregados na memória e executados usando o interpretador *java*.

Os *applets*, as aplicações ou qualquer outro programa Java pode se conectar com uma base de dados, realizar comandos SQL e processar resultados.

A Java *API* tem uma imensa coleção de classes e métodos para a realização de cálculos matemáticos comuns, manipulações de *strings*, de caracteres, entrada/saída, checagem de erros e muitas outras operações úteis, fazendo com que o trabalho do programador fique mais fácil, pois esses métodos fornecem muitas capacidades que o programador precisa para trabalhar com eficiência (Nascimento e Fornari, 2003).

Além de ser integrada com a *Internet*, Java também é uma ótima linguagem para desenvolvimento de aplicações em geral, pois implementa o modelo de objetos e dá suporte ao desenvolvimento de *software* em larga escala.

Segundo Rios (1999), “para a elaboração da especificação desta ferramenta é necessária a definição de qual tecnologia será utilizada na construção do sistema de software e de como este sistema de software será construído”.

4 Revisão Bibliográfica

Nesse capítulo será apresentada uma Revisão Bibliográfica sobre os assuntos que são tratados nessa dissertação, fazendo um estudo mais aprofundado sobre os aspectos funcionais de um BD *Standby*, *logs*, tratamento de falhas e a alteração do fluxo de transação através da ferramenta SABaDO.

4.1 Aspectos Funcionais

A principal característica para se ter um ambiente de *backup* utilizando Banco de Dados *Standby* é possuir um Servidor de Banco de Dados de Produção, onde as transações estão sendo processadas e um outro Servidor de *Backup* possuindo a mesma estrutura física ou de *hardware* do Servidor de Produção. Nesse Servidor de *Backup* é necessário ter instalado o SGBD na mesma versão do SGBD do Banco de Dados de Produção. É necessário que esses Servidores estejam se conectando através de um meio de comunicação, podendo ser rede local ou *Internet*.

Havendo uma igualdade dos Servidores, será necessário realizar um *backup* inicial e completo do Banco de Dados de Produção, que será transferido para o Servidor de *Backup*. Nessa fase, os dois bancos estarão em sincronismo. Na medida em que o Banco de Dados de Produção passa a receber novas transações, começa o processo de desigualdade dos Servidores. Para manter esse sincronismo, o Banco de Dados de Produção deverá gerar arquivos de *log* ou WAL contendo as transações realizadas.

A Ferramenta SABaDO deverá atuar em três níveis distintos: na replicação de dados, no tratamento de falhas e na alteração do fluxo de transação.

4.2 Replicação de Dados

King (1991) fala em seu artigo sobre *logs* de transação como descrito a seguir:

> no Servidor Principal são geradas entradas de *logs* para todas as transações e depois de criados, são enviados para o Servidor de *Backup*. Quando uma transação é finalizada com o COMMIT no Banco de Dados Principal, é atribuído um *ticket*, ou seja, é gerado um número seqüencial que representa em ordem as transações que foram finalizadas.

Segundo King (1991), é preciso seguir algumas regras básicas de consistência de Banco de Dados para as transações. O mecanismo de processamento de transação no Banco de Dados Principal garante que sejam executados um conjunto de transações T seguindo suas dependências, ou seja, tendo T_1 e T_2 duas transações de um Banco de Dados, tal que T_1 é efetivada antes de T_2, diz-se que $T_1 \rightarrow T_2$ se as duas transações acessam dados em comum e pelo menos uma delas realiza algum tipo de modificação na informação. As dependências podem ser classificadas como:

a) write-write (W-W), ou seja, as duas transações alteram as informações;

b) write-read (W-R), ou seja, a primeira transação altera a informação e a segunda faz leitura; e,

c) read-write (R-W), ou seja, a primeira transação faz leitura e a segunda escrita. As leituras efetuadas no Banco de Dados Principal não são tratadas no Banco de Dados de *Backup*, somente as alterações são espelhadas no Banco Secundário.

Para que a transferência dos *logs* de transações seja coerente, é necessário seguir algumas exigências, tais como:

a) Atomicidade: ou seja, se uma ação de uma transação T for executada no Banco de Dados Principal, então, todas as ações de T deverão estar no *log* de transação, fazendo com que não exista transação parcial (King, 1991);

b) Consistência Mútua: assumindo que T_1 e T_2 pertencem a T, então $T_1 \rightarrow T_2$ são processados no Banco de Dados Principal; essas transações devem estar no Banco de *Backup*, isso garante que o *backup* seja equivalente ao Banco de Dados Principal (King, 1991).

Para exemplificar, diz-se que T_1 é uma transação que faz uma venda de uma passagem aérea para um cliente, essa transação grava um registro contendo o nome, data, número do vôo e informações sobre o pagamento. A Transação T_2 verifica os passageiros de um determinado vôo, alterando o atributo que diz se o passageiro está no vôo. A Transação T_2 não pode ser executada no Servidor de *Backup* sem que a Transação T_1 seja executada, caso contrário não haverá registro para ser alterado;

c) Consistência Local, as dependências *read-write* não causam violações de consistências de *log*. Caso T_1 e T_2 tenham dependências, onde T_1 executa uma leitura e T_2 executa uma alteração, os valores consultados por T_1 não precisam ser enviados para o Servidor de *Backup.* Mas somente os valores alterados por T_2 (King, 1991).

A idéia básica de King (1991) é reproduzir as ações realizadas no Banco de Dados Principal para o Banco de Dados de *Backup* utilizando um *log*. O *log* poderá ter informações de *undo*, ou seja, informações para desfazer a transação e *redo*, ou seja, as transações propriamente ditas. Para evitar o tráfego de rede o método de King (1991) não trata de informações de *undo*.

Cada arquivo de *redo log* deverá conter informações suficientes para garantir que as transações sejam aplicadas no Banco de Dados de *Backup*. Esses arquivos de *logs* deverão ter a seguinte estrutura: *log* → *S, H, T, act, tbl, key, val*, sendo que *S* é o meio de armazenamento da operação; *H* é o Servidor que controla a transação; *T* é o identificador da transação; *act* é o comando da transação; *tbl* é o nome da tabela da transação; *key* é o identificador do registro (código) para as transações que sofreu alteração; *val* é a imagem depois que o registro foi alterado. Nem todas as transações terão todas as informações de *log*, por exemplo, para um

comando DELETE não é necessário ter a entrada de *val* e quando for pertinente o uso do comando CREATE TABLE não é necessário ter a informação de *key*.

Molina (1990) comenta em seu artigo a respeito de dois tipos particulares de *backup* remoto, sendo *1-safe* e *2-safe*. No modelo *1-safe* as transações efetivadas no Banco de Dados Principal são propagadas para o Servidor de *Backup* e aplicadas. Isso significa que, em caso de desastres, algumas transações que foram executadas e não foram geradas *logs* de transação serão perdidas.

No modelo *2-safe*, o Servidor Principal e o de *Backup* executam *commit* em duas fases, para certificar que as transações serão aplicadas em ambos os Servidores. Essa opção degrada a *performance* do Banco de Dados Principal, pois para cada transação efetivada é necessário esperar um retorno do Servidor de *Backup* sinalizando que foi executado com sucesso.

A Ferramenta SABaDO terá a tarefa de realizar uma cópia dos arquivos WAL para um local seguro e para um lugar no Banco de Dados de *Backup*, sendo feito a aplicação desses arquivos, com o propósito de manter sincronismo com o Servidor de Produção. Através da Ferramenta SABaDO, será possível realizar consultas sobre os *logs* gerados no Banco de Dados de Produção, seu arquivamento (*backup*) e sua transferência para o Banco de Dados de *Backup*. Será utilizada uma estrutura de fila para controlar os *logs*, como pode ser observado na Figura 4.1:

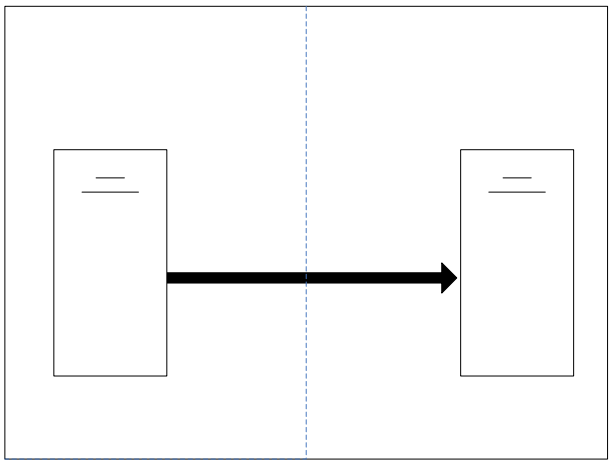

Figura 4.1: Exemplo de Controle de Transferências de WAL *(Fila).*

De acordo com a Figura 4.1, os arquivos WAL são organizados em uma estrutura de filas do mais antigo para o mais recente. A Ferramenta SABaDO fará a transferência obedecendo essa fila, ou seja, não é possível transferir o arquivo Log_2 sem antes transferir o arquivo Log_1.

A transferência dos arquivos WAL será feita obedecendo uma periodicidade informada pelo administrador do Banco de Dados, podendo ser alterado a qualquer momento.

4.3 Tratamento de Falhas

Uma aplicação, independente de qual linguagem de programação foi desenvolvida, realiza transações no Banco de Dados. Essas transações têm a garantia de que se a transação tenha sido finalizada, suas informações são garantidas contra qualquer tipo de falha.

Analisando a Figura 4.2 tem-se um exemplo, cuja transação foi iniciada às 12h10min e finalizada às 12h25min. Às 12h50min ocorreu uma falha no disco. Nessa transação foi gerado um arquivo de *log* e em seguida aplicado no Banco de Dados S*tandby*, quando esse estiver disponível para os usuários, essa transação será de domínio público.

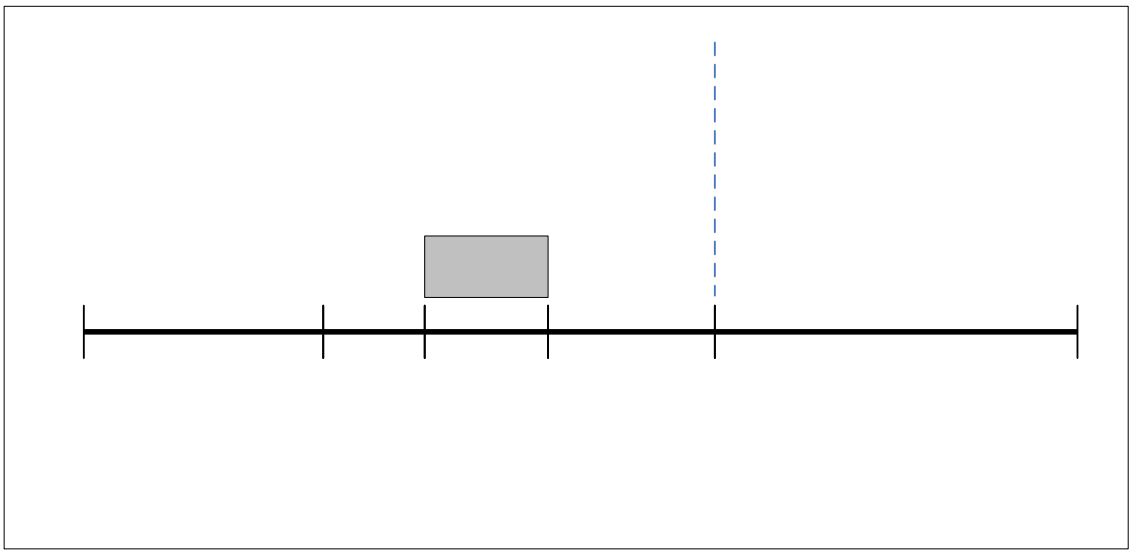

Figura 4.2: Exemplo de Representação de um Cenário de Falha

Analisando a Figura 4.3, é possível observar uma situação diferente: a transação é iniciada às 12h40min, às 12h50min ocorreu uma falha no Servidor e a transação não havia sido finalizada, ou seja, seguindo a regra de atomicidade de transação, não é gerado um *log* dessa transação. Em conseqüência, essa transação não vai para o Banco de Dados S*tandby*. Quando o Banco de Dados S*tandby* entrar em produção, essa transação que não havia sido finalizada deverá ser executada novamente pelo usuário.

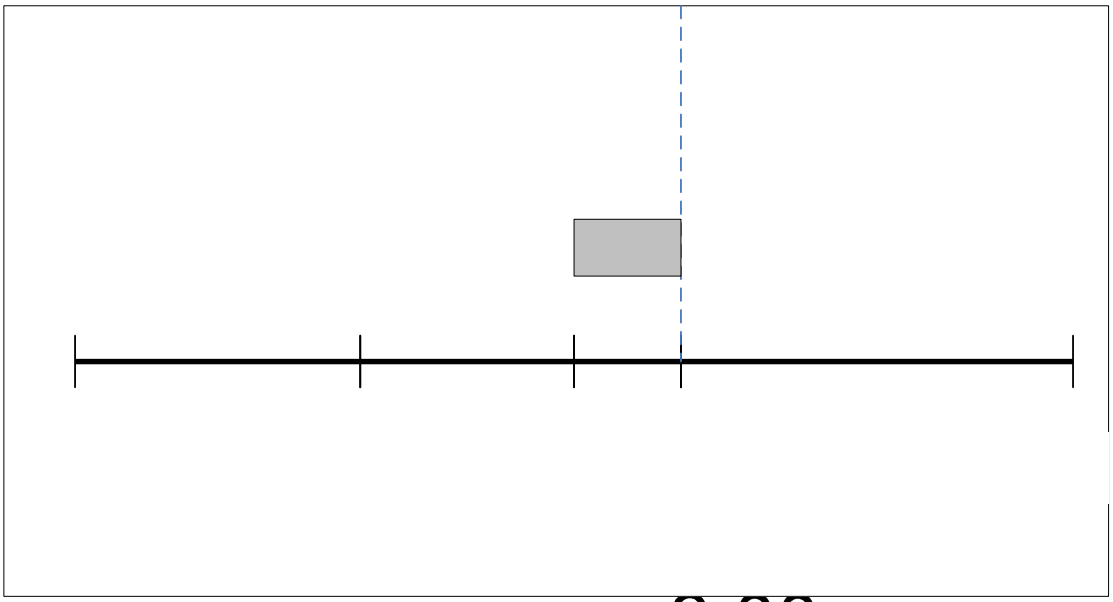

Figura 4.3: Exemplo de Representação de um Cenário de Falha com Perda de Transação

3:00

Backup completo do
Banco de Dados

Conforme ilustrado na Figura 4.2, houve a falha na Base de Dados às 12h50min e, após a detecção da falha, será necessário voltar *backup*, e o último *backup* realizado tenha sido feito às 3hs e não há o recurso de Banco de Dados S*tandby*. Após a volta do *backup* a transação que teve início às 12h10min e finalizada às 12h25min será perdida e deverá ser entrada novamente pelo usuário responsável, pelo fato de não ter o recurso de *log* ativado. A propósito, todas as transações geradas entre o período de 3hs e 12h50min deverão ser entradas novamente.

