Ciências da Computação

Disciplina

Banco de Dados 2

Conteúdo

Programação de Banco de Dados. Funções, gatilhos e procedimentos armazenados. Organização de Dados e Estruturas de Armazenamento. Transações. Controle de concorrência. Recuperação após falhas. Segurança.

Termos

Controle de concorrência

Abort

Bloqueio

Arquivo

Índice

Stored procedure

Árvore B+

Tabela

Deadlock

Recuperação após falhas

Atomicidade

Rollback

Chave primária

Relacionamento

Isolamento

SQL

Álgebra relacional

Commit

Restrição de integridade

Consistência

Trigger

Hashing

Restrição de integridade

Durabilidade

Processamento de consultas

Log

Entidade

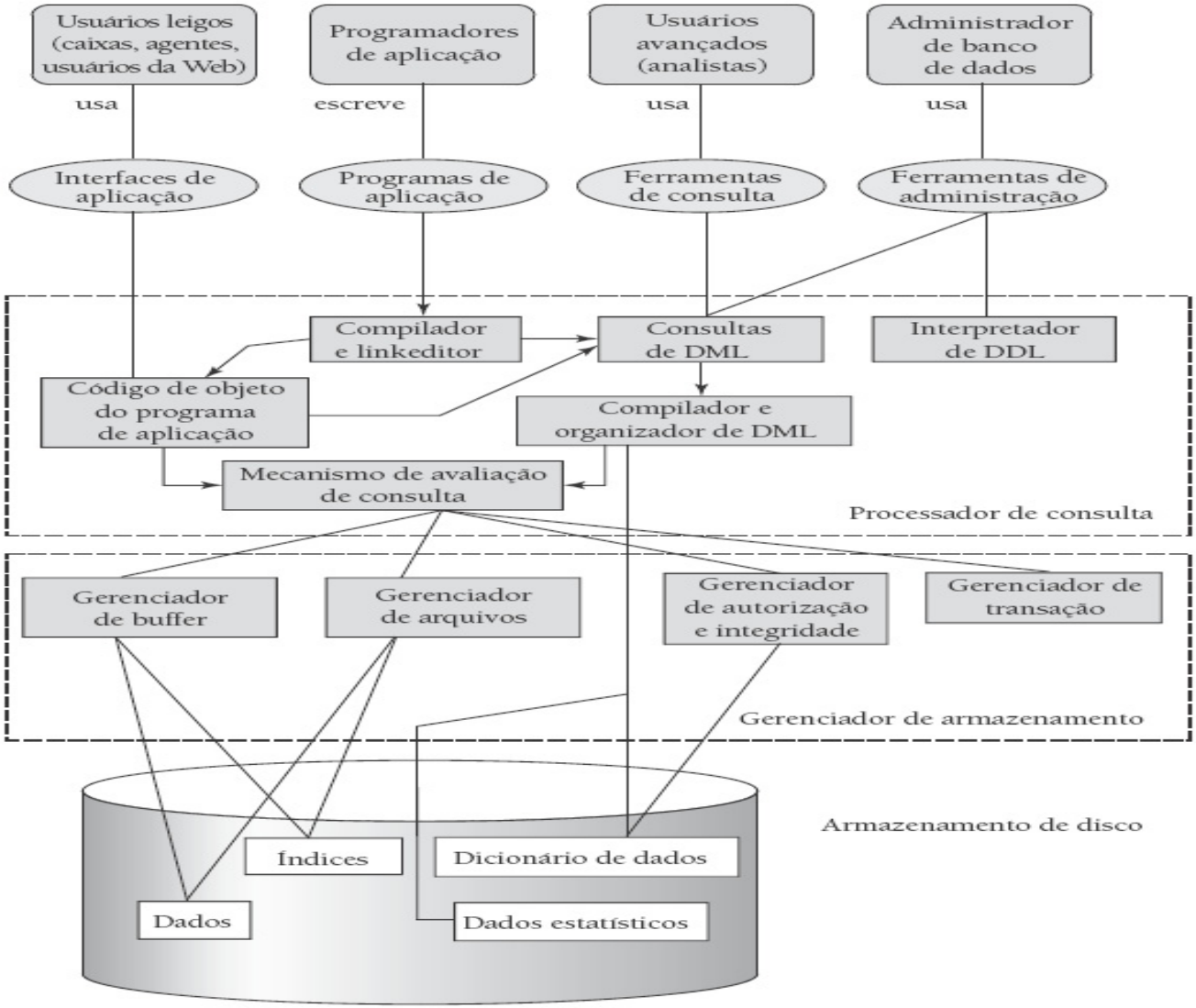
Serialização

Chave estrangeira

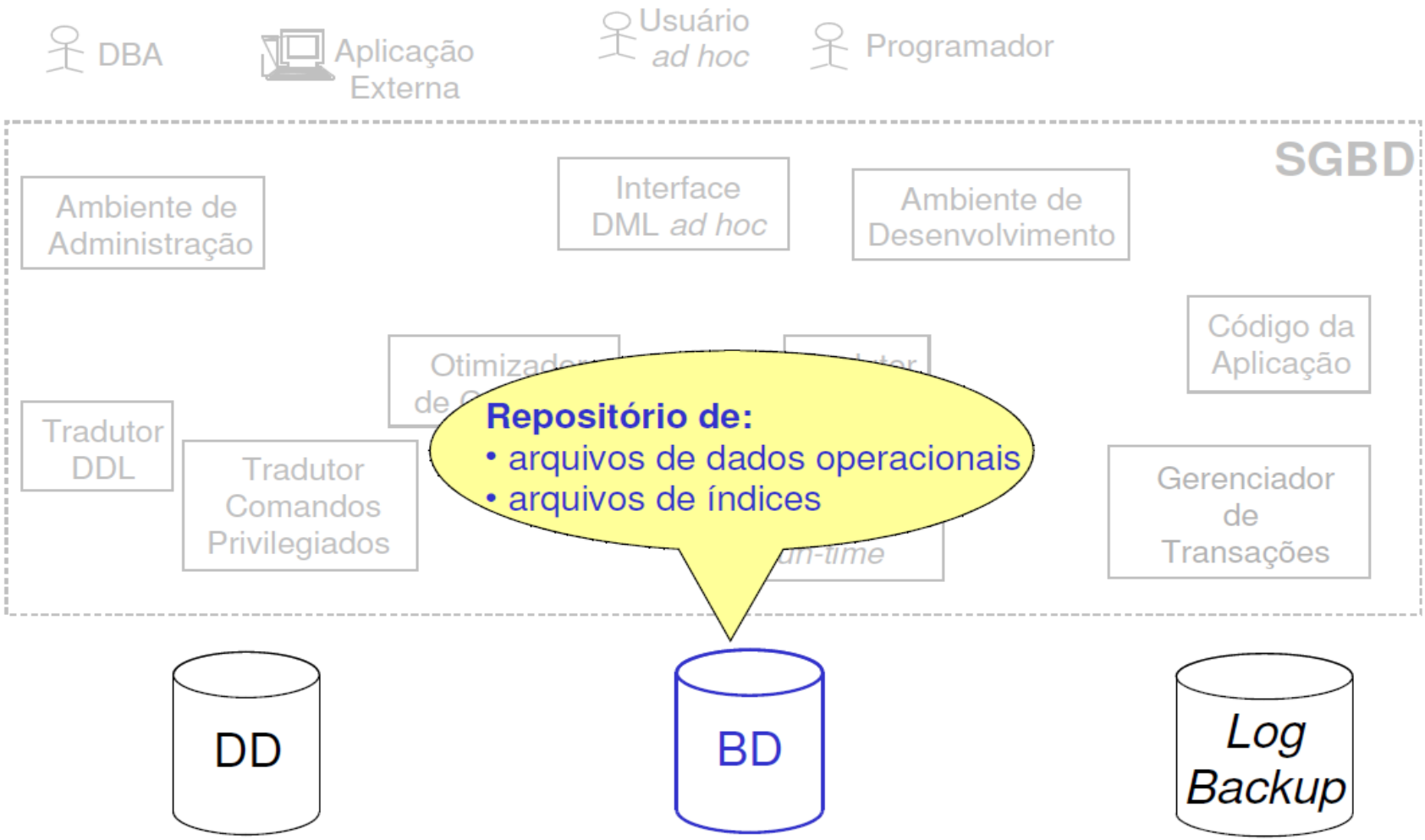
Escalonamento

Transação

