

SQL

Feliz Gouveia
UFP

SQL Language

English form:

Find the order number, part number, description, date and quantity for all parts ordered from supplier number 797 during 1975.

SQL form:

```
SELECT ORDERNO, ORDERS.PARTNO, DESCRIP, DATE, QTY
FROM ORDERS, PARTS
WHERE ORDERS.PARTNO = PARTS.PARTNO
AND DATE BETWEEN '75
AND SUPPNO = '797';
```

English form:

For each supplier that supplies part number 010007, list the minimum and maximum quoted price for that part number.

SQL form:

```
SELECT SUPPNO, MIN(PRICE), MAX(PRICE)
FROM QUOTES WHERE PARTNO = '010007'
GROUP BY SUPPNO;
```

\$LET C1 BE

```
SELECT NAME, SALARY INTO $X, $Y
FROM EMP WHERE JOB = $Z;
```

\$OPEN C1; /* BINDS VALUE OF Z */

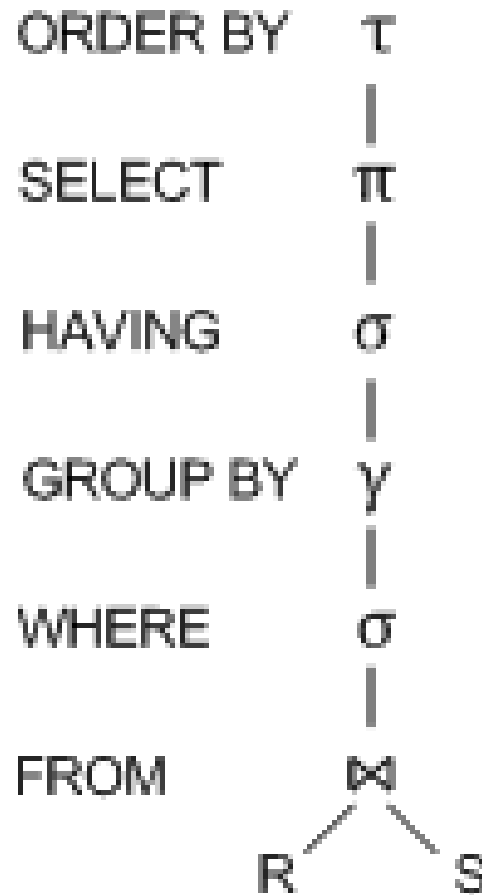
\$FETCH C1; /* FETCHES ONE EMPLOYEE INTO X AND Y */

\$CLOSE C1; /* AFTER ALL VALUES HAVE BEEN FETCHED */

The SQL language

- Declarative language (say “what” not “how”)
- Do not assume any particular execution order; the optimizer will choose the best
- SQL works with bags (duplicate values allowed), relational algebra with sets (no duplicates)
- Current SQL adoption mostly complies with SQL99, also known as SQL3
- PostgreSQL complies with most of SQL:2011

SQL and Relational algebra operators



SQL queries

- Queries return selected rows with projected columns
 - This result can be fed to other queries
 - Most queries are of this type
- Scalar queries: return only one column and one value
 - This result can only be used where a scalar is expected, for example in a test (`=`, `>`, `<`,)

SELECT

- Selects a set of rows (selection) and columns (projection)
- Can use alias for row names and for table names
 - select id as number from student;
 - select id from student s;
 - select * from student;
- Can do computations
 - select name, $\text{ects} * 26$ as credits from course;
- Restrictions in the WHERE clause can be combined logically with AND, OR, NOT

SELECT

- Eliminate duplicates:
 - `select distinct city from student;`
- Do “paging” with LIMIT and OFFSET
 - `select * from student limit 10 offset 0;`
- Can use a SELECT in the FROM clause
 - `select * from (select name, date from student) s where date > now();`
 - This is not the standard way of writing queries but it is useful in some situations

Restrictions in WHERE

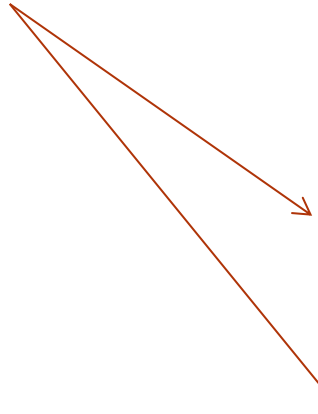
- `select * from student where id=23;`
- `select * from student where name LIKE '%jo%';`
- `select * from student where id=23 AND city='porto';`
- `select * from student where city IN ('porto', 'beja');`
- `select * from student where born BETWEEN '1992-01-01' AND '1996-01-01';`
- We'll see later that `SELECT` can also be used in `WHERE`

SORTING

- Use ORDER BY *column(s)*
- `select * from student order by name ASC;`
- `select * from student order by name, lastname ASC;`
- Sorting can be tricky for text: respect case, collation (does 'á' comes before 'a'?)
- Check the default collation parameter of your database (do a “psql -l”)

GROUP BY and HAVING

- GROUPing is done after the WHERE clause
- HAVING can eliminate grouped rows
- `select count(*), type from disciplina group by type having count(*) > 10`



count(*)	type
12	PRA
5	TP
11	PL
9	LAB

GROUP BY and HAVING

- Check the Manual example:

```
SELECT product_id, p.name, (sum(s.units) *  
    (p.price - p.cost)) AS profit
```

```
    FROM products p LEFT JOIN sales s  
    USING (product_id)
```

```
    WHERE s.date > CURRENT_DATE -  
    INTERVAL '4 weeks'
```

```
    GROUP BY product_id, p.name, p.price,  
    p.cost
```

```
    HAVING sum(p.price * s.units) > 5000;
```