Caso o Banco de Dados S*tandby* tenha sido ativado, será necessária a volta integral do *backup*, deixando o Servidor de Banco de Dados na posição das 3hs, e em seguida, será necessário aplicar os *logs* gerados das 3hs às 12h50min. Nessa situação, a transação que teve início às 12h10min e finalizada às 12h25min não precisará ser entrada novamente pelo usuário.

A situação que ocorre na Figura 4.3 é um pouco diferente, pois houve a falha às 12h50min e ficou uma transação pendente, ou seja, ela não foi finalizada com sucesso até o momento da falha. Nessa transação não foi gerado um *log* e conseqüentemente não será possível restaurá-la através da volta do *backup*, ou seja, após a volta do *backup*, o usuário obrigatoriamente deve entrar com a transação novamente.

4.4 Alteração do Fluxo de Transação

A declaração de um desastre deve ser feita por um administrador de Banco de Dados. Isso é feito porque é muito difícil para o Banco de Dados de *Backup* distinguir quando o Servidor Principal sofreu um desastre. Os terminais dos usuários poderão estar conectados aos dois Servidores, e a conexão com o *Standby* permanece inutilizada, e acionada no momento da falha do Servidor Principal.

O modelo de falha para essa dissertação somente considera desastres no Servidor Principal, ou seja, para esse modelo, o Servidor de *Backup* nunca falha. Durante o processamento normal, esse Servidor recebe e processa *logs* do Servidor Principal. Quando um desastre é declarado, o Servidor de *Backup* termina a aplicação de *logs* disponíveis e então se torna o Servidor Principal e continua o processamento de transações.

Segundo King (1991), um Servidor de Banco de Dados pode estar em três modos diferentes: produção, *backup* ou em recuperação. Nas situações normais de operações, o Banco de Dados pode receber processamento de transações e esse gera *logs* para o Servidor de *Backup*. O modo de recuperação fica aplicando os *logs* gerados no Servidor Primário para manter consistência entre o Banco de Dados de Produção e de *Backup*. Quando ocorre um desastre no Servidor Principal, o Servidor que estava até o momento em *backup* inicia a operação e se torna o Banco de Dados Principal, passando a receber todas as transações.

Após a declaração do desastre, o Banco de Dados de *Backup* deve receber os *logs* ou os últimos *logs* que foram gerados pelo Servidor Principal, e aplicá-los. Terminada essa etapa, esse Banco de Dados está disponível para receber processamento de transações. Para os terminais que não tenham o Servidor de *Backup* configurado é preciso fazer essa configuração, ou seja, informar que o Servidor que estava em produção não é mais o mesmo.

Na Figura 4.4 são mostrados os passos que devem ser executados para que um Servidor de *Backup* possa se tornar um Servidor Principal de Banco de Dados. Na Etapa 1, depois de ter diagnosticado a falha, é feita a aplicação do *backup* físico do Banco de Dados, caso ainda não tenha sido feito nenhum.

Esse *backup* físico vai deixar o Banco de Dados válido em um determinado tempo, por exemplo, foi feito um *backup* físico do Banco de Dados de Produção às 8hs e a falha ocorreu às 12h15min, quando é voltado esse *backup*, tem-se a situação do banco às 8hs. Das 8hs até o momento da falha não está no *backup* físico, mas sim em arquivos de *log*. A Etapa 2

tem a função de aplicar os *logs* gerados das 8hs até o momento da falha. Realizada a restauração dos *logs*, o Banco de Dados está recuperado até o momento da falha e disponível para produção.

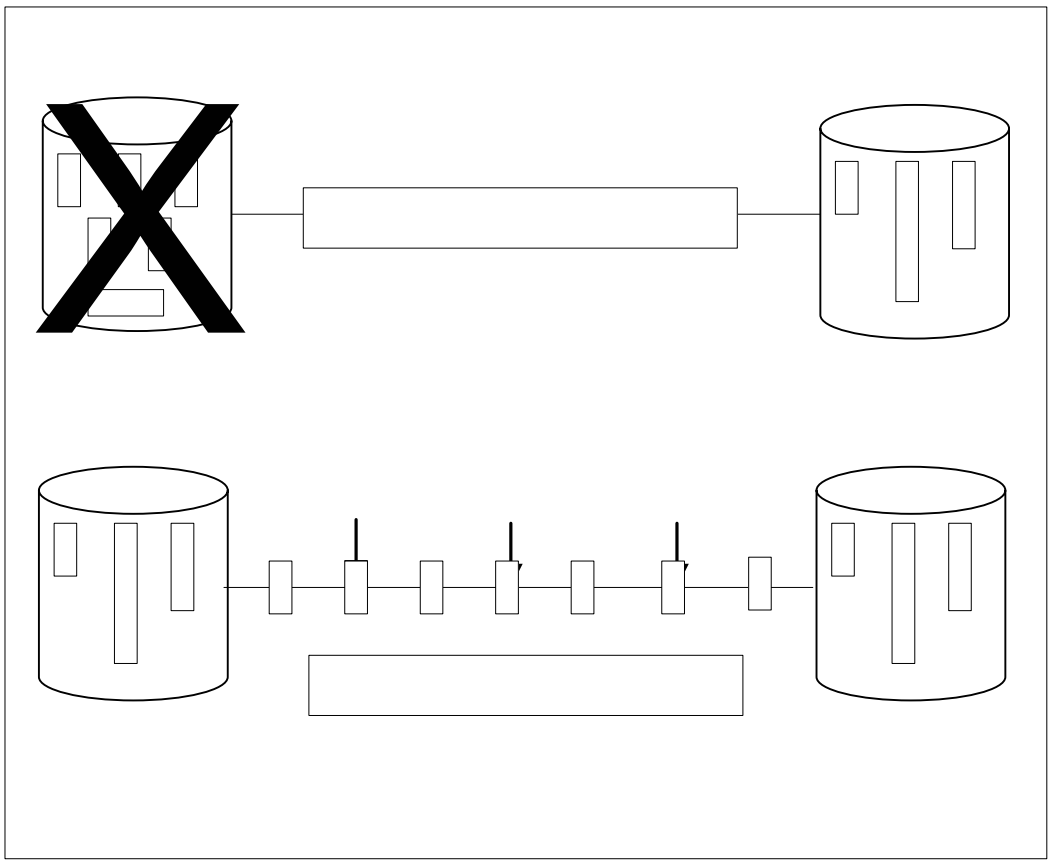

Figura 4.4: Fase de Recuperação de Banco de Dados

O cenário da Figura 4.4 representa uma situação completa de uma restauração a partir de um *backup* válido. O responsável pelo *backup* deve realizar *backups* físicos constantemente e, além disso, deve guardar os *logs* gerados em um local seguro, sendo uma opção excelente, o Banco de Dados *Standby*, onde esses *logs* serão aplicados.

Após a volta do *backup* é necessária a aplicação dos *logs* desde a volta do *backup* até o momento da falha, para deixar o Banco de Dados operante com o mínimo de perda possível, isto é, se uma transação tiver sido finalizada antes da falha, essa transação será mostrada depois que o Banco de Dados *Standby* entrar em produção.

5 FERRAMENTA SABaDO

O papel de uma política de *backup* apropriada e bem definida, nos dias atuais, é de grande importância para qualquer sistema computacional. Quando sistemas computacionais são indispensáveis para gerenciar áreas críticas, não se pode permitir que perda de dados coloque em risco vidas humanas, patrimônios ou investimentos.

SABaDO (*Standby Application of Backup on Database Operation*) é uma ferramenta escrita em Java para gerenciar o *backup* de maneira consistente do Banco de Dados. Essa ferramenta foi desenvolvida para atender a diversos requisitos de *backup*, tais como: exportação lógica do Banco de Dados, *backup full* da Base de Dados, *backup* e aplicação dos arquivos de transações.

5.1 Estrutura Funcional

Durante o tempo em que o SGBD PostgreSQL está recebendo transações, ele mantém um registro de escrita prévia, WAL *(write ahead log)*. O WAL contém todas as alterações realizadas nos arquivos de dados do Banco de Dados.

O WAL existe, principalmente, para fornecer segurança contra quedas, ou seja, caso haja alguma falha de *hardware* ou *software*, o Banco de Dados pode estar em um estado inconsistente, sendo necessário refazer as entradas gravadas desde o último ponto de controle.

Vários benefícios são identificados com a utilização dessa estrutura de arquivo de *log*, sendo elas:

a) Como pode ser reunida uma seqüência grande de arquivos de WAL para serem refeitos, pode ser obtida uma cópia de segurança contínua simplesmente realizando cópias dos

arquivos de segmento do WAL. Esse procedimento é útil para Bancos de Dados grandes, podendo não ser conveniente fazer cópias de segurança completas regularmente;

b) Essa técnica suporta a recuperação para um determinado ponto no tempo, ou seja, é possível restaurar um Banco de Dados voltando esse Banco para o estado em que se encontrava a qualquer instante posterior ao da realização da cópia de segurança base;

c) Se outro Servidor, carregado com a mesma cópia de segurança base do Banco de Dados, for alimentada continuamente com os arquivos de segmentos do WAL, será criado um sistema para reserva de *backup*, sendo o *backup standby*. A qualquer instante este outro Servidor pode ser ativado e se tornar servidor de Banco de Dados de Produção.

Um sistema carregado gera vários *megabytes* de transações para o WAL que precisam ser guardados. Portanto, o local de armazenamento desses *logs* deve possuir espaço suficiente para armazenar esses arquivos, sendo representado por Disco 2 e Disco 3 do Sistema de Produção na Figura 5.1. O Servidor de *Backup* também terá que ter espaço suficiente para receber esses arquivos de *log*, para que possa ser feita a aplicação para o Banco de Dados de B*ackup.*

Para que o Banco de Dados S*tandby* possa estar atualizado em relação ao Banco de Dados de Produção, é necessário ter todos os arquivos de WAL. Caso seja perdido algum arquivo, o Banco de Dados não conseguirá ser reconstruído pela falta de transações que se encontravam no arquivo de *log* perdido.

Na Figura 5.1 é exemplificado o fluxo completo de um sistema de *backup* utilizando Banco de Dados S*tandby*.

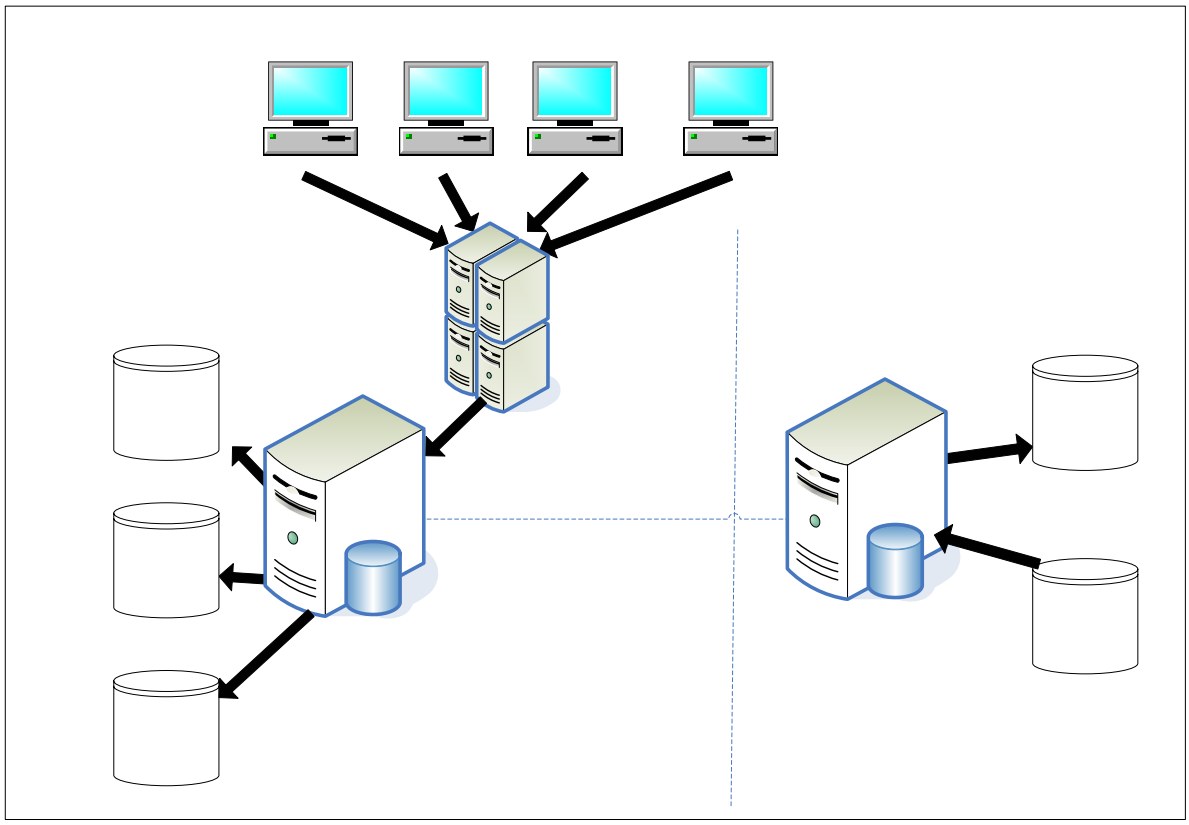

Figura 5.1: Exemplo de Cenário de um Banco de Dados Standby.

Na Figura 5.1 tem-se estações de trabalho acessando um Servidor de Aplicação (S_A) que faz acesso ao Servidor de Banco de Dados (S_B). Esse Servidor está recebendo inúmeras transações que estão sendo armazenadas no Disco 1. Para essas transações, são gerados arquivos WAL que serão gravados no Disco 2 e Disco 3. Esses arquivos serão copiados para o Disco 2 do Servidor de *Backup* (S_B) e serão aplicados, garantindo a consistência de dados entre os Servidores. O Disco 3 existe para a realização de cópia de segurança dos Arquivos WAL armazenados no Disco 2, para prevenir possível falha no Disco 2.