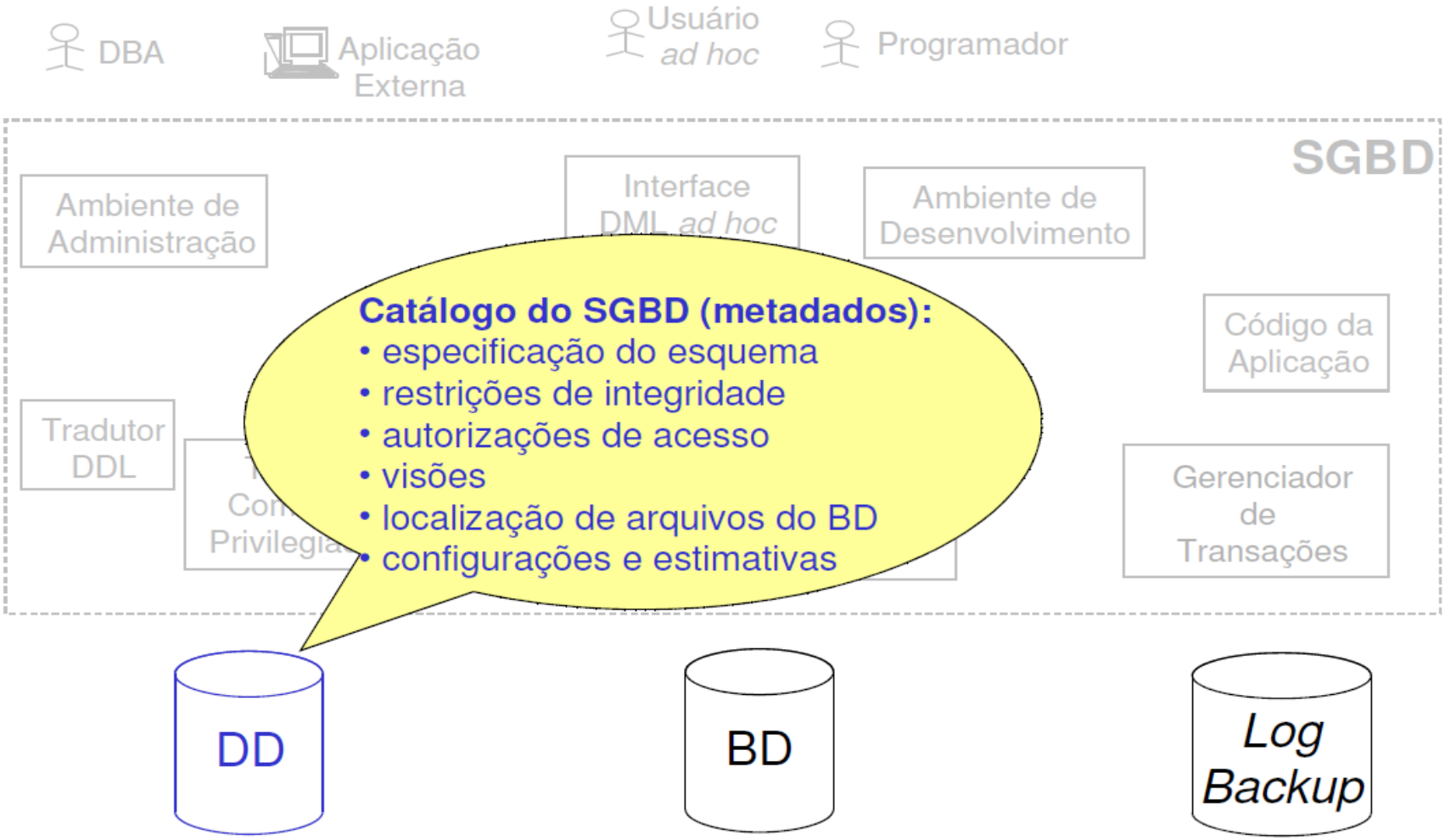
Estrutura Geral de um SGBD



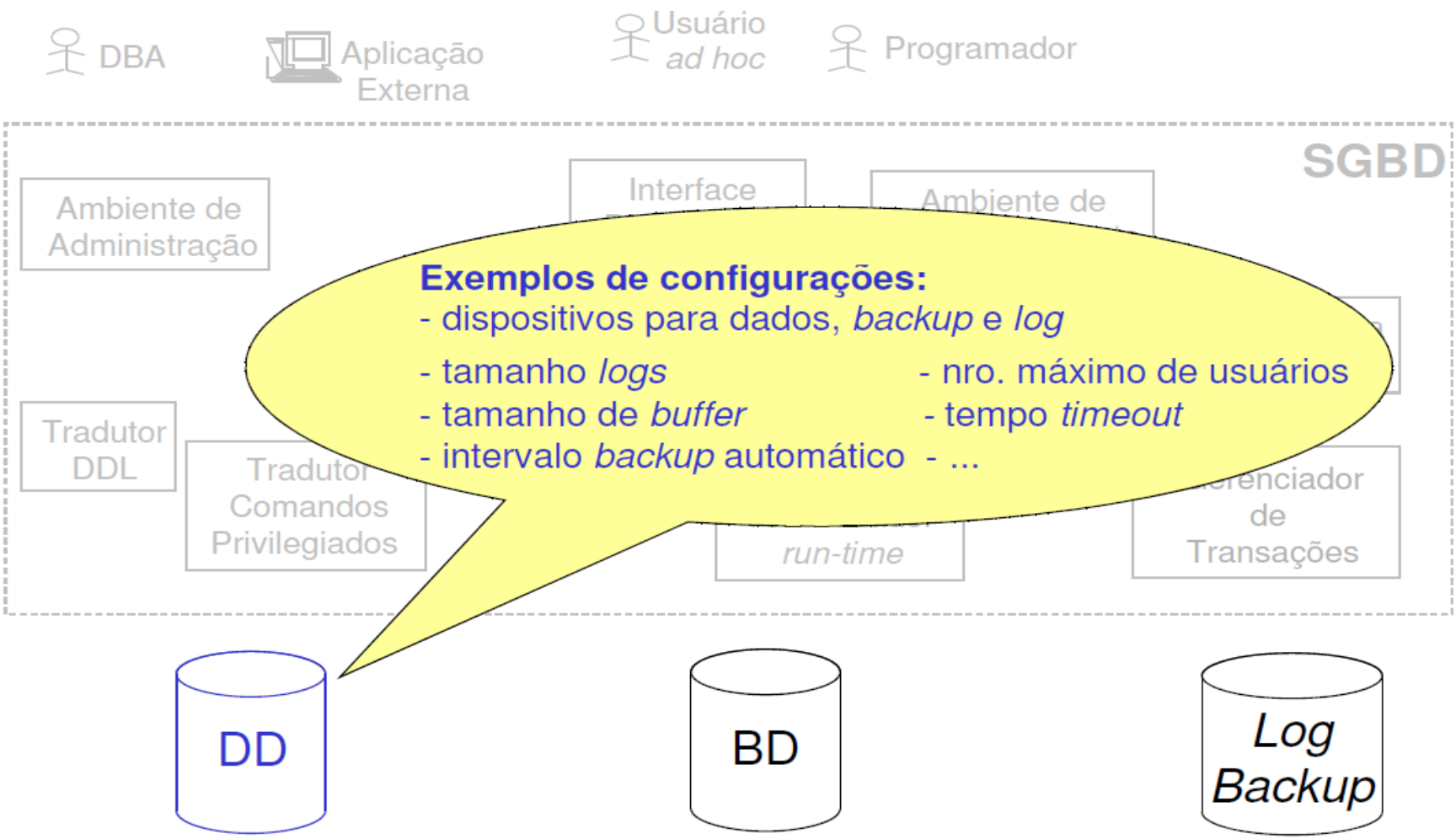
Meios de Armazenamento



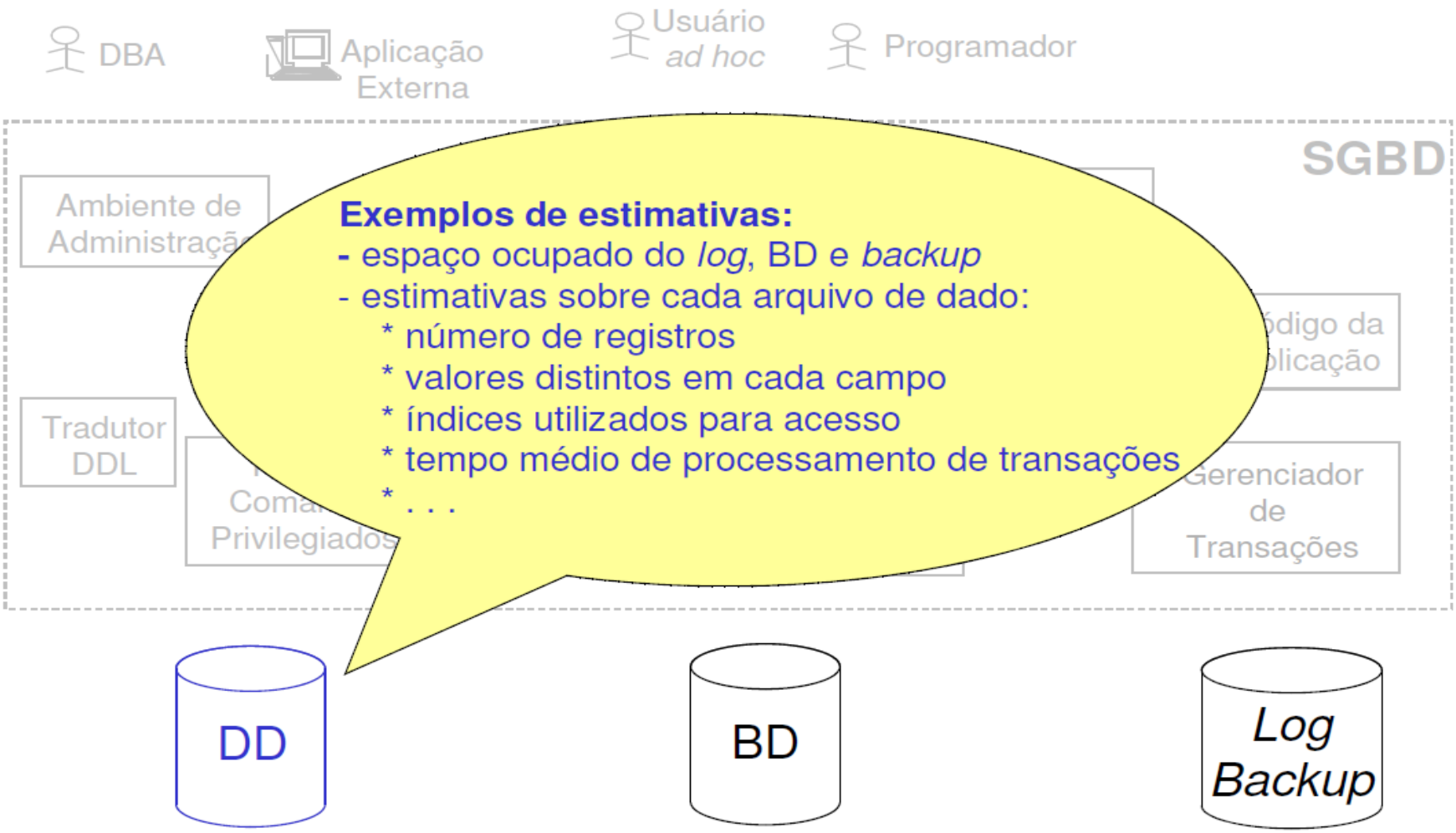
Meios de Armazenamento



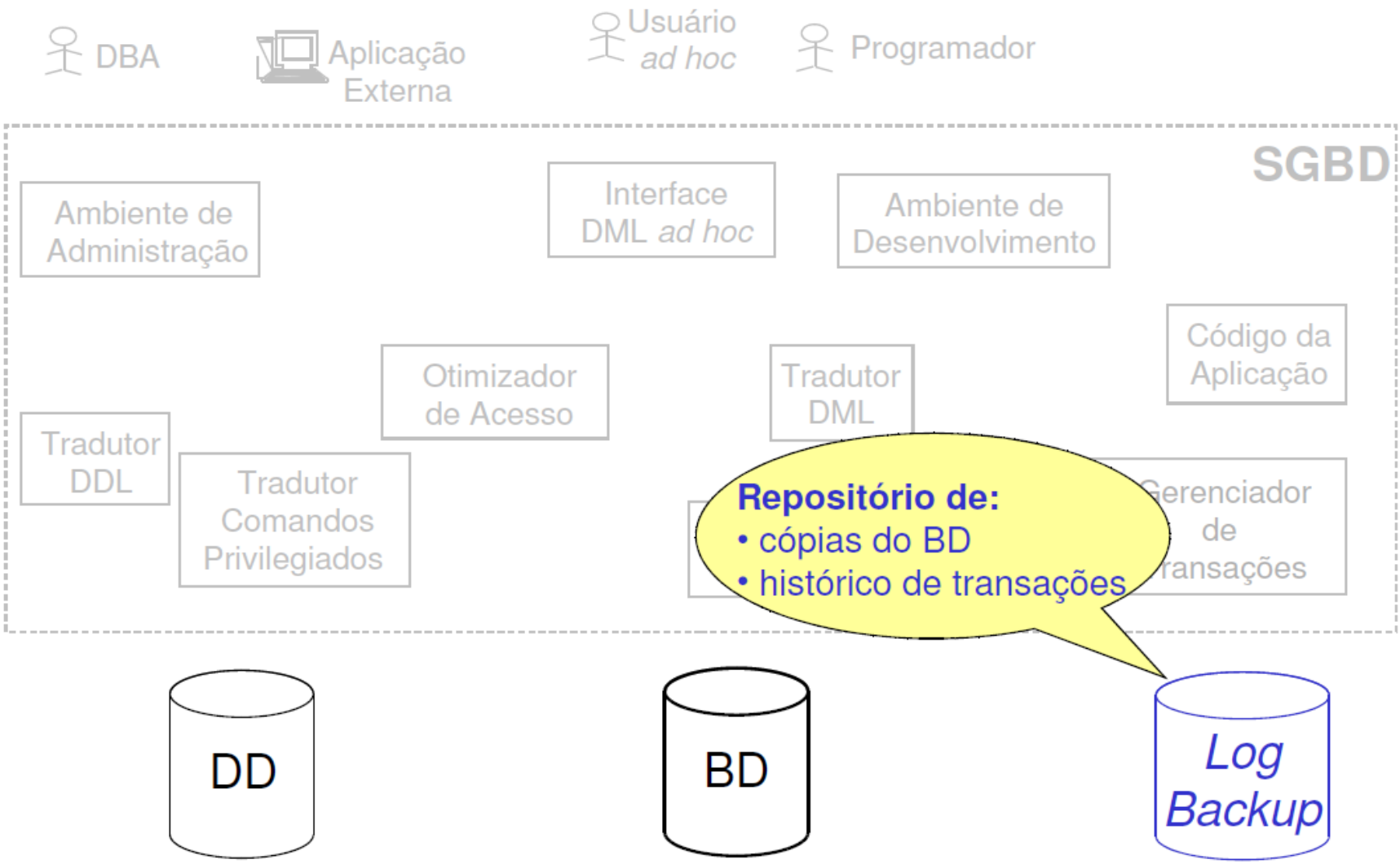
Meios de Armazenamento



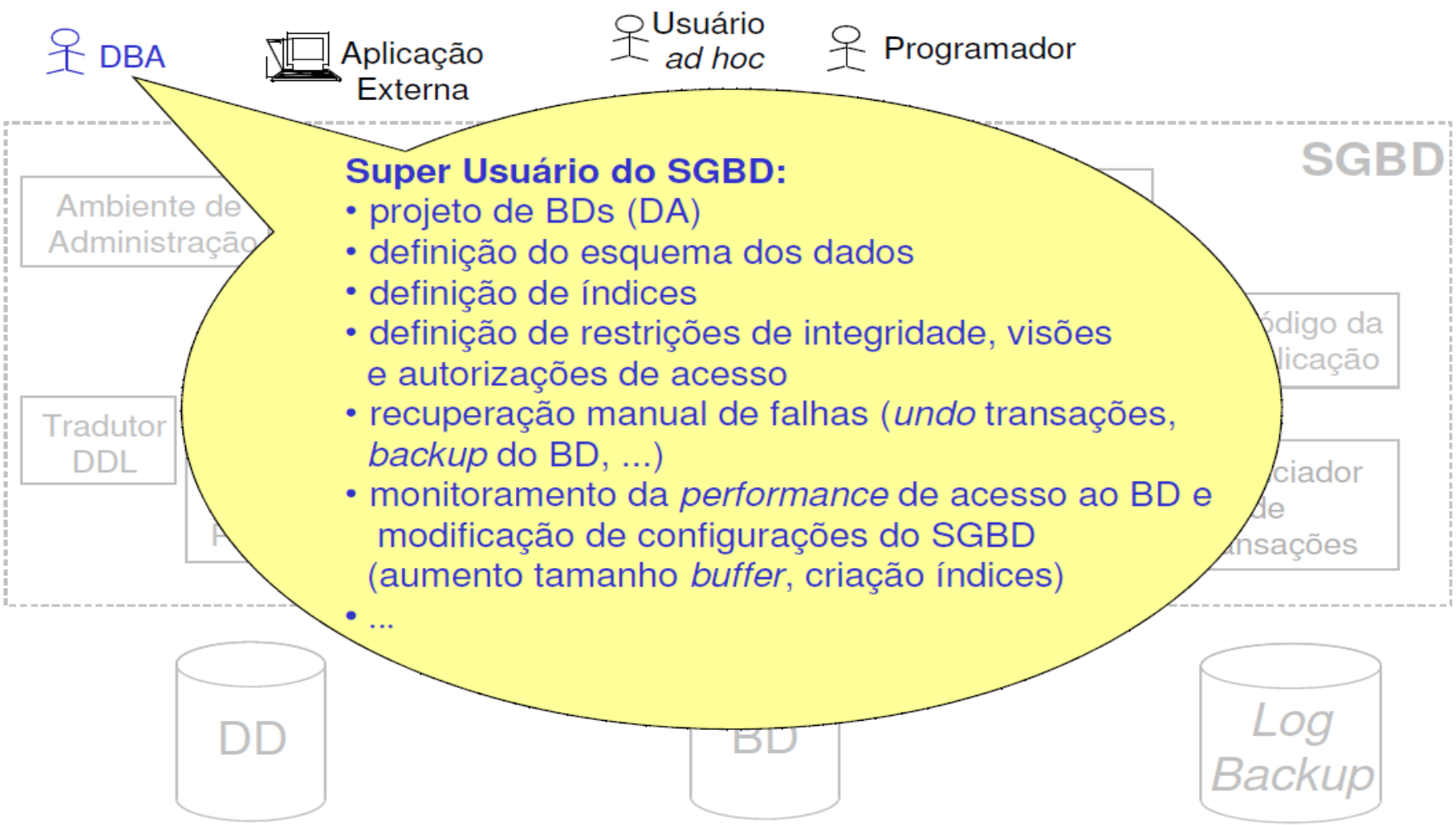
Meios de Armazenamento



Meios de Armazenamento



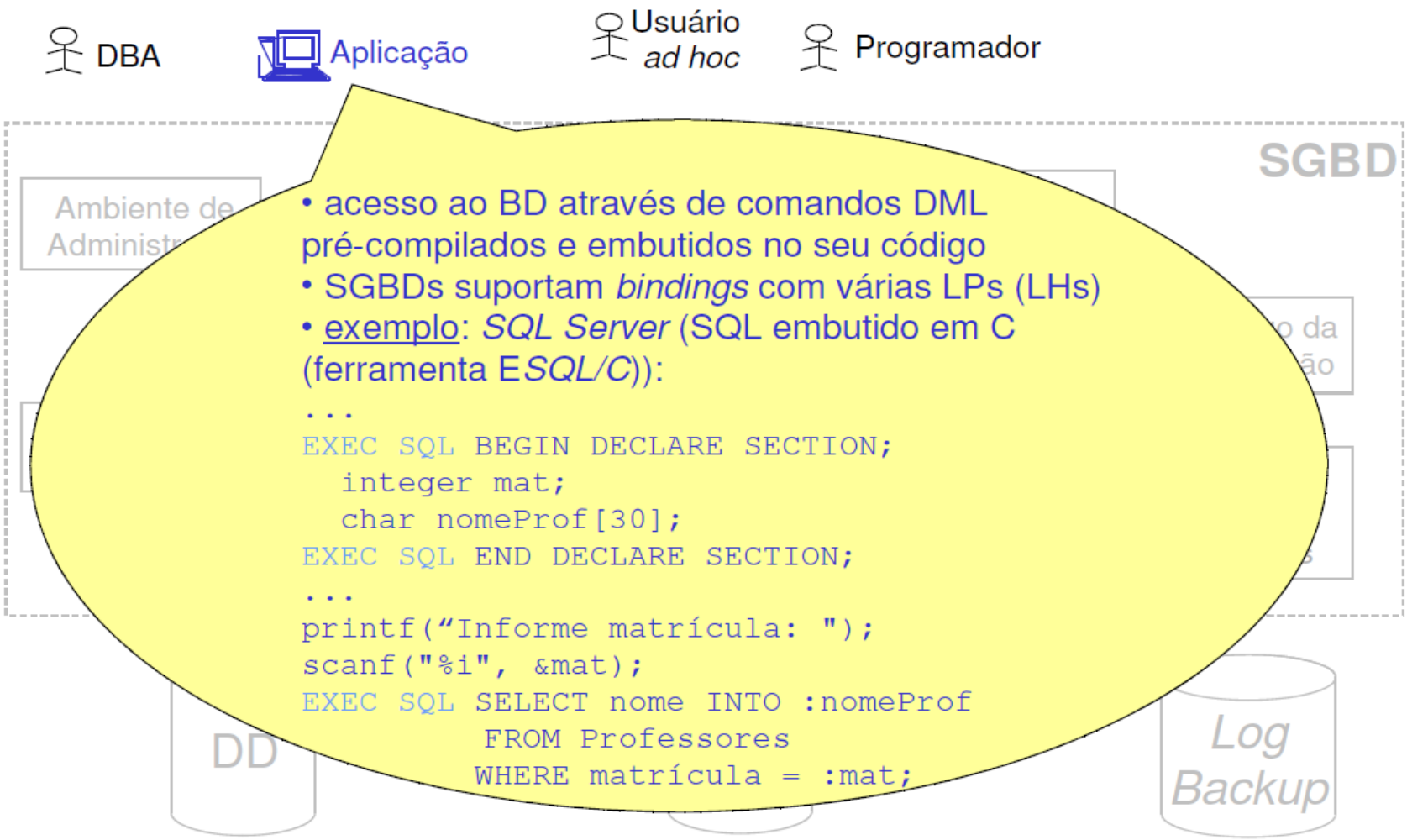
Usuários do SGBD



Super Usuário do SGBD:

- projeto de BDs (DA)
- definição do esquema dos dados
- definição de índices
- definição de restrições de integridade, visões e autorizações de acesso
- recuperação manual de falhas (*undo* transações, *backup* do BD, ...)
- monitoramento da *performance* de acesso ao BD e modificação de configurações do SGBD (aumento tamanho *buffer*, criação índices)
- ...