Aggregate functions

- `select student_id from grades where grade > (select avg(grade) from grades where course_id='abc')`
- `select count(distinct city) from student;`
- Aggregate functions:
 - count
 - sum
 - max
 - min
 - avg

Aggregate functions

- `select max(grade) as Max, min(grade) as Min from grades where course_id=21;`
- AVG, SUM and COUNT can use DISTINCT, avoiding duplicates
 - Remember DISTINCT is a costly operation
- `select count(distinct dept) from course;`

Joins

- INNER JOIN
- LEFT, RIGHT, FULL
- Join condition is specified with ON or USING (in an INNER JOIN can use WHERE)
- Use alias to have shorter names or to remove ambiguity
- `select s.name, sg.course_id from student s inner join sg on s.id=sg.student_id;`

Joins (1)

- There is a simple form, dropping INNER JOIN:
- `select s.student_id, c.name from signs s, course c where s.course_id=c.course_id;`
- `select s.student_id, c.name from signs s, course c where s.course_id=c.course_id AND s.student_id=23;`
- Note: if you forget the join condition, you have a product !
- For an explicit product, use CROSS JOIN

External joins

- Show students even if they are not in any course:
 - select s.* from student s **left join** signs sg on s.id=sg.student_id
- Unmatched rows in the table signs are empty (columns with null values)
- Also **right join** and **full join** (check the manual)

Nested queries

- Sometimes it is easier to write a nested query than a join
- The first select is called the **outer select**, and the select in the where clause is called the **inner select**
- Two types of nested queries:
 - Correlated, when the inner select uses information from the outer select
 - Non-correlated, when the inner select does not use information from the outer query

Nested queries

- Can be used to implement the semi-join and anti-join algebra operators
- Nested queries can return rows, or be scalar
- Nested queries can be transformed into joins (a process called “unnesting”)
- If a sub-query is not correlated, it can be computed once instead of each time for each row of the outer select

Sub-SELECTS

- `select * from student where student_id IN (select student_id from attends where status='pending');`
- Use an alias to refer to outer tables
- `select * from signs s where date = (select max(date) from signs where student_id = s.student_id);`
 - *select most recent sign OP operation for students*

EXISTS and NOT EXISTS

- Checks for an (not) empty set of results
- Example: select rooms not having classes
- `select * from room r where not exists (select s.room_id from schedule s where s.room_id = r.room_id)`
- This is the anti-join pattern
- Using EXISTS we have the semi-join pattern

Semi and anti-joins

- Can allow huge decrease in execution time
- A semi-join returns only one row per condition met
 - Can be more efficient than an equivalent join that returns the same row multiple times
- List courses having at least one student
- `select * from courses, signs where id=course_id`
- A course would appear dozens of times
- With a semi-join (EXISTS) it would only appear once

Semi and anti-joins (1)

- Evaluation of EXISTS / NOT EXISTS can skip when first row found
- Very efficient access patterns
- Careful with NULL values in NOT IN and NOT EXISTS
 - Same query may not be equivalent using either of them

Comparing scalars with sets

- ALL, SOME/ANY
- SOME and ANY are synonyms
- Give the students having grades greater than any of the grades of the Database course:
 - `select student_id from grades where grade > ALL (select grade from grades where course='bdad')`
- Operators: `<`, `>`, `<=`, `>=`, `=`, `!=`, `<>`
- NOTE: `"= ANY"` is the same as `"IN"`
`"<> ALL"` is the same as `"NOT IN"`

Quantifying sub-queries

- Which students have the highest gpa?
- `select * from student s where not exists (select * from student where gpa > s.gpa)`
- Translation: list students for which it doesn't exist students with higher gpa. Alternatives:
 - `select * from student s where not gpa < ANY (select gpa from student);`
 - `select * from student s where gpa >= ALL (select gpa from student);`

Auto-joins

- Sometimes we need to join a table with itself
- **student** {id, name, degree_id}
- What are the colleagues of student 1000?
- `select s2.name from student s1, student s2
where s1.degree_id = s2. degree_id and s1.id =
1000;`
- Can you improve it?

Auto-joins (1)

- **employee** {id, name, supervisor_id}
- List names of all employees and their supervisors
- `select e1.name, e2.name from employee e1, employee e2 where e1.supervisor_id = e2.id;`
- Students signing at least for two courses
- `select sg1.student_id from signs sg1, signs sg2 where sg1.student_id = sg2.student_id and sg1.course_id <> sg2.course_id;`

Auto-joins (2)

- **employee** {id, name, supervisor_id, dept}
- List pairs of employees of the same department:
- select e1.name, e2.name from employee e1,
employee e2 where e1.dept = e2.dept and e1.id
< e2.id order by e1.name, e2.name;
- Why the restriction “e1.id < e2.id” ?

Division

- There is no division in SQL
- Consider: signs and course

bd=> select aluno_id from inscrito where
disciplina_id in (select id from disciplina)
group by aluno_id
having count(*) = (select count(*) from disciplina);

Division (1)

- Alternate version (from Codd's definition)

bd=> select distinct i.aluno_id from inscrito i where
not exists

(select * from disciplina d where not exists
(select * from inscrito i1 where
i1.aluno_id=i.aluno_id and
i1.disciplina_id=d.id))

Find students with no courses for which they are
not signed for

Division

- What happens in the two previous versions if the divisor table courses is empty?
 - Divide by zero
- What happens in the two previous versions if the dividend contains the rows of the divisor and more?
 - Remainder not zero
- How do you interpret the results?