Havendo falhas no Servidor de Produção (S_A), o Servidor de *Backup* (S_B) assume o papel do Servidor Principal, como pode ser visto na Figura 5.2:

Disco 1 (dados e índices)

2Tb

1.2Tb ----------

Disco 2 (WAL)

2Tb

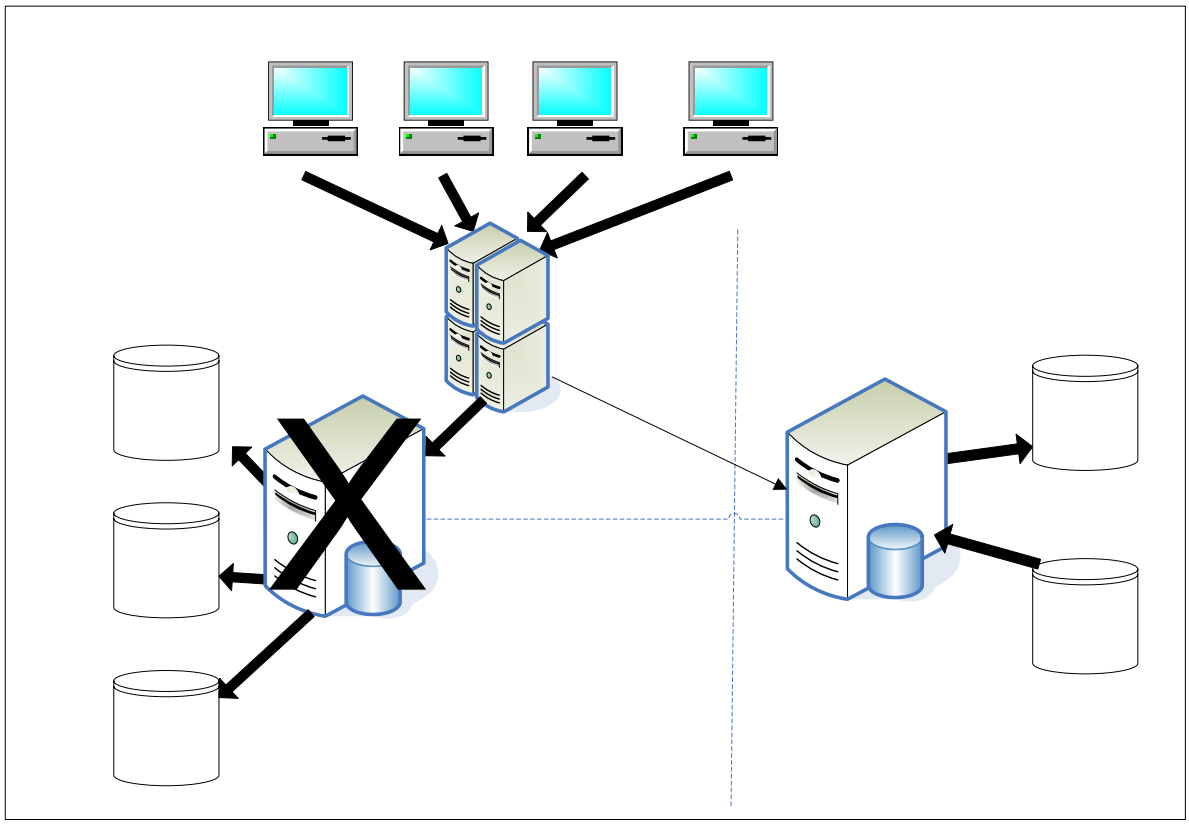

Figura 5.2: Exemplo de Cenário de um Banco de Dados Standby com Falha.

Na Figura 5.2 o Servidor de *Backup* (S_B) assume como sendo o Servidor de Produção e o Servidor de Aplicação direciona as transações para o novo Servidor de Banco de Dados, que passa a gerar os arquivos de WAL para garantir o armazenamento das transações desse novo Servidor de Produção. Para esse estudo, são considerados somente desastres no Servidor de Produção, ou seja, o Servidor de *Backup* nunca falha.

5.2 Arquitetura da Ferramenta SABaDO

A arquitetura da ferramenta desenvolvida neste trabalho e cujo esquema é apresentado na Figura 5.3, é composta por três componentes: a) *Exportar*, esse componente realiza um *backup* lógico do Banco de Dados; b) *Backup Full*, que gera um *backup* físico

Disco 1 (dados e índices)

2Tb

1,2Tb

Disco 2 (WAL)

2Tb

completo do Banco de Dados; c) *Backup de Log*, que realiza a transferência e aplicação dos *logs* de transações do Servidor Principal para o Servidor de *Backup*.

No diagrama da Figura 5.3 é possível verificar que o componente *Exportar* e *Backup Full* interagem com o Banco de Dados Principal e o componente *Backup* de *Log* interage somente com o Banco de Dados *Standby*.

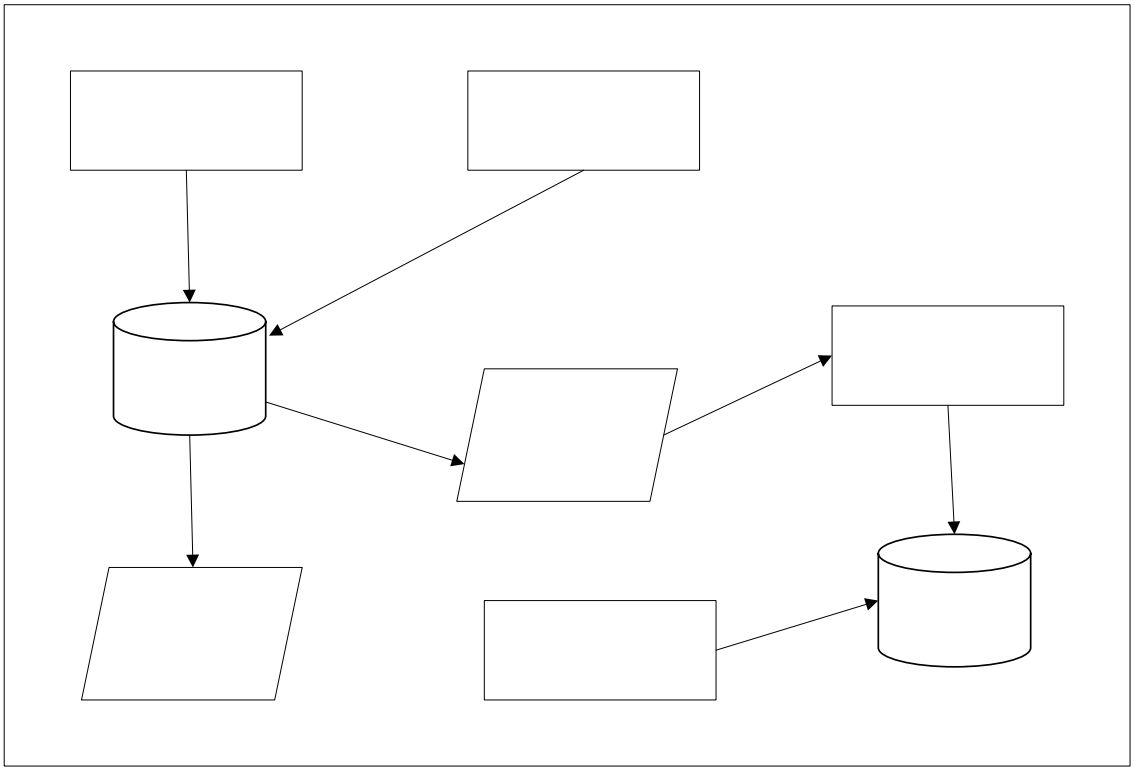

Figura 5.3: Arquitetura da Ferramenta SABaDO

O componente *Exportar* realiza um *backup* lógico do Banco de Dados baseado na linha de comando que é fornecida para ele, seguindo as regras de exportação do Banco de Dados PostgreSQL que pode ser obtida detalhadamente no Apêndice B dessa dissertação. A exportação de dados do Banco de Dados é feita utilizando o comando **pg_dumpall** que gera uma saída textual sobre o conteúdo do Banco de Dados, como definições de usuários, *triggers*, tabelas, dados e índices. A saída é direcionada para Ferramenta SABaDO para a tela do usuário e para um arquivo texto.

O componente *Backup Full* realiza um *backup* completo do Banco de Dados PostgreSQL. Esse componente é executado em uma periodicidade diária e é baseado em um horário que é informado para a Ferramenta. Outra informação importante a fornecer para esse componente é a localização dos arquivos de dados do PostgreSQL, para que sejam copiados para um Servidor ou dispositivo de armazenamento de *backup*. Esse componente altera o *status* do Banco de Dados para *begin backup,* ou seja, é alterado o *status* do Banco de Dados para cópia dos arquivos físicos sem ter que deixar inoperantes os usuários que estão conectados no momento do *backup*. Após a cópia de todos os arquivos do Banco de Dados, o componente retorna o *status* do Banco de Dados para normal, ou seja, executa um *end backup* no Banco de Dados.

Além da opção de ativação do *Backup Full* manualmente, existe também um temporizador, disparado no início da execução da Ferramenta SABaDO, que ativa o *backup* completo em períodos fixos de 24 horas. Logo no início da execução, o temporizador é configurado para disparar o *backup* completo no horário fornecido pelo administrador do Banco de Dados.

O componente *Backup Log*, baseado em uma periodicidade definida pelo administrador do Banco de Dados, realiza a cópia dos arquivos de *log* para um Servidor de *Backup* e, em seguida, para o Servidor *Standby* e já submete esse arquivo no Servidor *Standby*, fazendo o sincronismo de transações entre o Servidor S*tandby* e o Servidor de Produção.

O *Backup Log* move todos os arquivos WAL do Servidor Principal para o diretório de armazenamento especificado na Ferramenta SABaDO. A atualização do Servidor *Standby* é feita através de cópia dos arquivos de dados durante o último *Backup* completo mais os arquivos WAL e o arquivo **recovery.conf**. Antes de iniciar a atualização do Servidor *Standby*,

a Ferramenta SABaDO realiza um *shutdown* no Servidor *Standby*, apaga todos os arquivos de dados e copia os novos arquivos e realiza um *start* no Banco de Dados *Standby*.

O componente *Backup Log* só é ativado pelo temporizador, que é configurado no início da execução da Ferramenta SABaDO. Esse temporizador é responsável pelo agendamento periódico do *Backup* de WAL, com intervalos definidos pelo administrador do Banco de Dados.

O *Backup Log*, quando ativado, verifica se *o Backup Full* está sendo executado. Caso esteja, seu processo é cancelado e deixado para o período seguinte do agendamento, para evitar que os dois processos sejam executados ao mesmo tempo.

5.3 Caso de Uso

A Ferramenta SABaDO pode ser utilizada em qualquer plataforma que suporte a JVM[8] padrão. Foi implementada para facilitar a criação de *backups* válidos no SGBD PostgreSQL. Foi desenvolvida em módulos para facilitar a administração e os diferentes tipos de *backups* existentes para o Banco de Dados.

A funcionalidade do módulo Exportar está representado na Figura 5.4. Nessa fase da Ferramenta, é feita uma exportação do Banco de Dados inteiro, e sendo armazenado em um arquivo do sistema operacional. Esse processo deve ser realizado no Banco de Dados de Produção.

[8] Java Virtual Machine (JVM) é um mecanismo que permite executar código em Java em qualquer plataforma ou Sistema Operacional

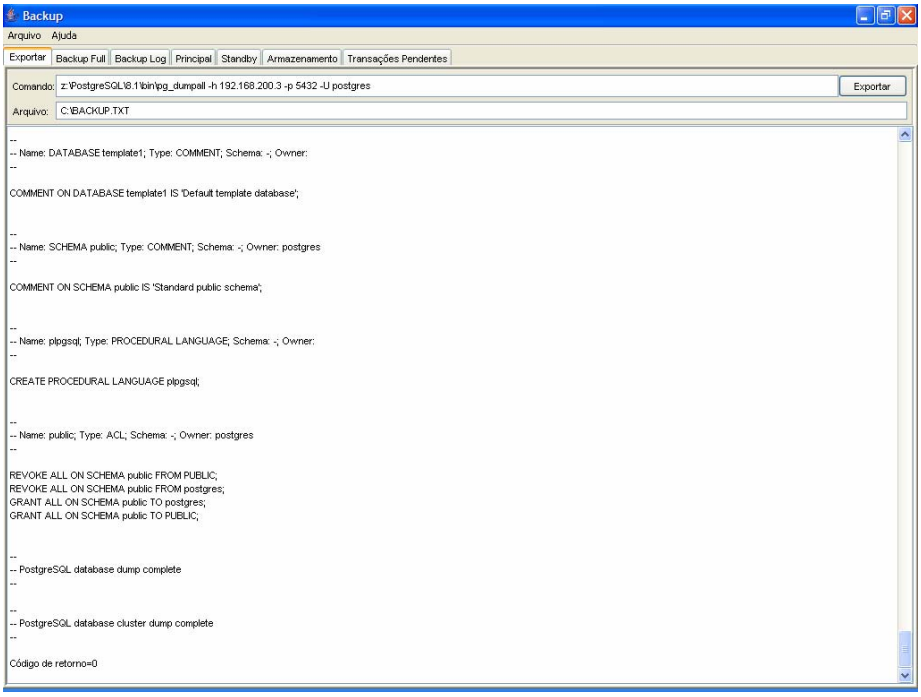

Figura 5.4: Módulo de Exportação da Ferramenta SABaDO

A Ferramenta SABaDO utiliza o comando *pg_dumpall*[9] usado para extrair um agrupamento de Bancos de Dados do PostgreSQL para um arquivo texto. Nesse arquivo, estão todos os comandos SQL que podem ser usados como entrada do *psql*[10] para restaurar os Bancos de Dados. Opcionalmente, o comando *pg_dumpall* pode ser substituído pelo comando *pg_dump*.

O *pg_dumpall* salva as informações necessárias para recriar todos os tipos, funções, tabelas, índices, agregações e operadores definidos pelos usuários. Adicionalmente, todos os dados são salvos no formato texto para poderem ser prontamente importados.

Antes de realizar a importação do Banco de Dados, alguns cuidados devem ser tomados, como verificação de *hardware* e localizações de *tablespaces*. O Servidor de *Backup* deverá ter os mesmos recursos de *hardware* do que o Servidor de Produção, como memória principal e discos. Além disso, toda a estrutura de diretórios deverá estar criada no Servidor de *Backup* para que possa realizar a importação do Banco de Dados.