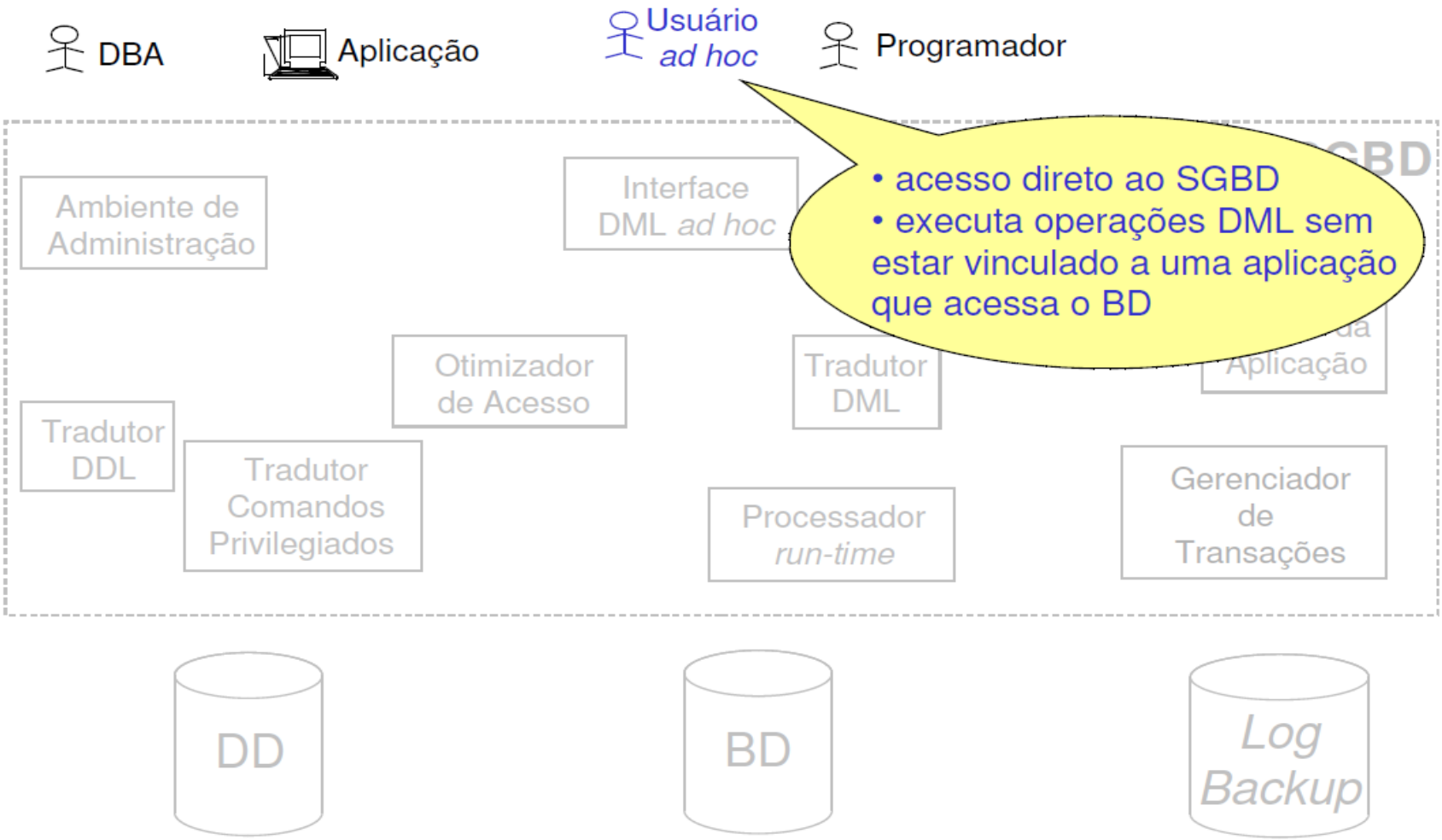
Usuários do SGBD



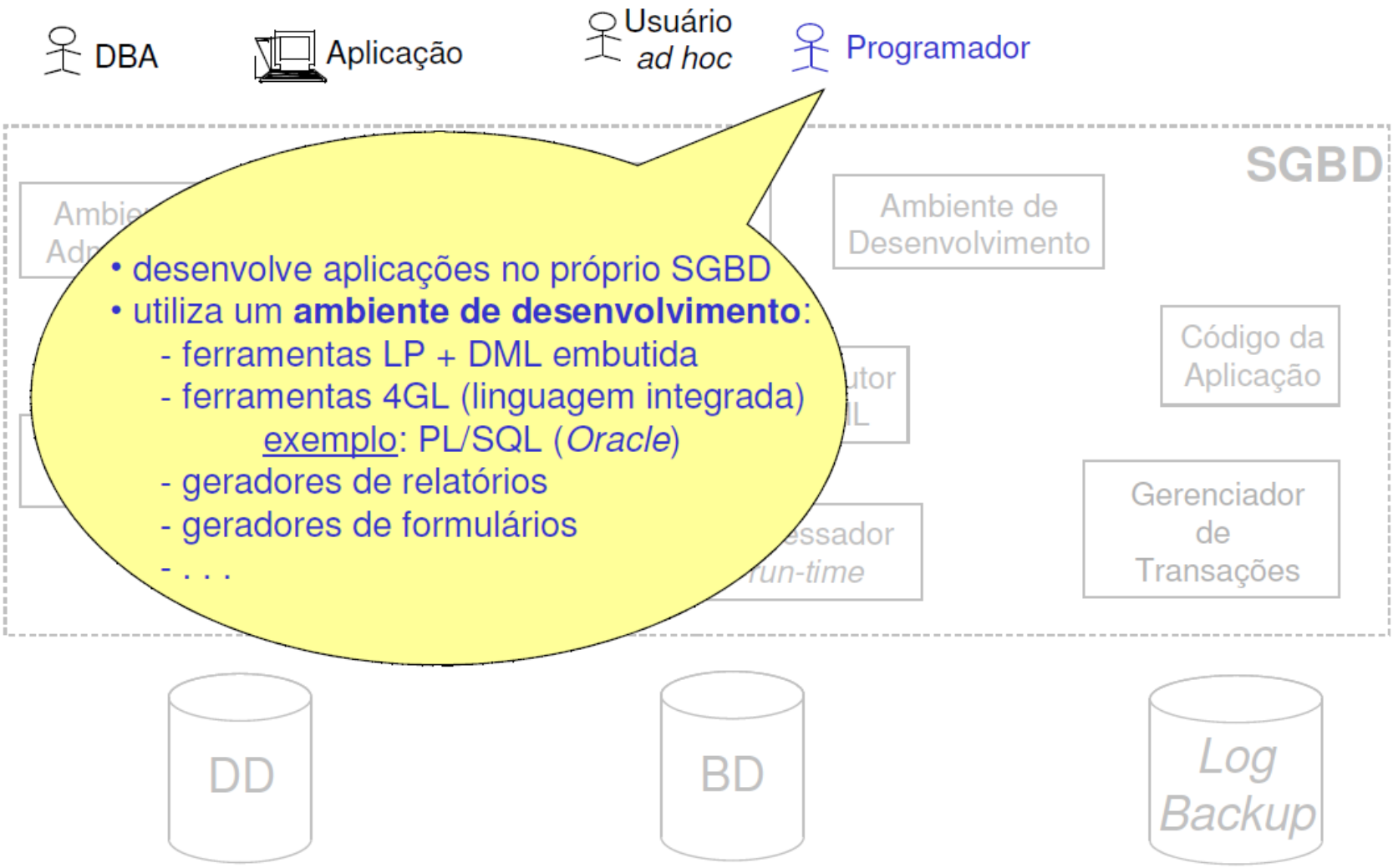
- acesso ao BD através de comandos DML pré-compilados e embutidos no seu código
- SGBDs suportam *bindings* com várias LPs (LHs)
- exemplo: *SQL Server* (SQL embutido em C (ferramenta E*SQL/C*)):

```
...
EXEC SQL BEGIN DECLARE SECTION;
  integer mat;
  char nomeProf[30];
EXEC SQL END DECLARE SECTION;
...
printf("Informe matrícula: ");
scanf("%i", &mat);
EXEC SQL SELECT nome INTO :nomeProf
         FROM Professores
         WHERE matrícula = :mat;
```

Usuários do SGBD



Usuários do SGBD



Algumas Funções do SGBD

Controle de concorrência

Processamento de Consultas

Segurança

Recuperação Após Falhas

Controle de Armazenamento

...

Métodos de Acesso

DDL (*Data Definition Language*)

>> especificação do esquema do BD (dados e seus tipos de dados, índices, ...)

DML (*Data Manipulation Language*)

>> manipulação de dados (I, A, E, C)

Processamento eficaz de consultas

>> considera relacionamentos, predicados de seleção, volume de dados, índices, ...
>> exemplo: buscar professores que lecionam em turmas lotadas em salas do quarto andar.

Processamento de Consultas

Extração de informações do BD

Consulta SQL

– adequada para uso humano

– não adequada para processamento pelo SGBD

>> não descreve uma sequência de passos (procedimento) a ser seguida
>> não descreve uma estratégia eficiente para a implementação de cada passo no que diz respeito ao acesso a nível físico (arquivos do BD)

O SGBD deve se preocupar com este processamento!

>> módulo Processador de Consultas

Módulo Processador de Consultas

Objetivo
– otimização do processamento de uma consulta

Envolve:
>> tradução, transformação e geração de uma estratégia (plano) de execução.

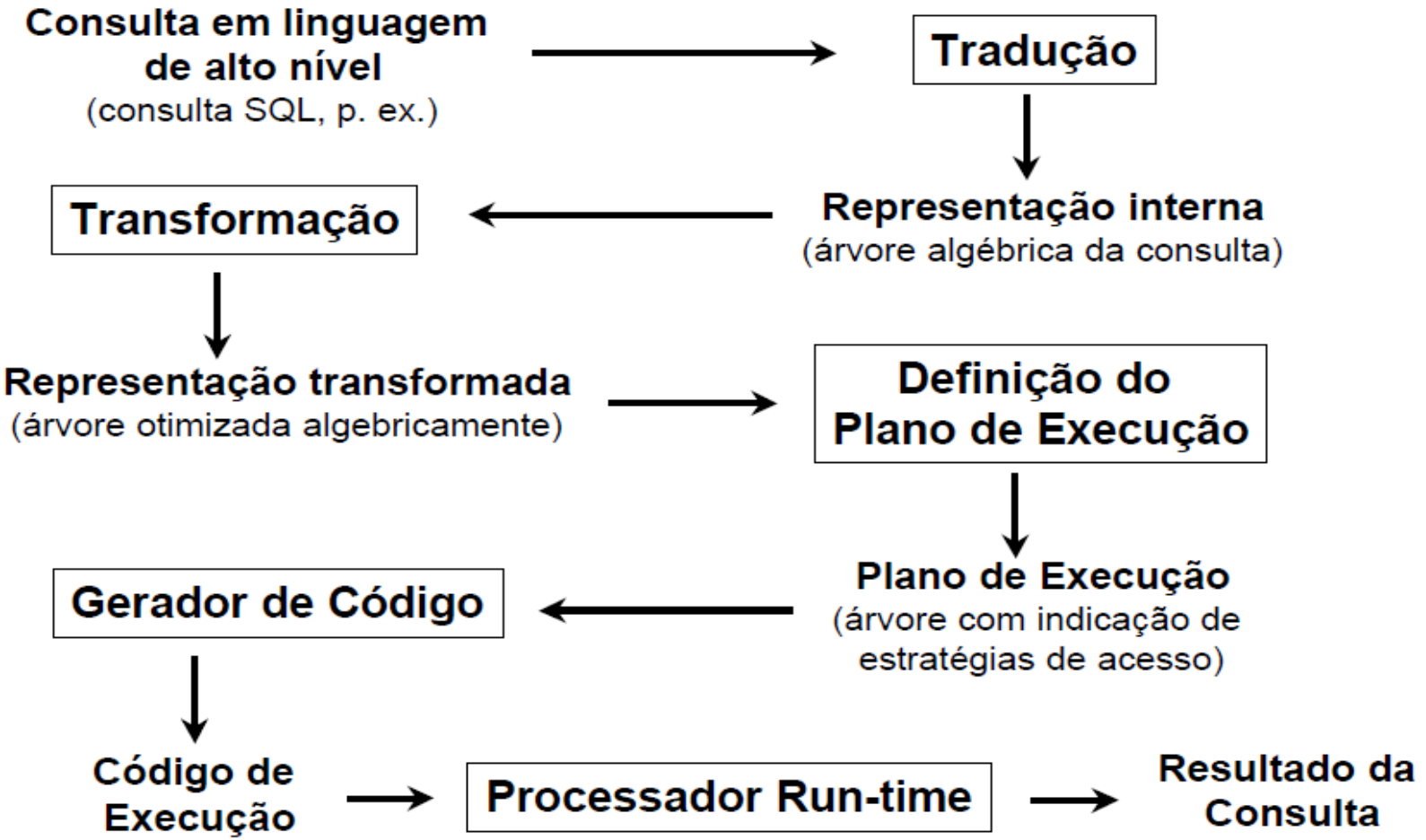
Estratégia de acesso
>> leva em conta algoritmos predefinidos para implementação de passos do processamento e estimativas sobre os dados.