Set operators

- SQL set operators are:
 - UNION, INTERSECT, EXCEPT
- Relations must be compatible: same number of columns and same domains
- Duplicates are eliminated unless ALL is used:
 - UNION ALL,...
 - Works as if each element has a count (nber of times it appears)
 - Union sum the count, Except subtracts the count, Intersection takes the minimum

Behaviour of NULL values

- NULL values are used to specify “incomplete”, “unknown” or “not applicable”
- Any arithmetic operation with NULL returns NULL
- Any comparison returns UNKNOWN (which **WHERE** considers as false)
- GROUP BY and DISTINCT retain only one NULL value
- Aggregation functions ignore NULL values and return NULL if there are only NULL, except count that returns 0; count(*) doesn't care about nulls

Behaviour of NULL values

- As comparisons returning UNKNOWN are considered false by SELECT it is important to understand the queries
- To test for NULL values uses IS NULL or IS NOT NULL
- `select * from client where phone2 is null;`

Insert, update, delete

- insert into student ...values ...;
 - Specify the columns
 - Do not specify the columns
 - Use a select in VALUES
- update student set city='porto' where id=2;
- delete from student where id=34;
- If you do not specify the rows in the WHERE, it applies to the whole table

Views

- Views are “virtual” tables, being the result of a select
- Views can be used anywhere a table is used

```
create view courses_2_ects AS select  
course_id,name from course where ects=2;
```

```
select * from courses_2_ects;
```

Window functions

- A window function performs a calculation across a set of table rows that are related to the current row
- Similar to aggregate functions, but rows are not grouped into a single one

Window functions

- Example: list each course showing the average ECTS for its degree:
- `select title, ects, avg(ects) over (partition by degree_id) from course;`

title	ects	avg
dbms	5	5.000000000000000000
intro psy	2	1.500000000000000000
psy II	1	1.500000000000000000

Window functions

- We have: avg(ects) over (partition by degree_id)
- It's a kind of group by
- AVG is applied OVER rows partitioned (grouped) by degree_id
- If PARTITION is omitted all rows are included
- select title, ects, avg(ects) over () from course;

title	ects	avg
dbms	5	2.6666666666666667
into to psy	2	2.6666666666666667
psy II	1	2.6666666666666667

Window functions

- If there is ORDER BY only rows processed up to the current row are included:
- `select title, ects, avg(ects) over (order by ects)`
from course;

• title	• ects	• avg
• -----+	• -----+	• -----
• psy II	• 1	• 1.000000000000000000000000
• into to psy	• 2	• 1.500000000000000000000000
• dbms	• 5	• 2.6666666666666666667

Window functions

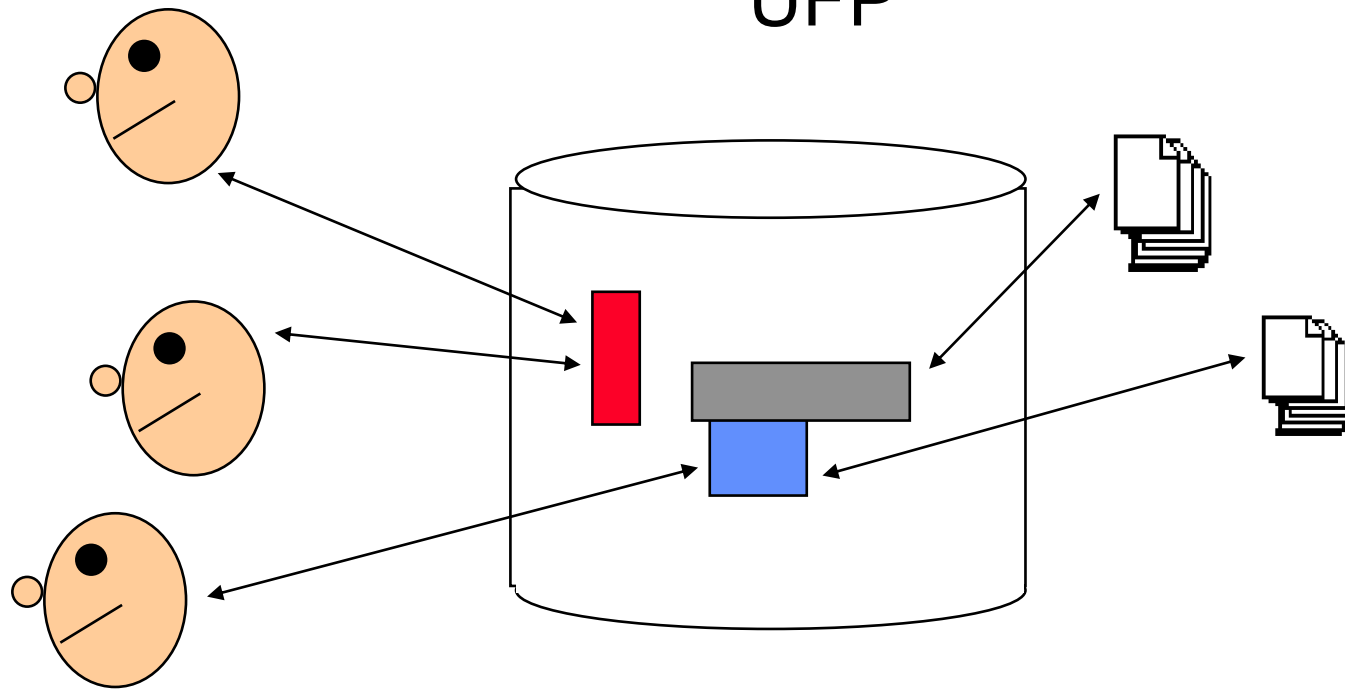
- Other functions than aggregate functions:

```
select title, ects, rank() over (order by ects) from course;
```