[9] Comando utilizado para realizar exportação do Banco de Dados. Suas variações estão no Apêndice B
[10] Aplicativo do PostgreSQL. Suas variações estão no Apêndice B

O processo de construção do Banco de Dados *Standby* inicia com a configuração do Servidor de Banco de Dados Principal, conforme ilustra a Figura 5.5:

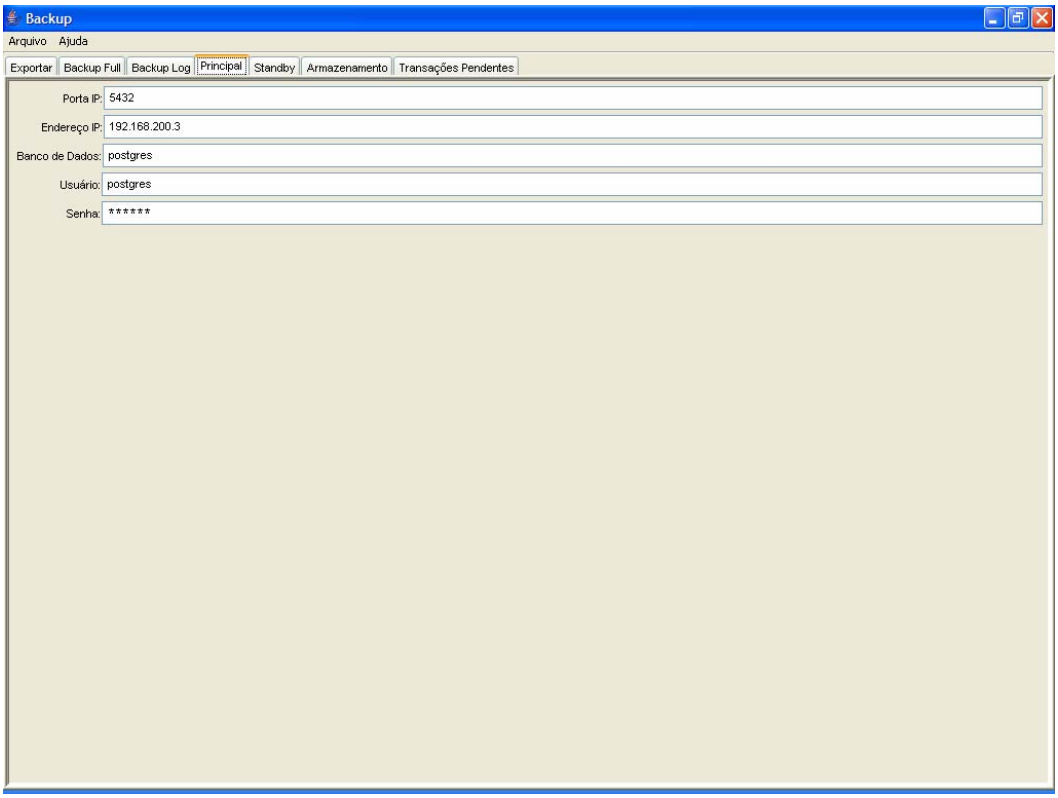

Figura 5.5: Módulo Principal da Ferramenta SABaDO.

A Ferramenta SABaDO deve ser executada no Servidor *Standby* para evitar processamento desnecessário no Servidor Principal. Para que a Ferramenta possa fazer conexão ao Banco de Dados de Produção é necessário fornecer algumas informações importantes como a Porta IP onde o Banco de Dados está aguardando conexão, o endereço IP do Servidor Principal, onde o Banco de Dados de Produção está instalado, o nome do usuário com privilégio de *Superusuário*, ou seja, usuário que possui todos os privilégios do Banco de Dados e a Senha desse usuário Administrador do Banco de Dados.

O processo de construção do Banco de Dados *Standby* é iniciado na Funcionalidade *Backup Full*, conforme Figura 5.6.

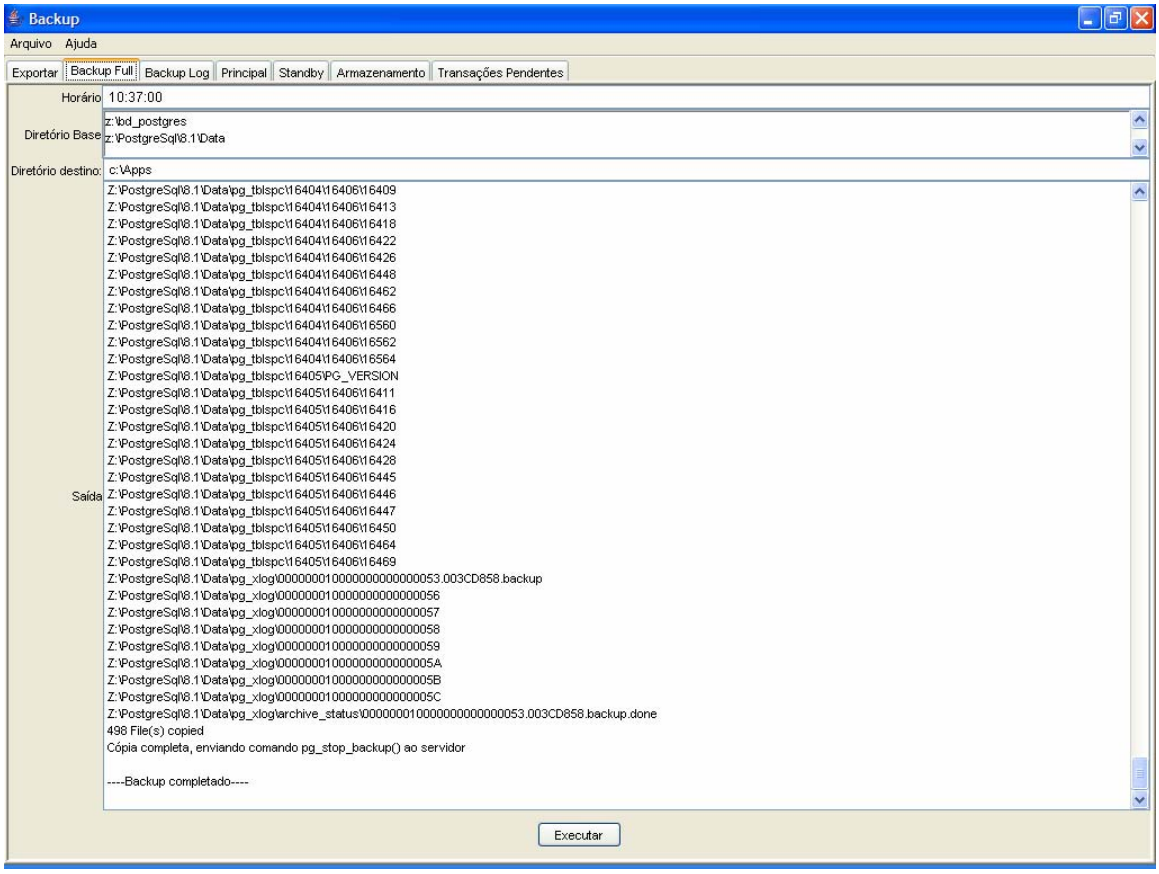

Figura 5.6: Módulo Backup Full da Ferramenta SABaDO

O procedimento para fazer a cópia de segurança base envolve alguns passos importantes, conforme Figura 5.7:

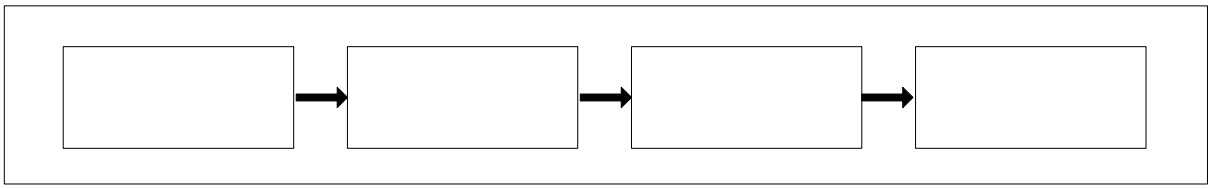

Figura 5.7: Exemplo de Procedimentos de Cópia de Segurança Base.

1) Garantir que a cópia dos arquivos de segmento do WAL esteja habilitada e funcionando. Em um sentido abstrato, a execução do sistema PostgreSQL produz uma seqüência indefinidamente longa de entradas no WAL. O sistema divide fisicamente esta seqüência em *arquivos de segmento* do WAL, normalmente com 16 MB cada. São atribuídos

nomes numéricos aos arquivos de segmento para refletir sua posição na seqüência abstrata do WAL.

Quando não é feita cópia dos arquivos de segmento do WAL, normalmente o sistema cria apenas uns poucos arquivos de segmento e, depois, recicla-os renomeando os arquivos que não são mais de interesse com número de segmento mais alto. Assume-se não existir mais interesse em um arquivo de segmento cujo conteúdo preceda o ponto de controle anterior ao último, podendo, portanto, ser reciclado.

Quando é feita a cópia dos arquivos de segmento do WAL, deseja-se capturar o conteúdo de cada arquivo quando este é completado, guardando os dados em algum lugar antes do arquivo de segmento ser reciclado para que possa ser reutilizado.

O PostgreSQL deixa o administrador escolher o comando a ser executado para copiar o arquivo de segmento completado para o local de destino. O comando a ser executado é especificado através do parâmetro de configuração *archive_command* que, na prática, é sempre colocado no arquivo **postgresql.conf**[11]. É importante que o comando para realizar a cópia retorne o *status* de saída zero se, e somente se, for bem-sucedido. Ao receber o resultado zero, o PostgreSQL assume que a cópia do arquivo de segmento do WAL foi bem-sucedida, e remove ou recicla o arquivo de segmento. Entretanto, um *status* diferente de zero informa ao PostgreSQL que o arquivo não foi copiado, serão feitas tentativas periódicas até ser bem-sucedida.

Havendo preocupação em poder recuperar até o presente instante, devem ser efetuados passos adicionais para garantir que o arquivo de segmento do WAL corrente, parcialmente preenchido, também seja copiado para algum lugar. Isto é particularmente importante no caso do Servidor gerar pouco tráfego para o WAL ou tiver períodos ociosos onde isto acontece, uma vez que pode levar muito tempo até que o arquivo de segmento fique totalmente preenchido e pronto para ser copiado. Então, a combinação dos arquivos de

segmento do WAL guardados, com o arquivo de segmento do WAL corrente guardado, será suficiente para garantir que o Banco de Dados pode ser restaurado até um minuto, ou menos, antes do presente instante.

Atualmente este comportamento não está presente no PostgreSQL porque não se deseja complicar a definição de *archive_command* requerendo que este acompanhe cópias bem-sucedidas, mas diferentes, do mesmo arquivo do WAL. O *archive_command* é chamado apenas para segmentos do WAL completados.

Deve ser lembrado que embora a cópia do WAL permita restaurar toda modificação feita nos dados dos Bancos de Dados do PostgreSQL, não restaura alterações feitas nos arquivos de configuração, ou seja, os arquivos **postgresql.conf**, **pg_hba.conf** e **pg_ident.conf**, uma vez que estes arquivos são editados manualmente. Aconselha-se a manter os arquivos de configuração em um local onde são feitas cópias de segurança regulares do sistema de arquivos.

2) Conectar-se ao Banco de Dados como um *superusuário* e executar o comando SELECT *pg_start_backup('rótulo')*, sendo que o rótulo pode ser qualquer cadeia de caracteres que se deseje usar para identificar unicamente esta operação de cópia de segurança. A função *pg_start_backup* cria o arquivo *rótulo da cópia de segurança*, chamado *backup_label*, com informações sobre a cópia de segurança, no diretório do agrupamento.

3) Realizar a cópia de segurança utilizando a Ferramenta SABaDO. Não é necessário nem desejado, parar a operação normal do Banco de Dados enquanto a cópia é feita. Essa cópia de segurança é feita uma vez ao dia no horário informado no seu respectivo campo, conforme Figura 5.8. O atributo Diretório Base informa os caminhos possíveis para criação dos arquivos do Banco de Dados de Produção, visto que este pode estar distribuído em mais de um local de armazenamento. É interessante guardar esse *backup* completo do Banco de Dados em um Servidor de *Backup* auxiliar ou em algum dispositivo de *Backup* permanente,

[11] Arquivo de configuração do PostgreSQL. Seu conteúdo está detalhado no Apêndice B

como uma fita *Dat*, por exemplo. O Servidor de *Backup* Auxiliar de armazenamento desse *backup full* deve ser informado na opção Armazenamento, conforme a Figura 5.8:

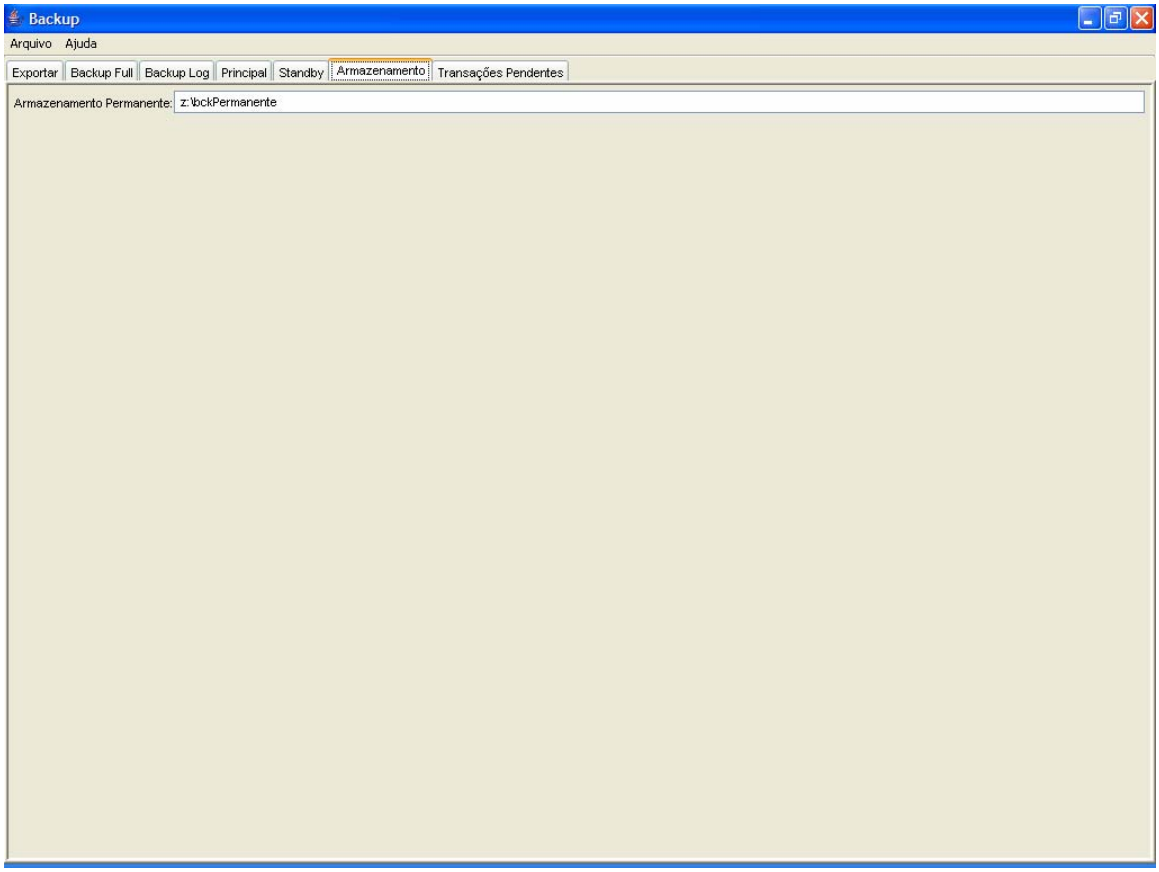

Figura 5.8: Módulo de Armazenamento da Ferramenta SABaDO

Sempre será criada uma estrutura de diretório contendo a data, hora e minuto do início do *backup*. Esses arquivos de *backup* são de extrema importância, pois guardam a situação do Banco de Dados no dia do *Backup*, para fins de históricos.

4) Conectar-se novamente ao Banco de Dados como um *superusuário* e executar o comando: SELECT *pg_stop_backup()*. Não é necessário ficar muito preocupado com o tempo decorrido entre a execução da função *pg_start_backup* e o início da realização da cópia de segurança, nem entre o fim da realização da cópia de segurança e a execução de *pg_stop_backup*.