Vale a pena todo este esforço? Sim!

– *Tx* = tempo para definir e executar uma estratégia otimizada de processamento
– *Ty* = tempo para executar uma estratégia não otimizada de processamento

Quase sempre: ***Tx*** **<<** ***Ty***

Etapas de Processamento de Consultas



Segurança

Objetivos primários:

>> evitar: (1) violação de consistência dos dados
(2) perda de dados

Segurança de acesso (usuários e aplicações)

>> matrizes de autorização
>> visões

Segurança contra falhas (*recovery*)

>> monitoração de transações
>> categorias de falhas: transação, sistema, meio de armazenamento ...
>> manutenção de histórico de atualizações (*logs*) e *backups* do BD

Log do Sistema

Transação (transferência bancária)

```
begin transaction

update Contas
set saldo = saldo - 50.00
where número = 100

update Contas
set saldo = saldo + 50.00
where número = 200

commit transaction
```

```
...
<begin transaction T256>
...
<T256,update,Contas,100,
 500.00,450,00>
...
<T256,update,Contas,200,
 350.00,400,00>
...
<end transaction T256>
...
```

Arquivo de *Log*

Transação

Requisitos que devem ser atendidos.

Propriedades ACID.

Atomicidade
Consistência
Isolamento
Durabilidade ou Persistência

Atomicidade

Contas

número	saldo	
1001-2	500.00	← x
2002-3	200.00	← y
. . .		

T_x (transferência bancária)

```
read(x)
x.saldo = x.saldo – 100.00
write(x)
read(y)
y.saldo = y.saldo + 100.00
write(y)
```

falha!

execução

Isolamento

T_1	T_2
read(A)	
A = A – 50	
write(A)	
	read(A)
	A = A+A*0.1
	write(A)
read(B)	
B = B + 50	
write(B)	
	read(B)
	B = B - A
	write(B)

escalonamento válido

T_1	T_2
read(A)	
A = A – 50	
	read(A)
	A = A+A*0.1
	write(A)
	read(B)
write(A) ←	
read(B)	
B = B + 50	
write(B)	
	B = B - A
	write(B) ←

T_1 interfere em T_2

T_2 interfere em T_1

escalonamento inválido

Atualização Perdida

Exemplo: uma transação **Ty** grava em um dado atualizado por uma transação **Tx**.

T1	T2
read(X)	
X = X – 20	
	read(Z)
	X = Z + 10
write(X)	
read(Y)	
	write(X)
Y = X + 30	
write(Y)	

a atualização de X por T1 foi perdida!

Leitura Suja

Exemplo: a transação **Tx** atualiza um dado **X**, outras transações posteriormente fazem a leitura **X**, mas depois **Tx** falha.

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
abort( )	

T2 leu um valor de X que não será mais válido!

Serialização

Um escalonamento não-serial de um conjunto de transações deve produzir resultado equivalente a alguma execução serial destas transações.

entrada:
X = 50
Y = 40

execução serial

saída:
X = 40
Y = 60

T1	T2
read(X)	
X = X – 20	
write(X)	
read(Y)	
Y = Y + 20	
write(Y)	
	read(X)
	X = X + 10
	write(X)

entrada:
X = 50
Y = 40

execução não-serial ***serializável***

saída:
X = 40
Y = 60

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
Y = Y + 20	
write(Y)	

Controle de Concorrência

Objetivo primário:

>> evitar conflitos de acesso simultâneo a dados por transações.

Principais técnicas

>> bloqueio (*lock*)\
>> *timestamp*

Técnicas Baseadas em Bloqueio

Conjunto de transações executando concorrentemente é muito dinâmico:

- novas transações estão sendo constantemente submetidas ao SGBD para execução.

Princípio de funcionamento:

- controle de operações *read(X)* e *write(X)* e postergação (através de bloqueio) de algumas dessas operações de modo a evitar conflito.

Todo dado possui um status de bloqueio:

- liberado (*Unlocked - U*);
- com bloqueio compartilhado (*Shared lock - S*);
- com bloqueio exclusivo (*eXclusive lock - X*).

Modos de Bloqueio - Exemplo

T1	T2
lock-S(Y)	
read(Y)	
unlock(Y)	
	lock-S(X)
	lock-X(Y)
	read(X)
	read(Y)
	unlock(X)
	write(Y)
	unlock(Y)
	commit( )
lock-X(X)	
read(X)	
write(X)	
unlock(X)	
commit( )	

Deadlock (Impasse) de Transações

Técnica esperar-ou-morrer (*wait-die*) – baseada em *timestamp*:

se ***TS(Tx) < TS(Ty)*** então
    ***Tx* espera**
senão
    início
        abort(*Tx*)
        start(*Tx*) com o mesmo *TS*
    fim

Técnica ferir-ou-esperar (*wound-wait*) – baseada em *timestamp*:

se ***TS(Tx) < TS(Ty)*** então
    início
        abort(*Ty*)
        start(*Ty*) com o mesmo *TS*
    fim
senão
    ***Tx* espera**

Recuperação Após Falhas

Garantia de atomicidade e durabilidade de Transações

- requer um SGBD tolerante a falhas.

Tolerância a falhas em Bds:

- capacidade de conduzir o BD a um estado passado consistente, após a ocorrência de uma falha que o deixou em um estado inconsistente;

- baseia-se em redundância de dados;

- não é um mecanismo 100% seguro;

- responsabilidade do subsistema de *recovery* do SGBD.

Exemplo de *Log*

Log

```
<start T3>
<write T3,B,15,12>
<start T2>
<write T2,B,12,18>
<start T1>
<write T1,D,20,25>
<commit T1>
<write T2,D,25,26>
<write T3,A,10,19>
<commit T3>
<commit T2>
...
```

read(A)
read(D)
write(D)

T_2

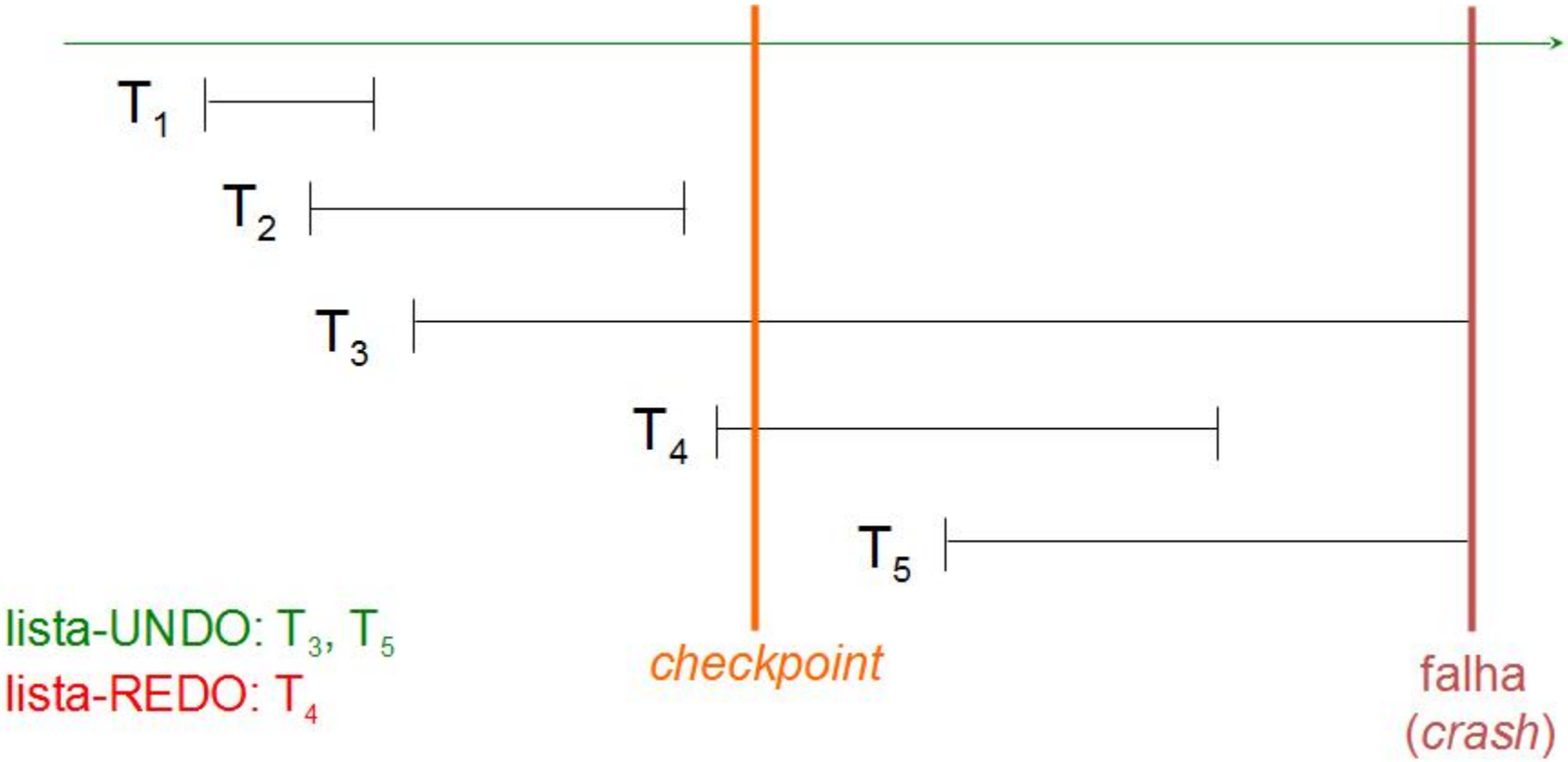
read(B)
write(B)
read(D)
write(D)

T_3

read(C)
write(B)
read(A)
write(A)

Técnica UNDO/REDO com *Checkpoint*

Transações *committed* antes do checkpoint não precisam sofrer REDO em caso de falha, pois já estão garantidamente no BD.