title	ects	rank
psy II	1	1
into to psy	2	2
dbms	5	3

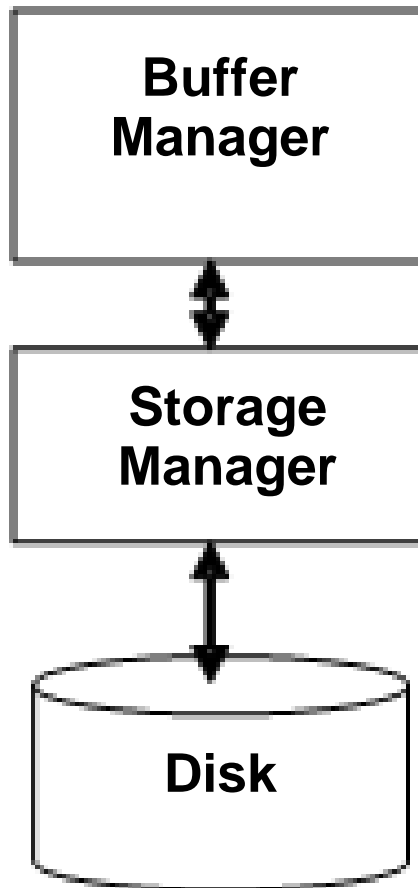
Storage

Database Management Systems Feliz Gouveia UFP



The Storage Manager

- It is the component in charge of managing the storage of relations on disk



Disks...

A 5MB disk from IBM on the road in 1956...



Storage levels

- Cache (in the motherboards, access $\sim 10^{-8}$)
- RAM (access $\sim 10^{-8}$)
 - A portion of RAM is allocated for the database buffers
- Virtual Memory (32 or 64-bit addressing)
- Secondary memory (access $\sim 10^{-4}$)
 - Typically magnetic disks, or SSD
 - To do: check your disk's access times

Disk technology

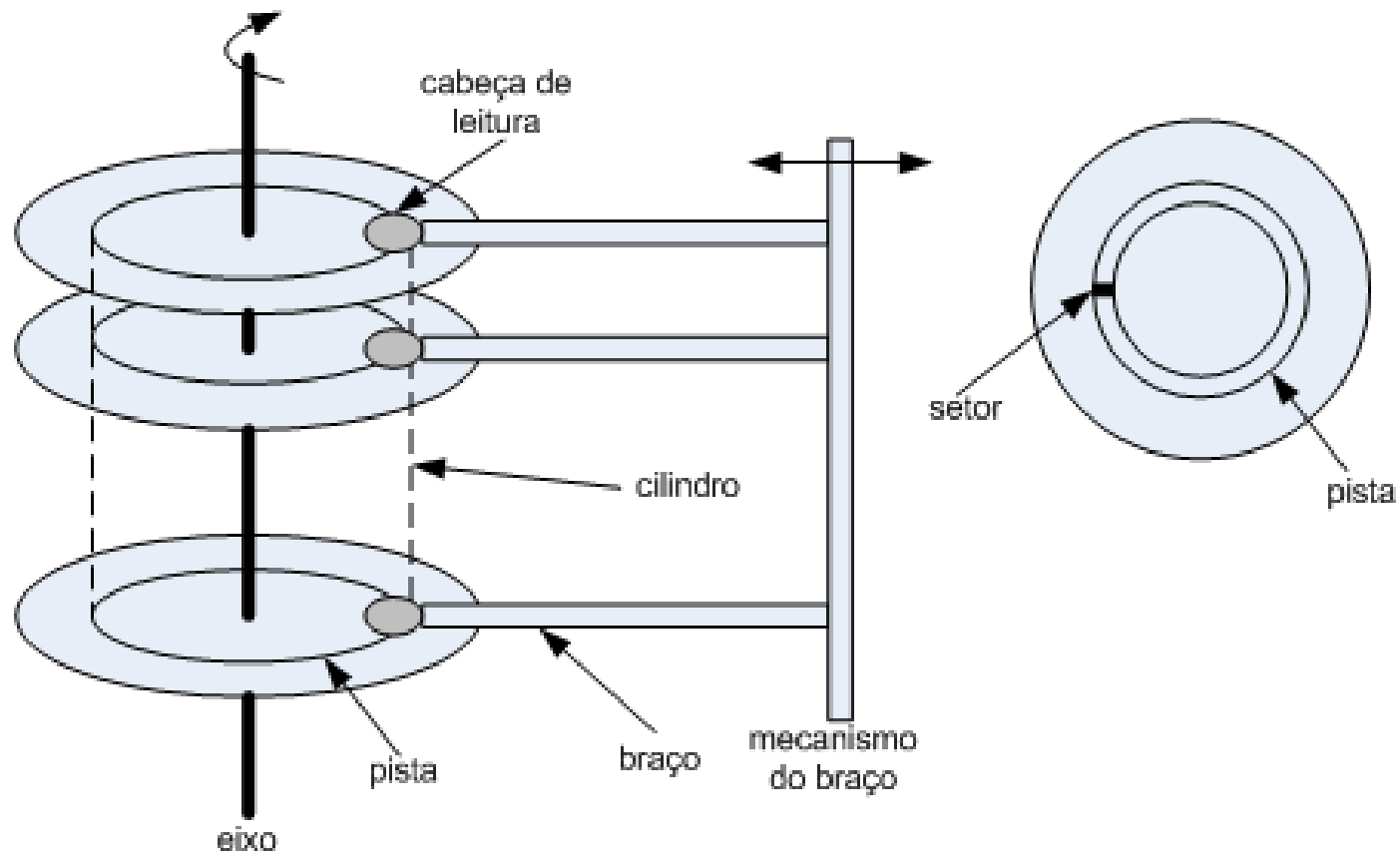
- SSD technology is faster for reading, while magnetic disks are more reliable (for now)