Para poder utilizar esta cópia de segurança base, devem ser mantidas todas as cópias dos arquivos de segmento do WAL gerados no momento ou após o início da mesma. Para

ajudar a realizar esta tarefa, a função *pg_stop_backup* cria o *arquivo de história de cópia de segurança*, que é armazenado imediatamente na área de cópia do WAL. Este arquivo recebe um nome derivado do primeiro arquivo de segmento do WAL que é necessário possuir para fazer uso da cópia de segurança. Por exemplo, se o arquivo do WAL tiver o nome *000000010000123400000055CD*, o arquivo de história de cópia de segurança vai ter um nome parecido com *000000010000123400000055CD.007C9330.backup.* A segunda parte do nome do arquivo representa a posição exata dentro do arquivo do WAL, podendo normalmente ser ignorada. Uma vez que o arquivo contendo a cópia de segurança base tenha sido guardado em local seguro, podem ser apagados todos os arquivos de segmento do WAL com nomes numericamente precedentes a este número.

O arquivo de história de cópia de segurança é apenas um pequeno arquivo texto. Contém a cadeia de caracteres rótulo fornecida à função *pg_start_backup*, assim como as horas de início e fim da cópia de segurança. Se o rótulo for utilizado para identificar onde está armazenada a cópia de segurança base do Banco de Dados, então basta o arquivo de história de cópia de segurança para se saber qual é o arquivo de cópia de segurança a ser restaurado.

O módulo *Backup Log* é executado baseado em uma periodicidade, como está sendo indicada na Figura 5.9.

Esse módulo é o mais importante, pois é ele quem garante o funcionamento do Banco de Dados *Standby*. Possui alguns parâmetros importantes, como localizar os arquivos WAL para serem aplicados através de seu diretório de origem e em que local deixar esses arquivos para que o Banco de Dados *Standby* possa utilizar, através do parâmetro diretório destino.

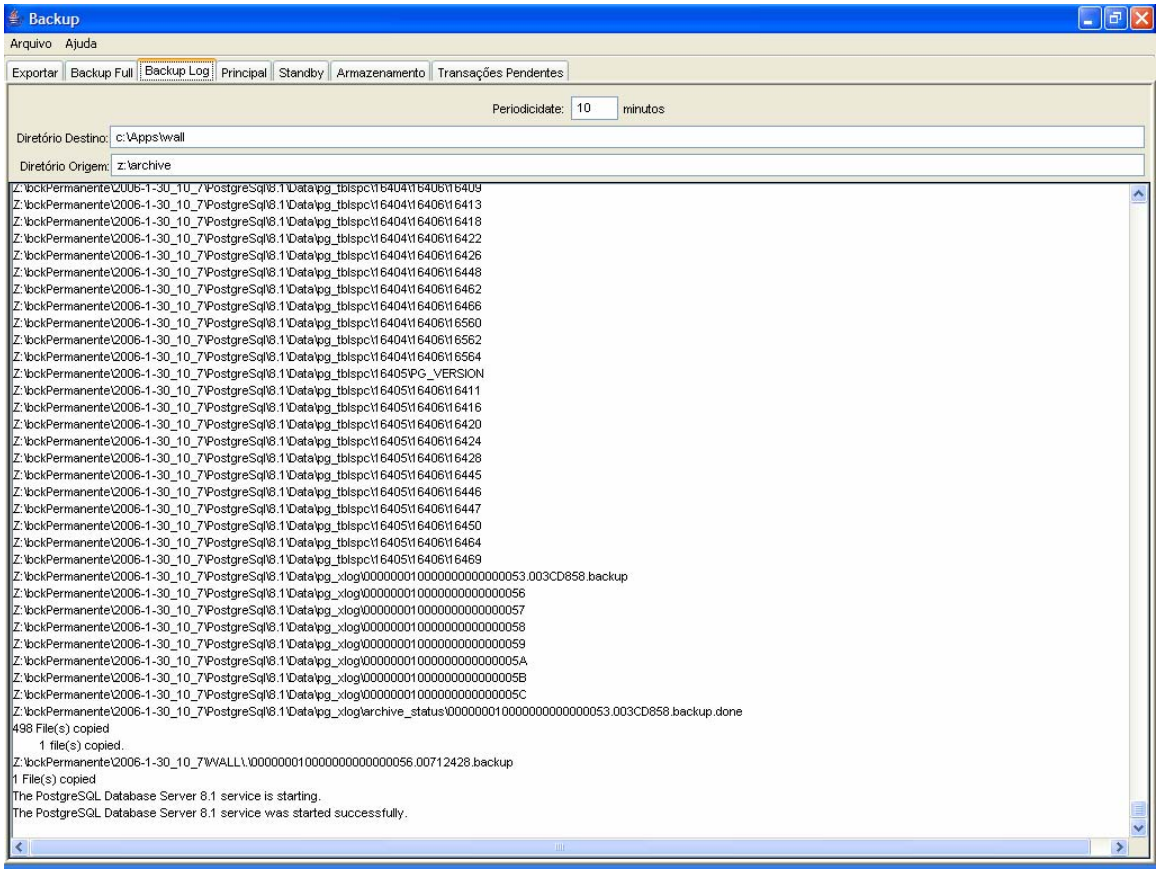

Figura 5.9: Módulo Backup Log da Ferramenta SABaDO

O diretório de destino é onde o arquivo **recovery.conf**[12] busca os arquivos WAL para poder aplicar no Banco de Dados *Standby*. Depois de aplicado todos os WAL pendentes esse arquivo é renomeado para **recovery.done,** sinalizando o término da aplicação dos arquivos WAL. O comando utilizado para inicializar o Banco de Dados deve ser informado no módulo *Standby*. Esse módulo também possui o comando utilizado para fazer um *shutdown* no Banco de Dados *Standby*, conforme mostra a Figura 5.10:

[12] Arquivo de configuração do PostgreSQL. Seu conteúdo está detalhado no Apêndice B

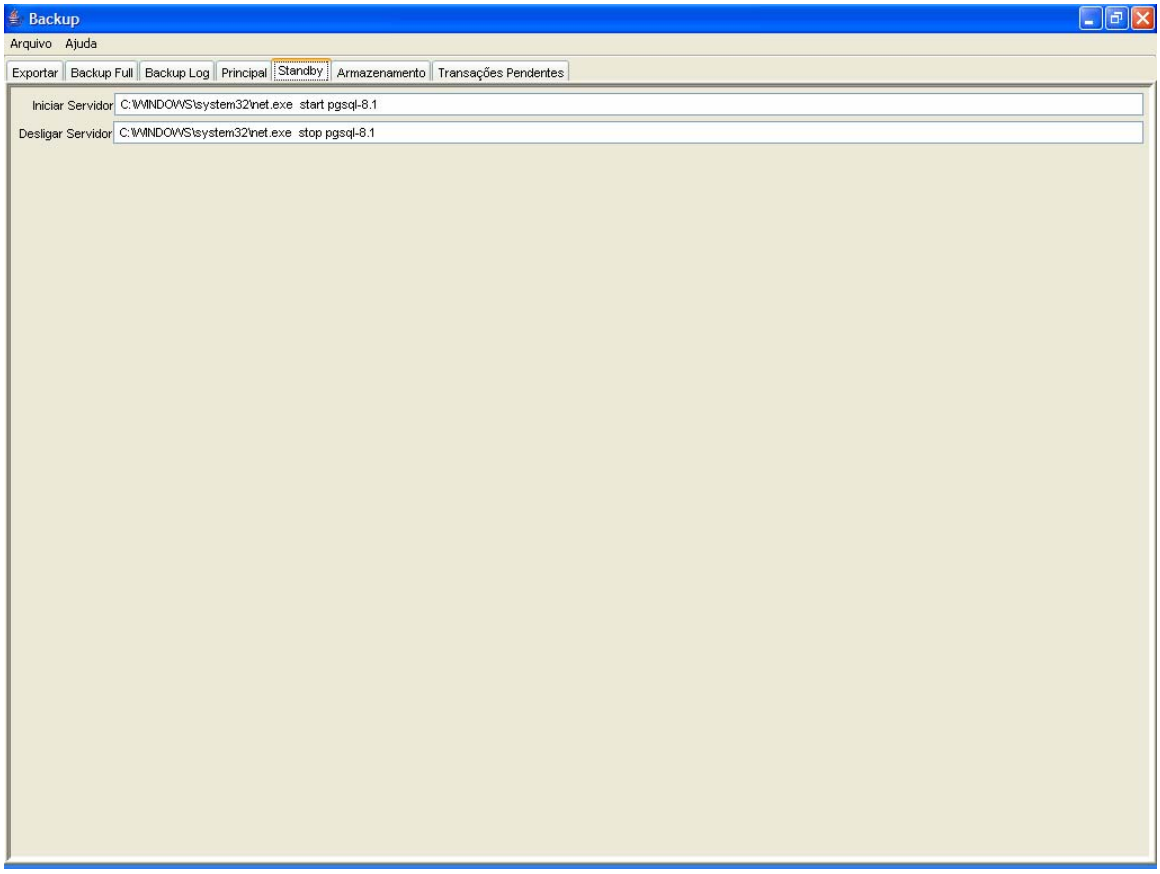

Figura 5.10: Módulo Backup Standby da Ferramenta SABaDO

O módulo Transações Pendentes tem o objetivo de descobrir quais transações foram iniciadas no Servidor de Produção e ainda não foram feitas no Servidor de Banco de Dados *Standby*, através do conceito de arquivos WAL, conforme Figura 5.11.

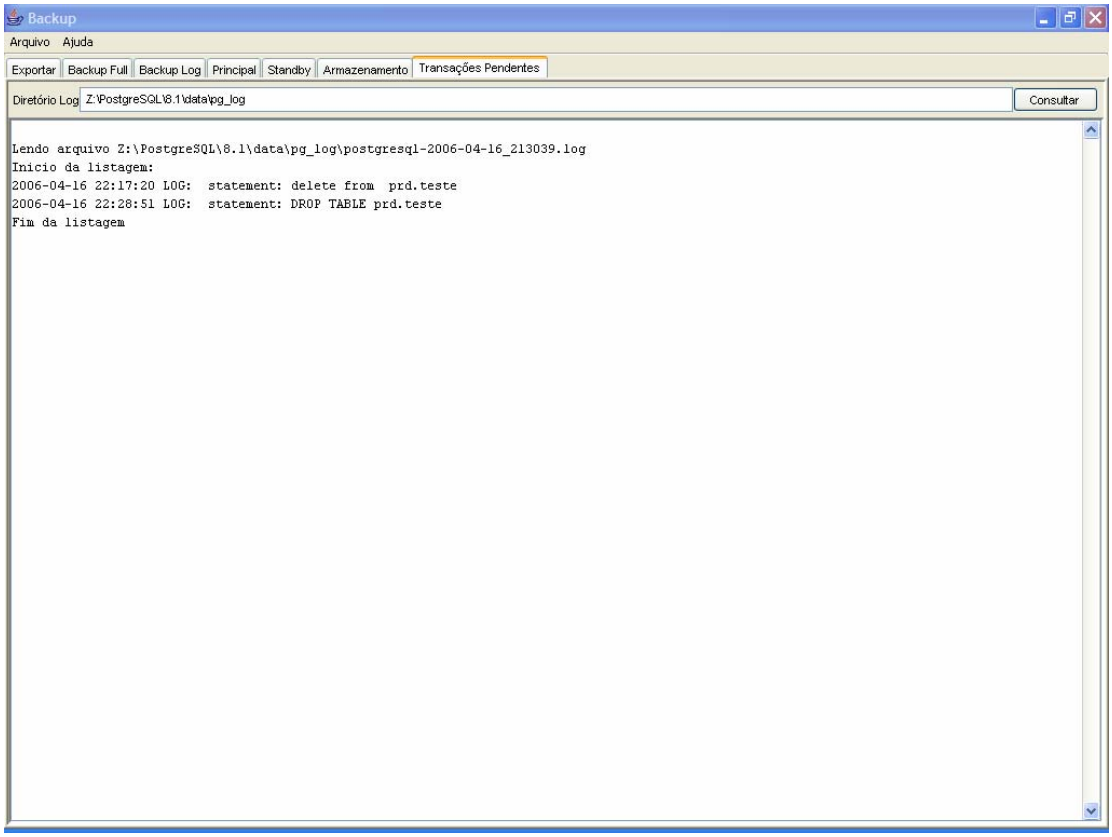

Figura 5.11: Módulo Transações Pendentes da Ferramenta SABaDO

É um módulo extremamente importante, pois é possível fazer um acompanhamento das transações que estão acontecendo no Servidor Principal, bem como saber se essas transações foram aplicadas no Servidor de Banco de Dados *Standby*. No caso de uma falha do Servidor Principal, esse módulo se torna essencial, pois é a única forma de saber as transações que serão perdidas no Banco de Dados de Produção.

6 Conclusão

Um dos principais objetivos desse trabalho é definir uma estratégia de *backup* utilizando Banco de Dados *Standby* para o SGBD PostgreSQL, com o propósito de garantir proteção contra interrupção no funcionamento do Servidor de Produção, proteção contra perda de arquivos de dados, proteção contra desastres e principalmente, reduzir o tempo de inatividade do Banco de Dados em caso de um desastre. Pode-se dizer que esse objetivo foi alcançado.

O fato do SGBD *Standby* estar sendo executado em um outro *hardware*, ou outro Servidor, suas operações não afetam o desempenho do Banco de Dados de Produção. Os recursos físicos exigidos para executar um Banco de Dados *Standby* são consumidos totalmente do Servidor *Standby*. Outro fator importante a ser levado em consideração é que a Ferramenta SABaDO é executada do Servidor *Standby*, e simplesmente realizando acessos ao Banco de Dados de Produção. Essa funcionalidade foi desenvolvida justamente para evitar utilização de CPU e memória do Servidor de Banco de Dados de Produção.

A Ferramenta SABaDO realização *backup full* ou *backup* completo do Banco de Dados em utilização pelos usuários, esse procedimento melhora a cópia dos arquivos físicos do Banco de Dados e não deixa o Banco de Dados de Produção inoperante em nenhum momento da criação do *backup*.

Estabeleceu-se uma terminologia para os arquivos de *logs* de transações do PostgreSQL, tendo como alvo principal a alimentação de um Servidor de Banco de Dados *Standby*.

A partir do exemplo de aplicação de um sistema de *Backup* foi possível observar que o tempo de inatividade após uma falha no SGBD Principal é praticamente nulo, pois o Servidor de Banco de Dados *Standby* contém praticamente todas as transações do Servidor de

Produção, ou simplesmente ele não possui os últimos minutos ou o tempo de transferência de *logs* informado pela Ferramenta. Consequentemente, o tempo de aplicação desses últimos *logs* é relativamente pequeno.