Organização de Registros

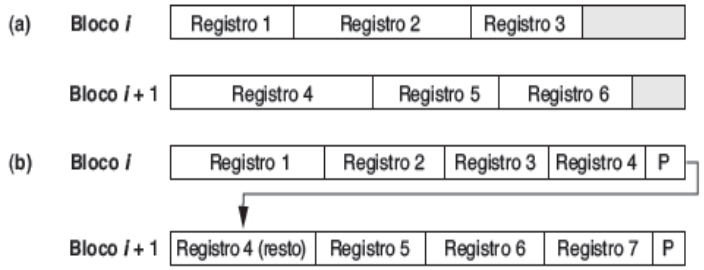


Figura 17.6
Tipos de organização de registro. (a) Não espalhada. (b) Espalhada.

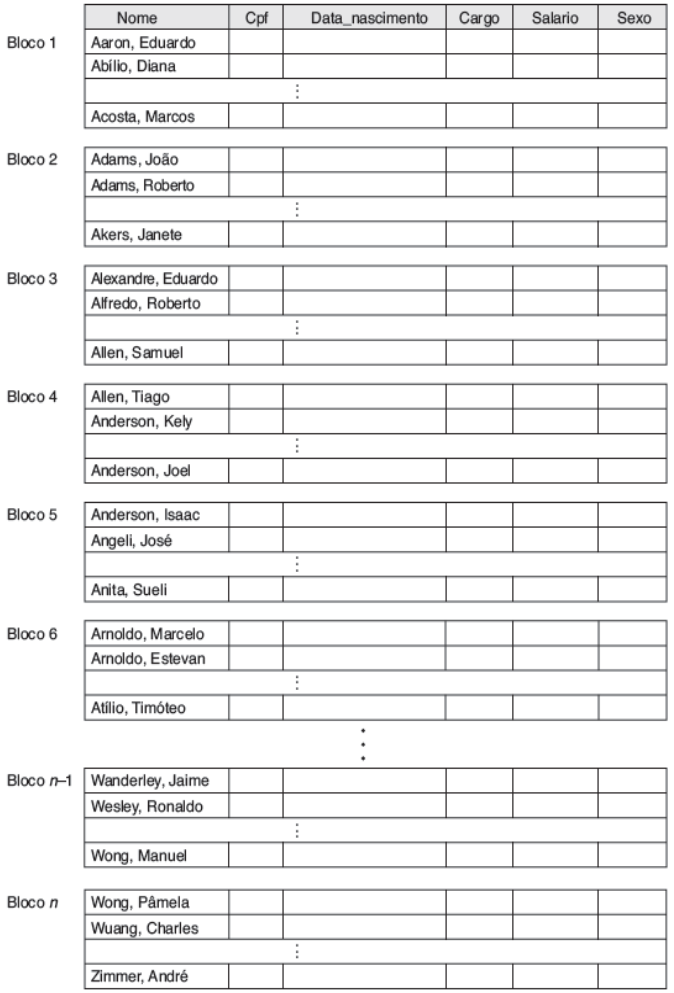


Figura 17.7
Alguns blocos de um arquivo ordenado (sequencial) de registros de FUNCIONARIO com Nome como campo-chave de ordenação.

Organização de Registros

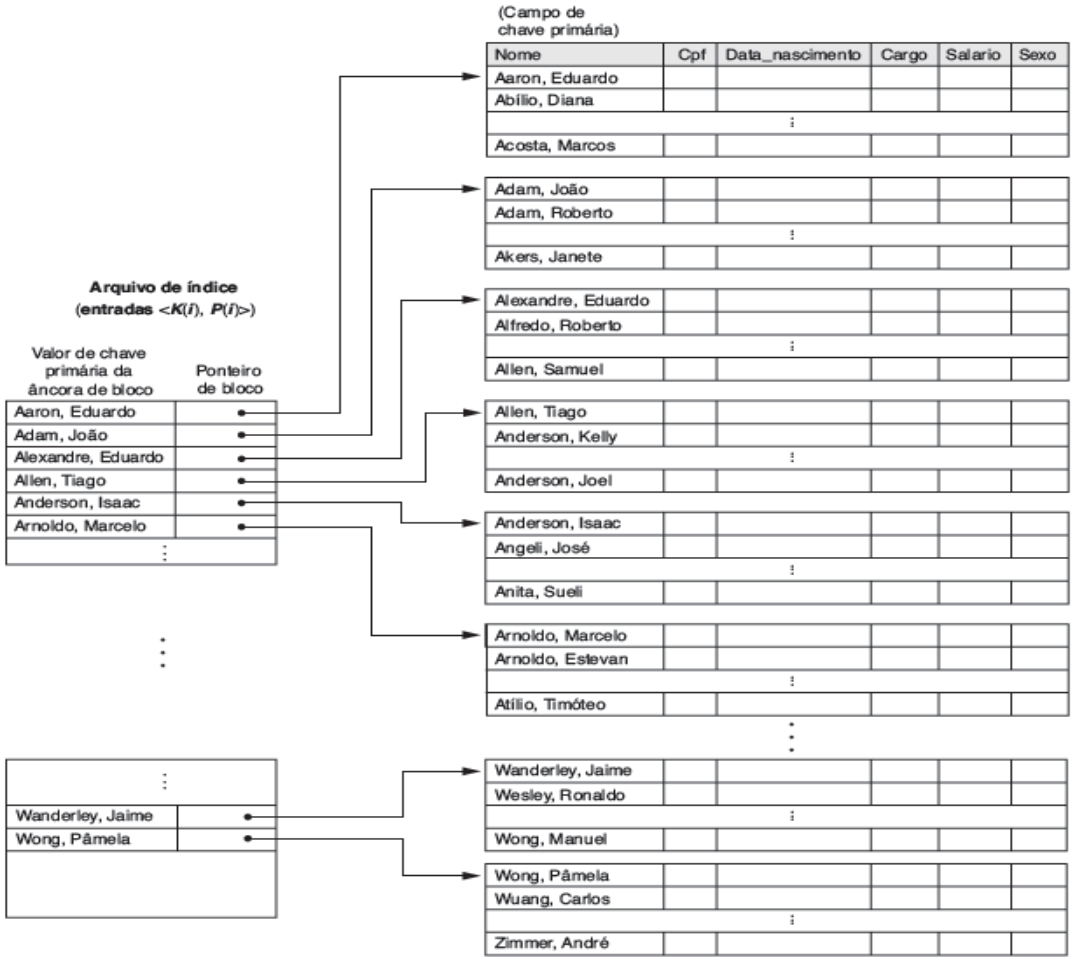


Figura 18.1
Índice primário no campo de chave de ordenação do arquivo mostrado na Figura 17.7.