imagens: eurogamer.net

Magnetic disks

- General overview of a disk



Disk access

- Transfer unit is a block (minimum)
- Disks rotate at speeds from 5400 to 15000 rpm (7500 is common)
- To read a block:
 - The arm moves to the track (seek time)
 - Waits for the sector to rotate under the head (rotational latency time)
- Read time = seek time + rotational delay

Disk access

- After positioning, blocks are read at the disk transfer speed
 - Can vary from 75 to 250 MB/s
 - Note the controller can be serving several requests
- Common interfaces
 - SATA: 600 MB/s
 - SAS (Serial Attached SCSI): 600 MB/s

Database operations

- The DBMS should be able to answer “select * from student” without reading **all** blocks from disk
 - Solution: group blocks of the same relation in the same file
- The DBMS should be able to answer “select * from student where sid=123” without reading **all** blocks of relation **student**
 - Solution : use an index (will see later)

Database storage

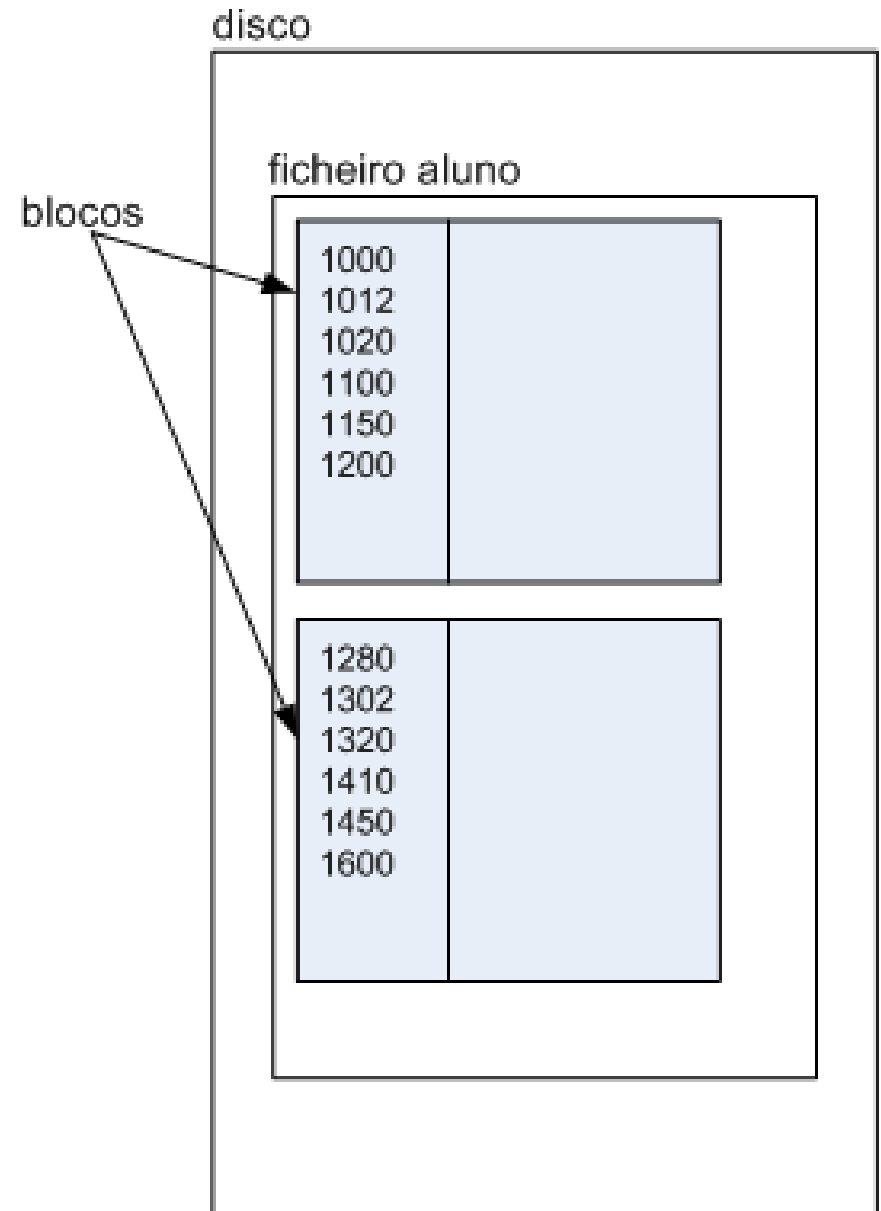
- The database is organized in files
- A file is organized in fixed length blocks (called pages)
- Blocks are the transfer unit between disk and the memory buffers
- The memory buffers have a limited allocation
- Main goal of the DBMS is to minimize traffic between disk and memory

Some examples

- MySQL uses a format FRM, and keeps the files in the folder datadir/***database_name***:
 - student.frm
- PostgreSQL keeps the files in data/***id_of_db***
 - One file for the table and other for the index

Database storage

- File Student on disk
- One relation per file
- The relation is stored in several contiguous blocks



Record storage

- Fixed length records: stored sequentially, easy to read and write. Sometimes a record occupies two blocks, which complicates access
- Variable length records: an end of record marker is used (\perp). The beginning and end of records is known (this is the classical model, also called *slotted-page*)

PostgreSQL: blocks

- By default 8K, but can be changed in postgresql.conf
- 20 byte header, with several information including pointers to free space
- List of pointers to the records with (offset, size)

Files

- Physical reads and writes are at the block level
- A file is a collection of blocks (of the same relation, but some DBMS allow several relations)
- File organization options:
 - Sequential: records are inserted by order of some column.
 - *Heap*: records are inserted in the first free space available. No order.