6.1 Contribuições da Dissertação

As contribuições desta dissertação são caracterizadas pelo desenvolvimento de estratégias de *backup* e *recovery* utilizando Banco de Dados *Standby* em um SGBD de tecnologia aberta, sendo escolhido como SGBD, o PostgreSQL. As principais contribuições são apresentadas e discutidas a seguir:

a) A primeira contribuição deste trabalho foi observar e pesquisar as técnicas de *Backup* do PostgreSQL. Foi constatado que esse SGBD não possui técnicas de *backup* avançadas, daí a necessidade da criação de uma técnica de *backup* mais segura e com um tempo menor de inatividade das operações do Banco de Dados Principal.

b) Geração de um *backup* lógico do Banco de Dados Principal. Essa forma de *backup* é extremamente importante para transferência de Banco de Dados de um Servidor para outro e principalmente, é uma forma de *backup* importante e que deve ser guardada em um dispositivo de armazenamento permanente, pois através desse *backup* é possível restaurar uma ou mais tabelas de um determinado usuário.

c) Criação de um *backup full* do Banco de Dados. Essa forma de *backup* copia todos os arquivos físicos do Banco de Dados que também devem ser guardados em um dispositivo de armazenamento permanente. Essa forma de *backup* é extremamente importante para a criação inicial do processo de Banco de Dados *Standby*.

Essa forma de backup copia todos os arquivos físicos do Banco de Dados que também devem ser guardados em um dispositivo de armazenamento permanente e é extremamente importante para a criação inicial do processo de Banco de Dados *Standby*.

d) Ativação e cópia dos arquivos de transações. O PostgreSQL armazena todas as transações executadas no Banco de Dados em arquivos de *log*, denominados de WAL. A Ferramenta SABaDO guarda esses arquivos após uma cópia integral do Banco de Dados e principalmente, transfere para um Servidor *Standby* onde inicia o processo de aplicação desses arquivos de transação. É nessa etapa que é garantida todo o armazenamento das transações que foram executadas no Servidor Principal.

e) Ativação do Servidor *Standby*. Um simples *start* do Banco de Dados *Standby* disponibiliza esse Banco de Dados para Produção após a aplicação dos últimos arquivos de *logs*. Esse *start* inicial é feito pela Ferramenta SABaDO e transparente para o administrador do Banco de Dados.

f) Tempo de retorno do Banco de Dados. O tempo de retorno de um Banco de Dados danificado se resume a poucos minutos com a utilização da Ferramenta SABaDO. No processo tradicional de volta de *backup* seria gasto algumas horas para voltar um Banco de Dados de aproximadamente 2GB.

6.2 Resultados Obtidos

A Ferramenta SABaDO teve como base de teste três empresas de domínio privado do estado do Paraná, uma situada em Maringá, outra em Marialva e a terceira em Paranaguá. São empresas com atividades distintas que utilizam ferramentas de desenvolvimento diferenciadas, mas em comum, possuem o SGBD PostgreSQL.

As empresas adquiriram Servidores de Banco de Dados complementares para a implantação da tecnologia *Standby*. Para a aquisição desses equipamentos foram sugeridos *hardwares* compatíveis com os Servidores de Banco de Dados de Produção, para evitar problemas de alocação física de arquivos de dados.

Os critérios de utilização da Ferramenta SABaDO foram os mesmos critérios para as três empresas, levando em consideração: a criação de *Backup* Lógico, *Backup* Físico da Base de Dados, transferência de arquivos de *logs*, aplicação dos arquivos de *logs* e condições para se tornar disponível o Servidor *Standby* para produção.

As três empresas não possuíam estruturas de *Backups* válidas e coerentes, o *Backup* Lógico foi uma excelente estratégia, pois com ele é possível recuperar partes específicas do Banco de Dados, como tabelas ou *storage procedures* armazenadas no Banco de Dados. Esse *Backup* é feito diariamente e armazenado em discos e também em um armazenamento externo, como fita *dat*, onde serão arquivados e guardados por um período mínimo de dois anos, conforme critério adotado pelas empresas.

A estratégia de criação de *Backup* Físico foi uma saída importante para a recuperação integral do Banco de Dados, em caso de falha de *hardware*. Diariamente será feito um *Backup* integral dos arquivos físicos do Banco de Dados e armazenados em um disco de *Backup* e também serão enviados para a fita *dat*. Essa forma de *Backup* fornece a garantia de ter o Banco de Dados inteiro armazenado no *Backup*, onde será possível fazer a volta integral desse Banco até o momento de sua realização. Na empresa de Paranaguá, que possuía uma Base de Dados de 65G foi necessário fazer uma restauração completa desse *Backup*, visto que o disco que armazenava os dados do sistema sofreu uma falha. A volta do *Backup* integral levou menos de uma hora e não houve perda de informação ou de dados.

A situação mais importante e o principal objeto de estudo dessa dissertação foi a transferência e aplicação dos arquivos de *logs* das transações que estavam sendo executadas no Servidor Principal.

A partir de uma periodicidade informada na Ferramenta SABaDO, é realizada a transferência e aplicação dos arquivos de *logs*. Essa periodicidade é extremamente importante, pois os arquivos de *logs* são gerados baseados nas transações que estão ocorrendo no Banco de Dados. Portanto, alguns Bancos geram arquivos de *logs* com maior ou menor freqüência, dependendo do número de transações que o Servidor vem recebendo.

Um dos Servidores avaliados, por exemplo, recebeu poucas transações, o que justificou uma periodicidade maior na aplicação dos *logs*, onde para essa empresa foi definido um intervalo de 40 minutos para transferência e aplicação dos *logs*. Já as empresas de Maringá e Paranaguá, possuem movimentações maiores de Banco de Dados e teve esse parâmetro configurado para 10 minutos.

Em outra empresa foi feito um teste de simulação de falha do Servidor Principal, onde foi interrompido o fornecimento de energia desse Servidor para simular uma falha de *hardware*. Depois de detectada a falha, iniciou o processo de preparação do Servidor *Standby* para ser o Servidor Principal. Em 15 minutos o Banco de Dados *Standby* estava disponível para utilização e sem perda de informações (transações) que estavam sendo executadas no Servidor que falhou.

Em outras duas empresas foram realizadas acompanhamentos de transações que estavam sendo executadas e que ainda não tinham sido enviadas para o Servidor *Standby*. A Ferramenta SABaDO consegue visualizar as transações que foram executadas no Servidor Principal e ainda não foram enviadas para o Servidor *Standby*. Isso facilita muito, pois é possível saber quais transações deveriam ser executadas no Servidor *Standby* caso ocorra uma falha no momento em que a transação está sendo executada no Servidor Principal.

6.3 Trabalhos Futuros

As abordagens apresentadas nesta dissertação envolveram estratégias de *backup* de um Banco de Dados PostgreSQL utilizando o recurso *Standby*. Visando a um melhor planejamento de estratégias de *backup* pode-se visualizar direções de pesquisas futuras relevantes para a área de *backup on-line*.

Outra sugestão de trabalho futuro é aplicar essas técnicas de *Backup* em outros Sistemas Gerenciadores de Banco de Dados de tecnologia livre, visto que essa técnica pode garantir uma confiabilidade maior no armazenamento das informações, pois trabalha com o armazenamento em mais de um Servidor.

REFERÊNCIAS

ALMEIDA, Leandro Clementino; AZEVEDO, Humberto da Silva. **Utilização do Apache, PHP e Banco de Dados PostgreSQL para Desenvolvimento Web,** 2003.

BRETON, Robert. **Replication Strategies for High Availability and Disaster Recovery.** IEEE Computer Society Technical Committee on Data Engineering, 1988.

CHEN, Peter. **Modelagem de Dados: A Abordagem Entidade-Relacionamento para Projeto Lógico.** São Paulo: Makron Books, 1990.

CIRNE, Liliane. **Comunicação em Grupo em Java**. Dissertação de Mestrado – II Workshop Interno do LaSiD-Wola'98 – Universidade Federal da Paraíba – PB – Brasil, 1998.

COURTRIGHT, William V.; GIBSON, Garth A. **Backward Error Recovery in Redundant Disk Arrays.** Department of Electrical and Computer Engineering, 1994.

DATE, C. J. **Introdução a Sistemas de Banco de Dados.** 7 ed. São Paulo: Editora Campos, 2000.

DEITEL, H. M.; DEITEL, P. J. **JAVA Como Programar.** 3 ed. Porto Alegre: Bookman, 2001.

DRAKE, Sam; MCINNIS Dale M.; SKÖLD Martin; SHIVASTAVA Alok; THALMANN Lars; TIKKANEN Matti; TORBJORNSEN Oystein; WOLSKI Antoni. **Architecture of Highly Available Databases.** International Service Availability Symposium, 2004.

ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems.** 2 ed. Addison Wesley, 2005.

FERNANDES, Jorge Henrique Cabral. **Integrando Java e Banco de Dados**. XV Simpósio Brasileiro de Banco de Dados – Departamento de Informática e Matemática Aplicada – Universidade Federal do Rio Grande do Norte, João Pessoa – PB – Brasil, 2000.

FERNANDES, Lúcia. **Oracle 9i para Desenvolvedores ORACLE DEVELOPER 6i Curso Completo.** Rio de Janeiro: Axcel Books do Brasil Editora, 2002.

HEUSER, Carlos Alberto. **Projeto de Banco de Dados.** 4 ed. Porto Alegre: Instituto de Informática da UFRGS: Editora Afiliada, 2001.

KING, Richard P.; HALIM, Nagui; MOLINA, Hector Garcia; POLYZOIS, Christos A. **Management of a Remote Backup Copy for Disaster Recovery.** ACM Transaction on Database Systems, v. 16, n. 2, 1991.

KORTH, Henry F.; SILBERSCHATZ, Abraham; SUDARSHAN, S. **Sistema de Banco de Dados".** 3 ed. São Paulo: Makron Books, 2005.

MEHROTRA Sharad; HU Kexiang; KAPLAN Simon. **Dealing with Partial Failures in Multiple Processor Primary-Backup Systems.** Proceedings of the Sixth ACM International Conference on Information and Knowledge Management, 1997.

MOLINA, Hector Garcia; POLYZOIS, Christos A. **Two Epoch Algorithms for Disaster Recovery.** Proceedings of the Sixteenth International Conference on Very Large Databases, 1990.

MORAES, Regina Lúcia de Oliveira. **Estratégia para Testes de Componentes de Banco de Dados Orientados a Objetos utilizando Injeção de Falhas.** Dissertação de Mestrado – IC – UNICAMP, Campinas – SP – Brasil, 2003.

NASCIMENTO, Francisco Assis; FORNARI, Miguel Rodrigues. **Linguagem de Programação Java.** Universidade Luterana do Brasil, Canoas – RS – Brasil, 2003.

RIOS, Marcos Gonçalves. **Documentação de Sistemas de Software Integrada ao Processo de Desenvolvimento**. III Semana de Pós-Graduação em Ciência da Computação SPG'99 – Departamento de Ciência da Computação – Universidade Federal de Minas Gerais, Belo Horizonte- MG – Brasil, 1999.

SPOTO, Edmundo S. **Teste Estrutural de Programas de Aplicação de Banco de Dados Relacional.** Tese de Doutorado – DCA/FEEC – UNICAMP, Campinas – SP – Brasil, 2000.

SUN MICROSYSTEMS. **The Java Language a White Paper.** Sun Microsystems, 1995.

VERHOFSTAD, Joost S. M. **Recovery Techniques For Database Systems.** ACM Transaction on Database Systems, v. 10, n. 2, 1978.

WIESMANN, Matthias; PEDONE, Fernando; SCHIPER, André. **Database Replication Techniques: a Three Parameter Classification.** Proc. Of the 19th IEEE Symposium on Reliable Distributed Systems, 2000.

YOUNG, Chu S.. **Banco de Dados. Organização, Sistemas e Administração.** São Paulo: Editora Atlas S.A., 1990.

APÊNCICE A

Esta parte contém informação de referência para os comandos SQL suportados pelo PostgreSQL.

ALTER TABLE

O comando ALTER TABLE altera a definição de uma tabela existente.

Sintaxe:

ALTER TABLE [ONLY] nome [*] ADD [COLUMN] coluna tipo [restrição_de_coluna [...]] ALTER TABLE [ONLY] nome [*] DROP [COLUMN] coluna [RESTRICT | CASCADE]

BEGIN

Este comando inicia um bloco de transação. Por padrão, o PostgreSQL realiza as transações em modo não encadeado, também denominado de *autocommit* (auto-efetivação). Cada comando é executado em sua própria transação e uma efetivação é implicitamente realizada ao final do comando. Caso esse comando não termine com êxito é realizado um *rollback*.

O comando BEGIN inicia uma transação no modo encadeado, ou seja, todas as declarações após esse comando são executadas como uma única transação, até que se encontre um comando implícito COMMIT ou ROLLBACK.

Os comandos são realizados mais rapidamente no modo encadeado, pois cada início de transação requer uma atividade significativa de CPU e de disco.

A realização de diversos comandos em uma transação é requerida por razão de consistência, quando diversas tabelas relacionadas são alteradas. Os outros usuários não podem visualizar os estados intermediários enquanto todas as atualizações realizadas não forem finalizadas.

O comando BEGIN é uma extensão do PostgreSQL à linguagem. Esse comando não está explícito no SQL92, o início da transação é sempre implícita, finalizada pelo comando COMMIT ou pelo comando ROLLBACK.

Sintaxe:

BEGIN [WORK | TRANSACTION]

COMMIT

O comando COMMIT efetiva a transação corrente. Todas as modificações efetuadas pela transação ficam visíveis, e existe a garantia de permanecerem se uma falha ocorrer, baseado em *site* oficial do PostgreSQL.

Sintaxe:

COMMIT [WORK | TRANSACTION]

CREATE TABLE

O comando CREATE TABLE cria uma tabela, inicialmente vazia, no Banco de Dados atual. O usuário que executa o comando se torna o dono da tabela. Se o nome do esquema for fornecido, então a tabela é criada no esquema especificado, senão é criada no esquema corrente. As tabelas temporárias são criadas em um esquema especial e, portanto, o nome do esquema não pode ser especificado ao se criar tabelas temporárias. O nome da tabela deve ser diferente do nome de qualquer outra tabela, seqüência, índice ou visão no mesmo esquema. As cláusulas opcionais de restrição especificam as restrições que as linhas novas ou modificadas devem satisfazer para a operação de inserção ou de modificação ser aceita.