Organização de Registros

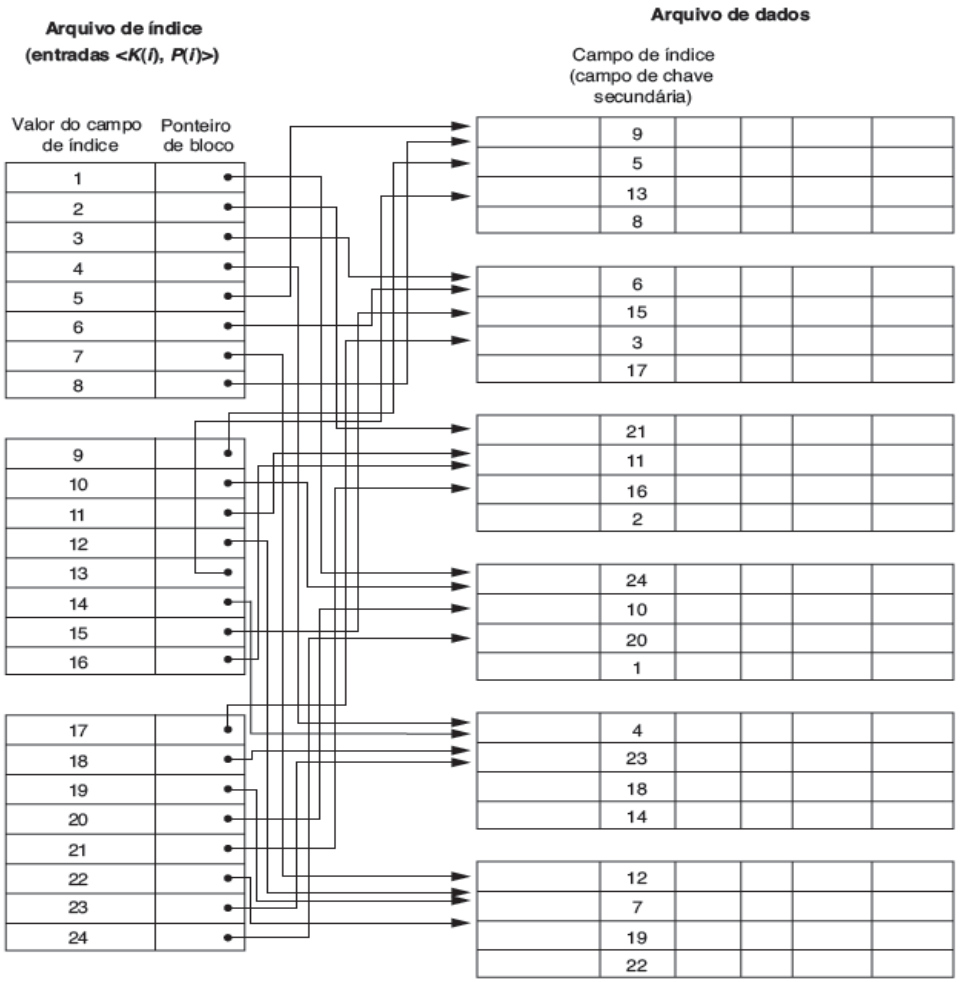
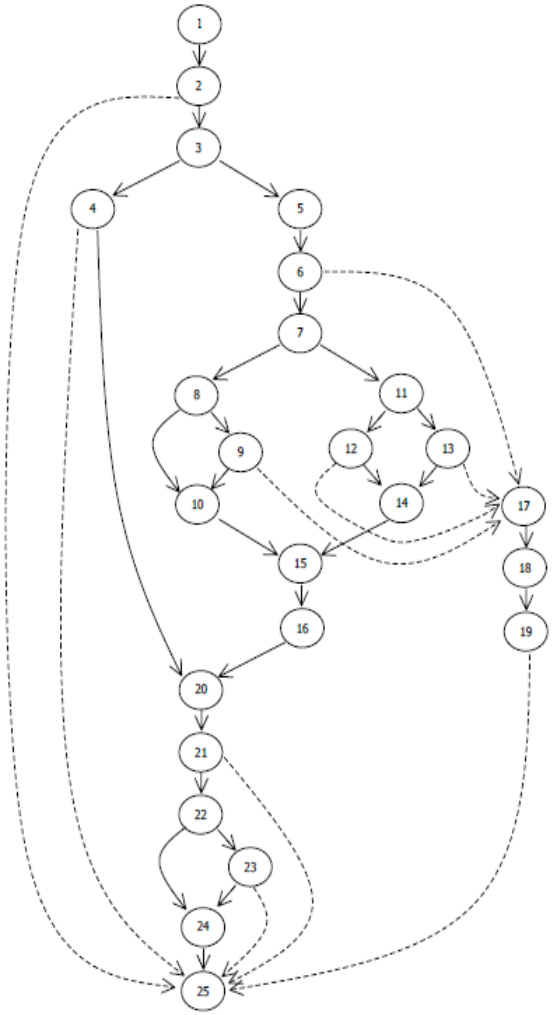


Figura 18.4
Um índice secundário denso (com ponteiros de bloco) em um campo de chave não ordenado de um arquivo.

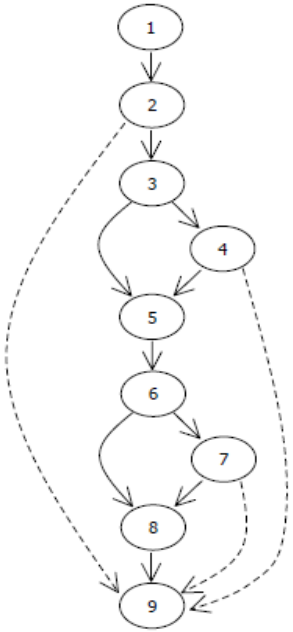
Gatilhos (*triggers*)

```
*01* CREATE OR REPLACE TRIGGER trg001
*01*     AFTER INSERT OR DELETE OR UPDATE ON tab001
*01*     FOR EACH ROW
*01* DECLARE v_exists INTEGER;
*01* BEGIN
*02*      UPDATE tab003
*02*      SET    att003_1 = att003_1 + NVL(:new.att001_2, 0) - NVL(:old.att001_2, 0)
*02*      WHERE  key003 = NVL(:new.key001, 0)
*02*      OR     key003 = NVL(:old.key001, 0);
*03*      IF DELETING THEN
*04*         DELETE FROM tab002
*04*         WHERE  key002 = :old.key001;
*05*      ELSE
*05*         BEGIN
*06*            SELECT COUNT(*) INTO v_exists
*06*            FROM   tab002
*06*            WHERE  key002 = :new.key001;
*07*            IF :new.att001_3 >= (:new.att001_2 - :new.att001_1) THEN
*08*               IF v_exists > 0 THEN
*09*                  DELETE FROM tab002
*09*                  WHERE  key002 = :new.key001;
*10*               END IF;
*11*            ELSE
*11*               IF v_exists > 0 THEN
*12*                  UPDATE tab005
*12*                  SET    att005_1 = :new.att001_2 - :new.att001_1 -:new.att001_3
*12*                  WHERE  key005 = :new.key001;
*13*               ELSE
*13*                  INSERT INTO tab002
*13*                     (key002, att002_1)
*13*                  VALUES
*13*                     (:new.key001, :new.att001_2 - :new.att001_1 -:new.att001_3);
*14*               END IF;
*15*            END IF;
*17*         EXCEPTION
*18*             WHEN OTHERS THEN
*19*                 RAISE_APPLICATION_ERROR (-20002,'Ins/upd error: ' || sqlerrm);
*16*         END;
*20*      END IF;
*21*      SELECT COUNT(*) INTO v_exists
*21*      FROM   tab004
*21*      WHERE  key004 = NVL(:new.key001, 0)
*21*      OR     key004 = NVL(:old.key001, 0);
*22*      IF v_exists > 0 THEN
*23*         UPDATE tab004
*23*         SET    att004_2 = att004_2 + NVL(:new.att001_2, 0) - NVL(:old.att001_2, 0)
*23*         WHERE  key004 = NVL(:new.key001, 0)
*23*         OR     key004 = NVL(:old.key001, 0);
*24*      END IF;
*25* END trg001;
```

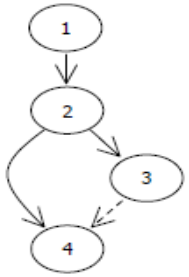


Gatilhos (*triggers*)

```
*01* CREATE OR REPLACE TRIGGER trg002
*01* AFTER INSERT OR DELETE OR UPDATE ON tab002
*01* FOR EACH ROW
*01* DECLARE v_count INTEGER;
*01* BEGIN
*02*       SELECT COUNT(*) INTO v_count
*02*       FROM   tab003
*02*       WHERE  key003 = NVL(:new.key002, 0)
*02*       OR     key003 = NVL(:old.key002, 0);
*03*       IF DELETING OR UPDATING THEN
*04*          UPDATE tab004
*04*          SET att004_1  = att004_1 * ( 1 + v_count / 10 )
*04*          WHERE key004 = :old.key002;
*05*       END IF;
*06*       IF INSERTING OR UPDATING THEN
*07*          UPDATE tab004
*07*          SET att004_1  = att004_1 * ( 1 - v_count / 10 )
*07*          WHERE key004 = :new.key002;
*08*       END IF;
*09* END trg002;
```




```
*01* CREATE OR REPLACE TRIGGER trg005
*01* BEFORE INSERT OR UPDATE ON tab005
*01* FOR EACH ROW
*02* WHEN (EXISTS(SELECT key003 FROM tab003 WHERE att003_1 IS NULL))
*03* BEGIN
*03*       RAISE_APPLICATION_ERROR (-20003,'Ins/upd error: ' || sqlerrm);
*04* END trg005;
```



Stored procedures

```
CREATE PROCEDURE proc_WHILE (IN param1 INT)
BEGIN
   DECLARE variable1, variable2 INT;
   SET variable1 = 0;

   WHILE variable1 < param1 DO
      INSERT INTO table1 VALUES (param1);
      SELECT COUNT(*) INTO variable2 FROM table1;
      SET variable1 = variable2 + 1;
   END WHILE;
END
```


Stored procedures

```
CREATE PROCEDURE `proc_CURSOR` (OUT param1 INT)
BEGIN
   DECLARE a, b, c INT;
   DECLARE cur1 CURSOR FOR SELECT col1 FROM table1;
   DECLARE CONTINUE HANDLER FOR NOT FOUND SET b = 1;
   OPEN cur1;

   SET b = 0;
   SET c = 0;

   WHILE b = 0 DO
      FETCH cur1 INTO a;
      IF b = 0 THEN
         SET c = c + a;
   END IF;
   END WHILE;

   CLOSE cur1;
   SET param1 = c;

END
```