Heap files

- A page directory has pointers to the file's pages, with free space in each one
- A *heap* file is a non-ordered file
- Nevertheless, access to records can be made via an index (will see later)

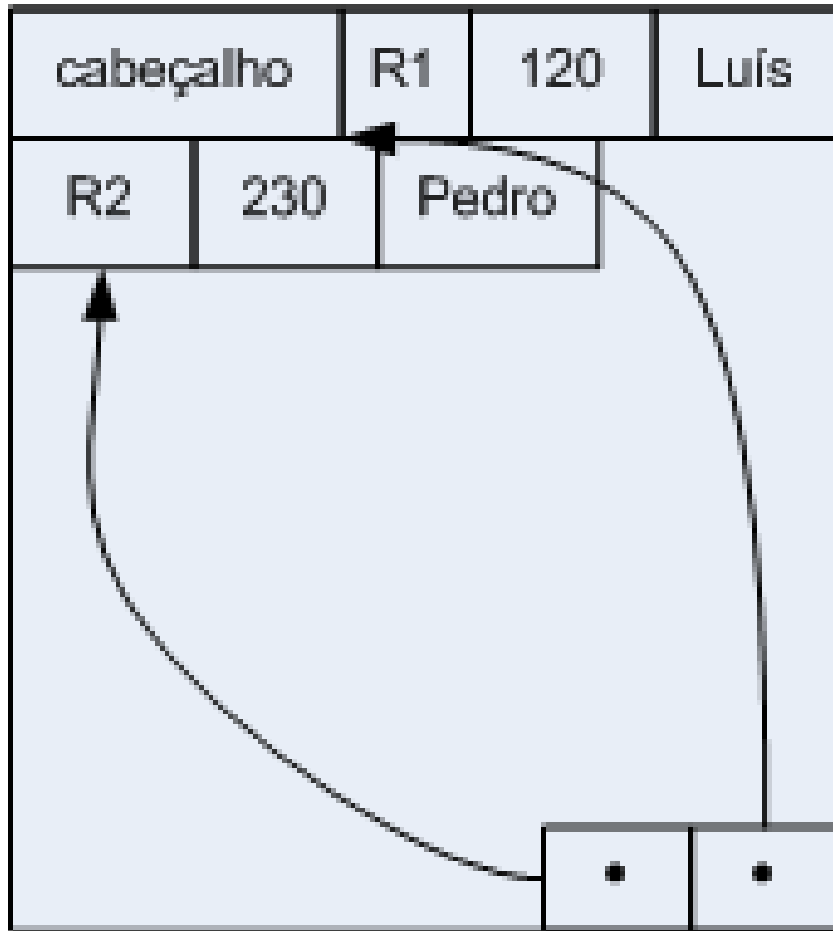
Disk organization

- How to organize and store records within a page?
- Common models
 - N-ary Storage Model (NSM, or slotted-page)
 - Decomposition Storage Model (DSM)
 - Partition Attributes Across (PAX)

N-ary Model (NSM, slotted-page)

- Records are stored in sequence, in the form {record_id, attributs}
- At the end of each page there is a pointer (an offset) to each record
 - PostgreSQL keeps it in the beginning
- The record address is virtual: records can be moved within the page without changing the external address
 - $\text{RecordID} = \text{pageId} + \text{offset}$

N-ary Model



Records have the same schema

The header keeps a pointer to the beginning of free space

Analysis

- When a block is read, full records are read. When we just want some attributes, this is a waste of memory
- When we alter the schema of a relation, the storage of the records must be changed
- It is nevertheless the most common model, and has a good overall performance

Decomposition Storage Model

- Proposed to solve the waste of NSM
- Each N-attribute relation is decomposed in N-sub-relations of one attribute each; each sub-relation is stored in different pages
- Each page has only records with one attribute (can use NSM internally)

DSM

cabeçalho	R1	120	R2
230			

A page having a binary sub-relation can be organized as NSM

Sub-relations are fixed length

cabeçalho	R1	Luís	R2
Pedro			

Analysis

- Solves NSM problems, because it reads only needed attributes
- When a query asks several attributes, implies “joining” pages
- When a schema is altered, implies only changing the pages with the changed attributes
- When locking, can lock at the attribute level (will see later)

Analysis (1)

- DSM needs 1 to 4 times more space than NSM
 - Keys are duplicated
 - Each relation is stored twice: indexed on the key, and indexed on the value