Sintaxe:

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE
nome_da_tabela ( { nome_da_coluna tipo_de_dado [ DEFAULT
expressão_padrão ] [ restrição_de_coluna [ ... ] ] |
restrição_de_tabela | LIKE tabela_ancestral [ { INCLUDING |
EXCLUDING } DEFAULTS ] } [, ... ]
```

DELETE

O comando DELETE exclui da tabela especificada as linhas que satisfazem a cláusula WHERE. Se a cláusula WHERE estiver ausente, o efeito produzido é a exclusão de todas as linhas da tabela. O resultado é uma tabela válida, porém vazia. Por padrão, o comando DELETE exclui linhas da tabela especificada, e de todas as suas descendentes. Se

for desejado excluir linhas apenas da tabela especificada, deve ser utilizada a cláusula ONLY. É necessário possuir o privilégio DELETE na tabela para excluir linhas da mesma, assim como o privilégio SELECT para todas as tabelas cujos valores são acessados na condição.

Sintaxe:

DELETE FROM [ONLY] tabela [WHERE condição]

DROP TABLE

O comando DROP TABLE remove tabelas do Banco de Dados. Somente o criador ou o dono do objeto pode remover a tabela. O comando DROP TABLE remove todos os índices, regras, gatilhos e restrições existentes na tabela. Entretanto, para remover uma tabela referenciada por uma restrição de chave estrangeira de outra tabela, deve ser especificado CASCADE, essa opção remove a restrição de chave estrangeira.

Sintaxe:

DROP TABLE nome [, ...] [CASCADE | RESTRICT]

INSERT

O comando INSERT permite inserir novas linhas na tabela. Pode ser inserida uma linha de cada vez, ou várias linhas do resultado de uma consulta. As colunas da lista de inserção podem estar em qualquer ordem. As colunas que não constam da lista de inserção

são inseridas utilizando o valor padrão, seja seu valor padrão declarado ou nulo. Se a expressão para cada coluna não for do tipo de dado correto, será tentada uma conversão automática de tipo. É necessário possuir o privilégio INSERT na tabela para poder inserir linhas. Se for utilizada a cláusula consulta para inserir linhas a partir de uma consulta, também é necessário possuir o privilégio SELECT em todas as tabelas usadas na consulta, segundo s*ite* oficial do PostgreSQL.

Sintaxe:

INSERT INTO tabela [(coluna [, ...])]
{DEFAULT VALUES | VALUES ({ expressão | DEFAULT } [, ...]) | consulta}

ROLLBACK

O comando ROLLBACK desfaz a transação corrente, desconsiderando todas as modificações realizadas por esta transação.

Sintaxe:

ROLLBACK [WORK | TRANSACTION]

SAVEPOINT

SAVEPOINTS são usados juntamente com o comando ROLLBACK. Ele tem a finalidade de marcar um ponto intermediário na transação.

Esse comando pode ser útil quando há um programa que realizada diversos comandos INSERT, UPDATE ou DELETE. Antes de cada um deles, pode ser marcado um SAVEPOINT e, caso haja uma falha no programa, pode ser feito ROLLBACK somente até o ponto marcado, desfazendo todas as alterações feitas pelo programa e executá-lo de novo com as correções necessárias.

Quando acontece um ROLLBACK para um SAVEPOINT, todas as alterações que ocorreram após os SAVEPOINTS são desfeitas e todos os *locks* adquiridos após este ponto também são liberados. No entanto, a transação não é efetivada. Todas as alterações ocorridas antes do SAVEPOINT continuam travadas sem efetivação, esperando que o usuário realize um COMMIT.

Sintaxe:

```
SELECT VL_SAL FROM FUNC
   WHERE NM_FUNC  IN (‘DAVI’ , ‘DANIEL’);
        VL_SAL
  - - - - - - - - - - -
        2774
        1918

UPDATE FUNC
      SET VL_SAL = 4000 WHERE NM_FUNC = ‘DANIEL’;
1 linha atualizada.

SAVEPOINT DANIEL_SAL;
Ponto de salvamento criado.
```

```
UPDATE FUNC
SET VL_SAL = 3500 WHERE NM_FUNC = 'DAVI';
1 linha atualizada.

SAVEPOINT DAVI_SAL;
Ponto de salvamento criado.
```

Nesta sintaxe, foram realizadas alterações em dois funcionários, Davi e Daniel. Foram marcados dois pontos de controle, um depois de ter sido atualizado o salário de Daniel e outro após ter sido atualizado o salário de Davi.

SELECT

O comando SELECT retorna linhas de uma ou mais tabelas. Todos os elementos da lista FROM são computados. Se mais de um elemento for especificado na lista FROM, é feita uma junção entre estes. Se a cláusula WHERE for especificada, todas as linhas que não satisfazem a condição são eliminadas da saída. Se a cláusula GROUP BY estiver especificada, a saída é dividida em grupos de linhas que correspondem a um ou mais valores. Se a cláusula HAVING estiver presente, são eliminados os grupos que não satisfazem a condição especificada. As linhas da saída são computadas utilizando as expressões de saída do comando SELECT para cada linha selecionada. Se a cláusula ORDER BY for especificada, as linhas retornadas são ordenadas segundo a ordem especificada. Se ORDER BY não estiver presente, as linhas são retornadas na ordem em que o sistema considerar mais fácil de produzir. A opção DISTINCT elimina as linhas duplicadas presentes no resultado. Se a cláusula LIMIT ou OFFSET for especificada, o comando SELECT somente retorna um

subconjunto das linhas do resultado. A cláusula FOR UPDATE faz o comando SELECT bloquear as linhas selecionadas contra atualizações concorrentes. É necessário possuir o privilégio SELECT na tabela para poder ler seus valores. A utilização de FOR UPDATE requer também o privilégio UPDATE.

Sintaxe:

```
SELECT [ ALL | DISTINCT [ ON ( expressão [, ...] ) ] ] * | expressão [ AS
        nome_de_saída ] [, ...] [ FROM item_de [, ...] ] [ WHERE condição ]
        [ GROUP BY expressão [, ...] ] [ HAVING condição [, ...] ] [ {
        UNION | INTERSECT | EXCEPT } [ ALL ] seleção ] [ ORDER BY
        expressão [ ASC | DESC | USING operador ] [, ...] ] [ LIMIT {
        contador | ALL } ] [ OFFSET início ] [ FOR UPDATE [ OF
        nome_da_tabela [, ...] ] ]
```

UPDATE

O comando UPDATE atualiza os valores das colunas especificadas em todas as linhas que satisfazem a condição. Somente as colunas a serem atualizadas precisam ser mencionadas no comando. As colunas que não são explicitamente atualizadas na cláusula SET mantêm seus valores anteriores. Por padrão, o comando UPDATE atualiza linhas na tabela especificada e nas suas descendentes. Para atualizar apenas a tabela especificada, deve ser utilizada a cláusula ONLY. É necessário possuir o privilégio UPDATE na tabela para atualizá-la, assim como o privilégio SELECT em todas as tabelas cujos valores são lidos pela expressão ou pela condição, segundo s*ite* oficial do PostgreSQL.

Sintaxe:

```
UPDATE [ ONLY ] tabela SET coluna = { expressão | DEFAULT } [, ...]
   [ FROM lista_de ]
   [ WHERE condição ]
```

APÊNDICE B

Esta parte contém informação de referência para os comandos suportados pela Ferramenta SABaDO desenvolvida nessa dissertação de Mestrado.

PG_DUMP

O *pg_dump* é um utilitário para fazer cópia de segurança de um Banco de Dados do PostgreSQL. São feitas cópias de segurança consistentes mesmo que o Banco de Dados esteja sendo utilizado.

As cópias de segurança podem ser feitas no formato de *script* ou em outros formatos. As cópias de segurança no formato de *script* são arquivos no formato texto puro, contendo os comandos SQL necessários para reconstruir o Banco de Dados no estado em que este se encontrava quando foi salvo. Para restaurar a partir destes *scripts*, deve ser utilizado o aplicativo *psql*.

Os formatos de arquivo de cópia de segurança alternativos devem ser utilizados com o utilitário *pg_restore* para reconstruir o Banco de Dados. Estes formatos permitem que o *pg_restore* selecione o que será restaurado, ou mesmo reordene os itens antes de restaurá-los.

Quando usado com um dos formatos de cópia de segurança alternativos, e combinado com o *pg_restore*, o *pg_dump* fornece um mecanismo flexível para cópias de segurança e transferência. O *pg_dump* pode ser usado para fazer a cópia de segurança de todo o Banco de Dados e, posteriormente, o *pg_restore* pode ser usado para examinar a cópia de segurança e/ou selecionar as partes do Banco de Dados a serem restauradas.

Esse comando possui algumas opções de escritas que podem ser vistas na tabela abaixo:

Tabela B1: Opções de escritas das diretivas de ...

Opção	Descrição	Significado
-a	*Data-only*	Salva somente os dados, não salva o esquema (definições de dados). Esta opção só faz sentido para o formato texto-puro. Para os formatos alternativos esta opção pode ser especificada ao chamar o *pg_restore*.
-b	*Blobs*	Inclui os objetos grandes na cópia de segurança. Deve ser selecionado um formato de saída não-texto.
-c	*Clean*	Inclui comandos para remover (*drop*) os objetos do Banco de Dados antes dos comandos para criá-los. Esta opção só faz sentido para o formato texto-puro. Para os formatos alternativos esta opção pode ser especificada ao chamar o *pg_restore*.
-C	*Create*	Inicia a saída por um comando para criar o Banco de Dados e conectar ao Banco de Dados criado. Com um script assim não importa qual Banco de Dados se está conectando antes de executar o *script*. Esta opção só faz sentido para o formato texto-puro. Para os formatos alternativos a opção pode ser especificada ao chamar o *pg_restore*.
-d	*Inserts*	Salva os dados como comandos INSERT, em vez de COPY. Torna a restauração muito lenta. Sua utilização principal é para fazer cópias de segurança que possam ser carregadas em outros Bancos de Dados que não o PostgreSQL. Deve ser observado que a restauração pode falhar inteiramente se a ordem das colunas tiver sido modificada.
-D	*Column-inserts*	Salva os dados como comandos INSERT explicitando os nomes das colunas (INSERT INTO *tabela* (*coluna*,...) VALUES ...). Torna a restauração muito lenta. Sua utilização principal é para fazer cópias de segurança que possam ser carregadas em outros Bancos de Dados que não o PostgreSQL.
-f	*File*	Envia a saída para o arquivo especificado. Se for omitido é usada a saída padrão.
-F	*Format*	Seleciona o formato da saída. O formato pode ser um dos seguintes: **p** → gera um arquivo de script SQL no formato texto-puro (padrão); **t** → gera um arquivo *tar* adequado para servir de entrada para o *pg_restore*. A utilização deste formato de arquivo permite reordenar e/ou excluir objetos do Banco de Dados ao fazer a restauração. Também é possível limitar os dados a serem recarregados ao fazer a restauração. **C** → gera um arquivo personalizado adequado para servir de entrada para o *pg_restore*. Este é o formato mais flexível,

		porque permite a reordenação da restauração dos dados, assim como das definições dos objetos.
-i	*Ignore*	Ignora a diferença de versão entre o *pg_dump* e o servidor de banco de dados. O *pg_dump* pode tratar Bancos de Dados de versões anteriores do PostgreSQL, mas as versões muito antigas não são mais suportadas (atualmente as anteriores a 7.0). Esta opção deve ser utilizada se for necessário desconsiderar a verificação de versão.
-n	*Schema*	Salva apenas o conteúdo do *esquema* (dono do objeto). Se esta opção não for especificada, todos os *esquemas* no Banco de Dados especificado são salvos.
-o	*Oids*	Salva os identificadores de objeto (OIDs) de todas as tabelas como parte dos dados. Esta opção deve ser usada quando a coluna OID é referenciada de alguma maneira (por exemplo, em uma restrição de chave estrangeira). Caso contrário, esta opção não deve ser usada.
-O	*No-owner*	Não gera comandos para definir o dono dos objetos correspondendo ou correspondentes ao do Banco de Dados original. Por padrão, o *pg_dump* emite os comandos ALTER OWNER ou SET SESSION AUTHORIZATION para definir o dono dos objetos de Bancos de Dados criados.
-s	*Schema-only*	Salva somente o esquema (definições dos dados), não os dados.
-t	*Table*	Salva somente os dados da tabela. É possível existirem várias tabelas com o mesmo nome em esquemas diferentes.
-v	*Verbose*	Especifica o modo verboso, fazendo o *pg_dump* colocar comentários detalhados sobre os objetos e os tempos de início/fim no arquivo de cópia de segurança, e mensagens de progresso na saída de erro padrão.
-p	*Port*	Especifica a porta TCP para conexão com o Banco de Dados.
-U	*Username*	Conectar como o usuário especificado.
-w	*Password*	Força a solicitação da senha, o que deve acontecer automaticamente quando o servidor requer autenticação por senha.

PG_DUMPALL

O pg_dumpall é um utilitário para salvar todos os Bancos de Dados de um agrupamento do PostgreSQL em um arquivo de *script*. O arquivo de *script* contém comandos SQL que podem ser usados como entrada do utilitário psql para restaurar os Bancos de Dados. Isto é feito chamando o pg_dump para cada Banco de Dados do agrupamento. O pg_dumpall também salva os objetos globais, comuns a todos os Bancos de Dados. Atualmente são incluídas informações sobre os usuários do Banco de Dados e grupos, e permissões de acesso aplicadas aos Bancos de Dados como um todo.

Portanto, o pg_dumpall é uma solução integrada para realizar cópias de segurança dos Bancos de Dados. Entretanto, deve ser observada a seguinte limitação: não é possível salvar "objetos grandes", porque o pg_dump não pode salvar estes objetos em arquivos texto. Havendo Bancos de Dados contendo objetos grandes, estes devem ser salvos usando um dos modos de saída não-texto do pg_dump.

Como o pg_dumpall lê tabelas de todos os Bancos de Dados, muito provavelmente será necessário se conectar como um *superusuário* para poder gerar uma cópia completa. Também será necessário o privilégio de *superusuário* para executar o *script* produzido, para poder criar usuários e grupos, e para poder criar os Bancos de Dados.

O *script* SQL é escrito na saída padrão. Devem ser usados operadores de linha de comando para redirecionar para um arquivo.

O pg_dumpall precisa se conectar várias vezes ao servidor PostgreSQL, uma vez para cada Banco de Dados. Se for utilizada autenticação por senha, provavelmente será solicitada a senha cada uma destas vezes.

Esse comando possui algumas opções de escritas que podem ser vistas na tabela abaixo:

Opção	Descrição	Significado
-a	*Data-only*	Salva somente os dados, não salva o esquema (definições de dados).
-c	*Clean*	Inclui comandos para remover (*drop*) os objetos do Banco de Dados antes dos comandos para criá-los.
-d	*Inserts*	Salva os dados como comandos INSERT, em vez de COPY. Torna a restauração muito lenta. Sua utilização principal é para fazer cópias de segurança que possam ser carregadas em outros Bancos de Dados que não o PostgreSQL. Deve ser observada que a restauração pode falhar inteiramente se a ordem das colunas tiver sido modificada.
-D	*Column-inserts*	Salva os dados como comandos INSERT explicitando os nomes das colunas (INSERT INTO *tabela* (*coluna*,...) VALUES ...). Torna a restauração muito lenta. Sua utilização principal é para fazer cópias de segurança que possam ser carregadas em outros Bancos de Dados que não o PostgreSQL.
-g	*Globals-only*	Salva somente os objetos globais (usuários e grupos), e não o Banco de Dados.
-i	*Ignore*	Ignora a diferença de versão entre o pg_dump e o servidor de Banco de Dados. O pg_dump pode tratar Bancos de Dados de versões anteriores do PostgreSQL, mas as versões muito antigas não são mais suportadas (atualmente as anteriores a 7.0). Esta opção deve ser utilizada se for necessário desconsiderar a verificação de versão.
-o	*Oids*	Salva os identificadores de objeto (OIDs) de todas as tabelas como parte dos dados. Esta opção deve ser usada quando a coluna OID é referenciada de alguma maneira (por exemplo, em uma restrição de chave estrangeira). Caso contrário, esta opção não deve ser usada.
-O	*No-owner*	Não gera comandos para definir o dono dos objetos correspondendo ao do Banco de Dados original. Por padrão, o pg_dump emite os comandos ALTER OWNER ou SET SESSION AUTHORIZATION para definir o dono dos objetos de Bancos de Dados criados.
-s	*Schema-only*	Salva somente o esquema (definições dos dados), não os dados.
-v	*Verbose*	Especifica o modo verboso, fazendo o pg_dump colocar

		comentários detalhados sobre os objetos e os tempos de início/fim no arquivo de cópia de segurança, e mensagens de progresso na saída de erro padrão.
-h	*Host*	Especifica o nome de hospedeiro da máquina onde o servidor está executando
-p	*Port*	Especifica a porta TCP para conexão com o Banco de Dados.
-U	*Username*	Conectar como o usuário especificado.
-W	*Password*	Força a solicitação da senha, o que deve acontecer automaticamente quando o servidor requer autenticação por senha.

PG_RESTORE

O pg_restore é um utilitário para restaurar um Banco de Dados do PostgreSQL, a partir de uma cópia de segurança criada pelo pg_dump em um dos formatos não-texto-puro. São executados os comandos necessários para reconstruir o Banco de Dados, no estado em que este se encontrava na hora em que foi salvo. Os arquivos de cópia de segurança também permitem ao pg_restore selecionar o que será restaurado, ou mesmo reordenar os itens antes de serem restaurados.

Os arquivos de cópia de segurança são projetados para serem portáveis entre arquiteturas diferentes. O pg_restore pode operar de dois modos: Se o nome do Banco de Dados for especificado, a cópia de segurança é restaurada diretamente no Banco de Dados (Os objetos grandes só podem ser restaurados utilizando uma conexão direta com o Banco de Dados como esta). Senão, é criado um *script*, no formato texto-puro, contendo os comandos SQL necessários para reconstruir o Banco de Dados (escrito em um arquivo ou na saída padrão), semelhante aos *scripts* criados pelo pg_dump.

Esse comando possui algumas opções de escritas que podem ser vistas na tabela abaixo:

Opção	Descrição	Significado
-a	*Data-only*	Salva somente os dados, não salva o esquema (definições de dados).
-c	*Clean*	Remove os objetos do Banco de Dados antes de criá-los.
-C	*Create*	Cria o Banco de Dados antes de restaurá-lo. Quando esta opção é utilizada, o Banco de Dados especificado na opção –d é usado apenas para executar o comando CREATE DATABASE inicial. Todos os dados são restaurados no Banco de Dados cujo nome aparece na cópia de segurança.
-d	*Dbname*	Conecta ao Banco de Dados *nome_do_banco_de_dados* e restaura diretamente neste Banco de Dados.
-e	*Exit-on-error*	Termina se for encontrado um erro ao enviar os comandos SQL para o Banco de Dados. O padrão é continuar e mostrar um contador de erros ao término da restauração.
-f	*File*	Especifica o arquivo de saída para o script gerado, ou para conter a listagem quando for utilizada a opção -l. Por padrão a saída padrão.
-F	*Format*	Especifica o formato do arquivo da cópia de segurança. Não é necessário especificar o formato, porque o pg_restore determina o formato automaticamente. Se for especificado, pode ser um dos seguintes: **T** → a cópia de segurança é um arquivo *tar*. Este formato de cópia de segurança permite reordenar e/ou excluir elementos do esquema ao restaurar o Banco de Dados. Também permite limitar quais dados são recarregados ao restaurar. **C** → a cópia de segurança está no formato personalizado do pg_dump. Este é o formato mais flexível, porque permite reordenar a restauração dos dados e dos elementos do esquema. Também, este formato é comprimido por padrão.
-i	*Ignore*	Ignora a verificação da versão do Banco de Dados.
-I	*Index*	Restaura apenas a definição do índice especificado.
-l	*List*	Lista o conteúdo da cópia de segurança. A saída desta operação pode ser usada com a opção -L para restringir e reordenar os itens a serem restaurados.

-L	*Use-list*	Restaura apenas os elementos presentes no *arquivo_da_listagem*, e na ordem que aparecem neste arquivo. As linhas podem ser movidas e, também, podem virar comentário colocando um **;** no seu início
-O	*No-owner*	Não gera comandos para definir o dono dos objetos correspondendo ao do Banco de Dados original. Por padrão, o pg_dump emite os comandos ALTER OWNER ou SET SESSION AUTHORIZATION para definir o dono dos objetos de Bancos de Dados criados.
-s	*Schema-only*	Restaura somente o esquema (definições de dados), não os dados. Os valores das seqüências são reiniciados.
-S	*Superuser*	Especifica o nome de usuário do ***superusuário*** a ser usado para desabilitar os gatilhos.
-t	*Table*	Restaura apenas a definição e/ou dados da tabela especificada.
-T	*Trigger*	Restaura apenas o gatilho especificado.
-p	*Port*	Especifica a porta TCP para conexão com o Banco de Dados.
-U	*Username*	Conectar como o usuário especificado.
-w	*Password*	Força a solicitação da senha, o que deve acontecer automaticamente quando o servidor requer autenticação por senha.

PSQL

O psql é um aplicativo cliente do PostgreSQL comum. Para conectar a um Banco de Dados são necessárias algumas informações como: o nome do Banco de Dados, o nome do hospedeiro, o número da porta do servidor e o nome do usuário a ser usado para conectar. O psql pode ser informado sobre estes parâmetros por meio das opções de linha de comando -d, -h, -p e -U, respectivamente. Se for encontrado um argumento que não pertence a nenhuma opção, este será interpretado como o nome do Banco de Dados (ou o nome do usuário, se o nome do Banco de Dados já tiver sido fornecido). O número padrão para a porta é

determinado na compilação. Uma vez que o servidor de Banco de Dados usa o mesmo padrão, não é necessário especificar a porta na maioria dos casos.

Esse comando possui algumas opções de escritas que podem ser vistas na tabela abaixo:

Opção	**Descrição**	**Significado**
-a	*Echo-all*	Envia todas as linhas de entrada para a saída padrão à medida que são lidas. É mais útil para o processamento de *scripts* do que no modo interativo. Equivale a definir a variável ECHO como *all.*
-d	*Dbname*	Especifica o nome do Banco de Dados a se conectar. Equivale a especificar *nome_do_banco_de_dados* como o primeiro argumento não-opção na linha de comando.
-f	*File*	Usa o arquivo *nome_do_arquivo* como origem dos comandos, em vez de ler os comandos interativamente. Após processar o arquivo, o psql termina.
-l	*List*	Mostra todos os Bancos de Dados disponíveis, e depois termina. As outras opções, fora as de conexão, são ignoradas. Semelhante ao comando interno *\list*.
-p	*Port*	Especifica a porta TCP onde o servidor está atendendo as conexões. O padrão é obter o valor a partir da variável de ambiente PGPORT, se esta estiver definida, senão usar o valor padrão compilado (normalmente 5432).
-q	*Quiet*	Especifica que o psql deve trabalhar em silêncio. Por padrão, são exibidas mensagens de boas-vindas e várias outras mensagens informativas. Se esta opção for usada, nenhuma dessas mensagens são mostradas.
-U	*Username*	Conecta ao Banco de Dados como o usuário *nome_do_usuário* em vez do usuário padrão.
-W	*Password*	Força o psql solicitar a senha antes de conectar ao Banco de Dados. O psql deve solicitar, automaticamente, a senha sempre que o servidor requerer autenticação por senha. Como atualmente a detecção de solicitação de senha não é inteiramente confiável, esta opção existe para obrigar a solicitação. Se a senha não for solicitada, e o servidor requerer autenticação por senha, a tentativa de conexão não será bem-sucedida.

O psql disponibiliza vários meta-comandos para utilização. Meta-comando é qualquer texto digitado no psql começando por uma contrabarra (\). Estes comandos ajudam a tornar o psql mais útil para administração. Os metacomandos são geralmente chamados de comandos de barra ou de contrabarra.

Segue abaixo alguns meta-comandos mais importantes:

Tabela A1:

Opção	**Significado**
connect	Estabelece a conexão com um Banco de Dados novo e/ou com um usuário novo. A conexão anterior é fechada. Se o *nome_do_banco_de_dados* for - (hífen), então é assumido o Banco de Dados corrente. Se o *nome_do_usuário* for omitido, então o nome do usuário corrente é utilizado.
dg	Lista todos os grupos de Bancos de Dados.
distvS	Este não é o nome do comando: As letras i, s, t, v, S correspondem a índice, seqüência, tabela, visão e tabela do sistema, respectivamente. Pode ser especificada qualquer uma ou todas as letras, em qualquer ordem, para obter a listagem de todos os objetos correspondentes. A letra S restringe a listagem aos objetos do sistema; sem o S, somente são mostrados os objetos que não são do sistema.
du	Lista todos os usuários do Banco de Dados
h	Fornece ajuda de sintaxe para o comando SQL especificado. Senão for especificado o *comando*, então o psql lista todos os comandos para os quais existe ajuda de sintaxe disponível.
q	Sair do programa psql.

Arquivo pontgresql.conf

O SGBD PostgreSQL utiliza um arquivo texto chamado POSTGRESQL.CONF para carregar parâmetros de Banco de Dados. Esse arquivo é lido toda vez que é iniciado o SGBD, portanto, qualquer alteração feita nesse arquivo é necessário tirar o Banco de Dados de operação e depois colocá-lo novamente.

Alguns parâmetros são de extrema importância para o SGBD PostgreSQL, pois eles definem utilização de memórias, localização de arquivos, informações de autenticações e conexões, arquivos WAL, otimização de consulta, *logs* de erros, estatísticas de comandos e conexões de clientes.

A tabela abaixo mostra alguns dos principais parâmetros utilizados pelo PostgreSQL.

Comando	**Significado**
listen_addresses	Especifica o endereço, ou endereços, de TCP/IP onde o servidor atende as conexões dos aplicativos cliente. O valor tem a forma de uma lista de nomes de hospedeiros, ou de endereços numéricos de IP, separados por vírgula. A entrada especial * corresponde a todas as *interfaces* de IP disponíveis. Se a lista estiver vazia, o servidor não atende nenhuma *interface* IP e, neste caso, somente podem ser utilizados soquetes do domínio Unix para conectar ao servidor de Banco de Dados. O valor padrão é *localhost*, que permite serem feitas apenas conexões locais.
fsync	Se o valor deste parâmetro for *true*, o servidor PostgreSQL utilizará a chamada de sistema fsync() em vários lugares para ter certeza que as atualizações estão fisicamente escritas no disco. Isto garante que o agrupamento de Bancos de Dados vai ser recuperado em um estado consistente após um problema de máquina ou do sistema operacional. Entretanto, a utilização de fsync() produz uma degradação de desempenho, quando a transação é efetivada, o PostgreSQL tem de aguardar o sistema operacional descarregar o *log* de escrita prévia (WAL) no disco. Quando fsync está desabilitado, o sistema operacional pode desempenhar uma melhor maneira de controlar os *buffers* de dados, ordenação e retardo na escrita. Isto pode produzir uma melhora significativa no desempenho. Porém, no caso de uma queda do sistema, podem ser perdidos, em parte ou por inteiro, os resultados das últimas transações efetivadas. Devido aos riscos envolvidos não existe uma definição universalmente aceita para fsync. Alguns administradores sempre desabilitam fsync, enquanto outros só desabilitam para cargas de dados pesadas, onde claramente existe um ponto de recomeço se algo de errado acontecer, enquanto outros administradores sempre deixam fsync habilitado. O valor padrão para fsync é habilitado, para obter o máximo de confiabilidade.
archive_command	O comando a ser passado para o interpretador de comandos para executar a cópia de segurança de um segmento da série de arquivos do WAL quando este é completado. Se for uma cadeia de caracteres vazia, opção *default*, a cópia de segurança do WAL é desabilitada.
log_statement	Controla quais declarações SQL são registradas. Os valores válidos são *none, ddl, mod* e *all*. ***ddl*** → registra todos os comandos de definição de dados, como CREATE, ALTER e DROP.

	mod → registra todos as instruções de DDL, mais INSERT, UPDATE, DELETE, TRUNCATE e COPY FROM. Também são registradas as instruções PREPARE e EXPLAIN ANALYZE, se os comandos contidos nestas instruções forem do tipo apropriado.

Arquivo pg_hba.conf

A autenticação do cliente é controlada pelo arquivo pg_hba.conf e é armazenado no diretório de dados do agrupamento de Bancos de Dados. HBA significa autenticação baseada no hospedeiro (*host-based authentication*).

O formato geral do arquivo pg_hba.conf é um conjunto de registros, sendo um por linha. As linhas em branco são ignoradas, da mesma forma que qualquer texto após o caractere de comentário #. Um registro é formado por vários campos separados por espaços ou tabulações. Os campos podem conter espaços em branco se o valor do campo estiver entre aspas. Os registros não podem ocupar mais de uma linha.

Cada registro especifica um tipo de conexão, uma faixa de endereços de IP de cliente, um nome de Banco de Dados, um nome de usuário e o método de autenticação a ser utilizado nas conexões que correspondem a estes parâmetros. O primeiro registro com o tipo de conexão, endereço do cliente, Banco de Dados solicitado e nome de usuário que corresponder é utilizado para realizar a autenticação.

Arquivo recovery.conf

O SGBD PostgreSQL utiliza um arquivo texto chamado RECOVERY.CONF para carregar parâmetros de restauração de Banco de Dados. A parte chave de todo este

procedimento é a definição do arquivo contendo o comando de recuperação, que descreve como se deseja fazer a recuperação, e até onde a recuperação deve prosseguir.

O único parâmetro requerido no arquivo é *restore_command*, que informa ao PostgreSQL como trazer de volta os arquivos de segmento do WAL copiados. Como no *archive_command*, este parâmetro é uma cadeia de caracteres para o interpretador de comandos.

A tabela abaixo mostra alguns dos principais parâmetros utilizados pelo arquivo RECOVERY.CONF.

Comando	**Significado**
restore_command	O comando, para o interpretador de comandos, a ser executado para trazer de volta os segmentos da série de arquivos do WAL guardados. É importante que o comando retorne o *status* de saída zero se, e somente se, for bem-sucedido.
recovery_target_time	Este parâmetro especifica o tempo até onde a recuperação deve prosseguir. O padrão é recuperar até o fim do WAL.
recovery_target_xid	Este parâmetro especifica o identificador de transação até onde a recuperação deve prosseguir. Deve-se ter em mente que enquanto os identificadores são atribuídos seqüencialmente no início da transação, as transações podem ficar completas em uma ordem numérica diferente. As transações que serão recuperadas são aquelas que foram efetivadas antes e opcionalmente incluindo a transação especificada.