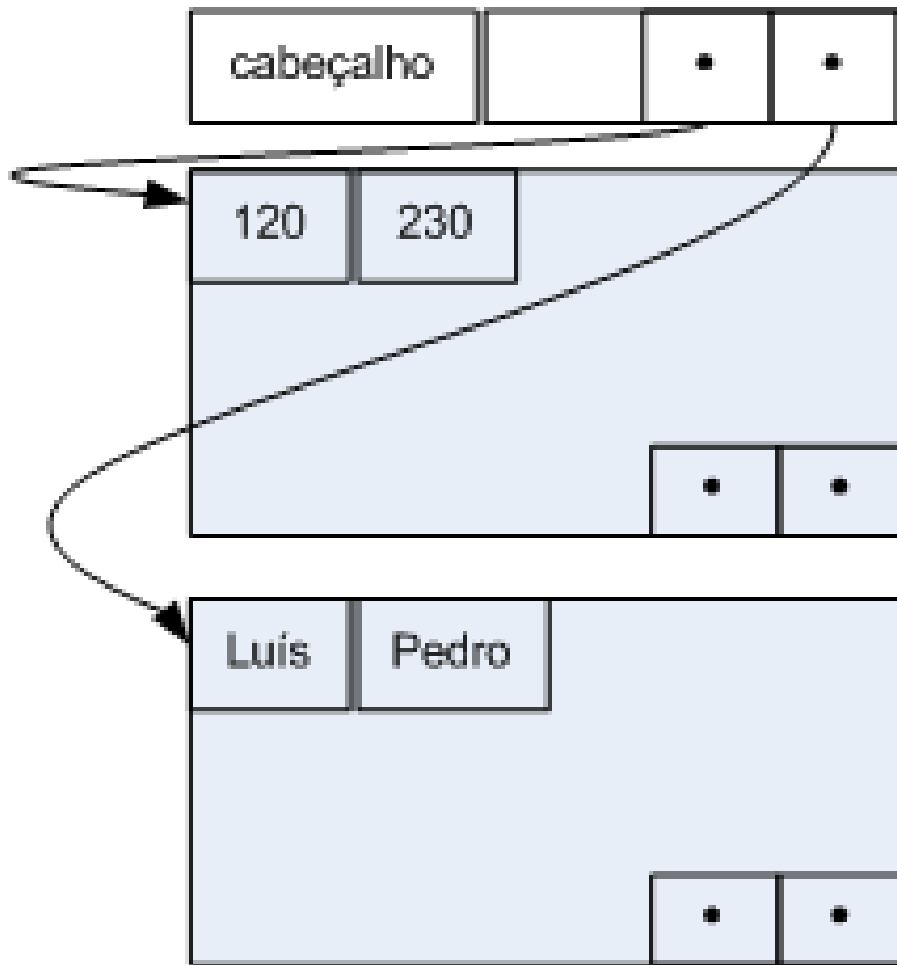
Analysis (2)

- When updating values:
 - 2 accesses in NSM (1 for the record, 1 for the index)
 - 3 accesses in DSM (2 for the record, 1 for the index)
- Insert/delete records
 - 2 accesses in NSM (1 for the record, 1 for the index)
 - 2 accesses in DSM **for each attribute**

Partition Attributes Across (PAX)

- The best of both worlds
- Each page is divided in mini-pages, according to the attributes
- It only changes the organization of records **within** a page

PAX model



Each page is organized in mini-pages

Header points to each mini-page

PAX only changes the internal organization of pages

Page header

- {page_id, ptr_mini_page1,...,ptr_mini_pageN, N_of_attrs, N_of_records, size_atr_i, size_atr_j}
- ptr_mini_page: pointer to mini-page
- N_of_attrs: number of attributes
- N_of_records: number of records in page
- size_atr_i: size of each variable-length attribute

Mini-page structure

- Mini-pages for fixed-length attributes
 - Size is kept in the header. At the end of each mini-page a bit is kept to tell if value is null or not
- Mini-pages for variable-length attributes
 - At the end of each mini-page pointers are kept for each record

Analysis

- When querying a small set of attributes, PAX behaves like DSM
- If joining is needed, it can be done in memory as each page has the required mini-pages
- The Storage Manager can decide, depending on the number of attributes, to have some pages in NSM and others in PAX

Analysis (1)

- PAX uses the same space as NSM for pointers
- PAX uses more space than DSM as the number of attributes increases

Comparing the 3 models

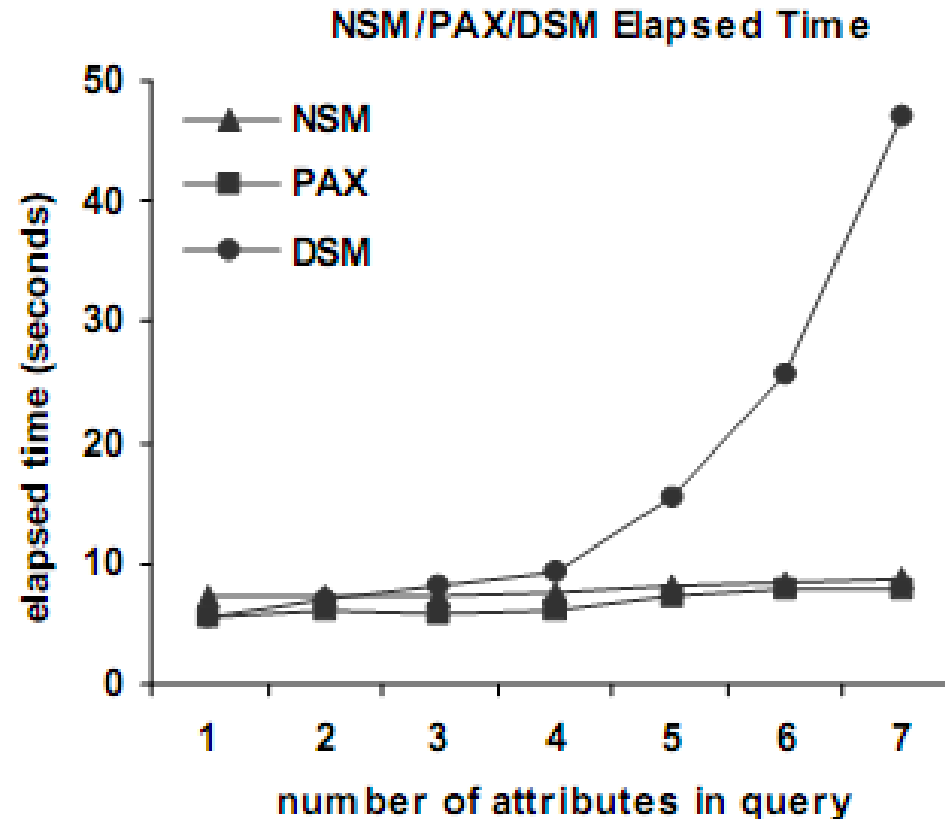


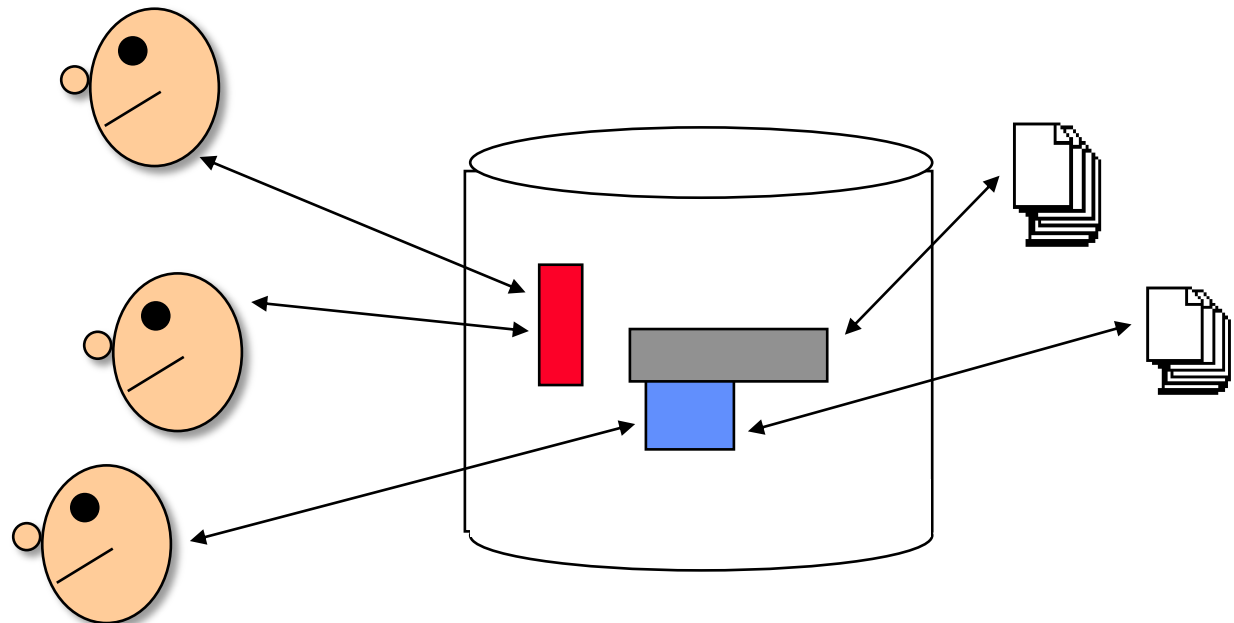
FIGURE 6: *Elapsed time comparison as a function of the number of attributes in the query.*

Ailamaki *et al* (2001) "Weaving Relations for Cache Performance"

INDEXAÇÃO

Sistemas de Gestão de Bases de Dados

Feliz Gouveia
UFP



Organização de ficheiros

- Em situações reais, não há espaço para guardar em memória todos os registos
 - O SGBD tem de paginar
- As relações são guardadas em ficheiros
- Um ficheiro é uma coleção de registos
- Um registo é um par <rid, dados>
- A organização dos registos no ficheiro e a facilidade de acesso, inserção e remoção têm grande influência no desempenho do SGBD

Representação dos registos

- Os registos são uma coleção de campos
- Têm um apontador para o seu esquema
- Guardam informação sobre o seu tamanho
- Têm uma marca temporal (última modificação, ou leitura)

Formato dos registos

- Cabeçalho, com:
 - 4 bytes para o esquema
 - 4 bytes para o comprimento
 - 4 bytes para a marca temporal (de inserção ou atualização)
- Dados, usando um alinhamento fixo (tipicamente 4 byte)
- Podem ser usados outros formatos
- PostgreSQL: valores muito grandes (> 8K) são armazenados noutra página (TOAST)

Representação dos dados

- CHAR(N), usa N caracteres e preenche os não usados com um caracter SQL ilegal
- VARCHAR(N), usa N+1 bytes, o primeiro byte guarda o tamanho (4 bytes em PostgreSQL)
- Vários tipos de inteiros (1, 2, 4, e 8 bytes)
- Datas
- Intervalos
- ...
- Para outros tipos de dados deve consultar o manual

Armazenamento dos registos

- Os registos podem ter comprimento:
 - Fixo
 - Variável
- Nalguns sistemas os campos são deslocados de múltiplos de 4 (preferível para inteiros) ou 8 (preferível para reais)
- Campos muito longos (como Blobs) podem ser armazenados em discos diferentes para melhorar a velocidade de acesso

Formato dos blocos (páginas)

- Um bloco tem um tamanho fixo: 4, 8, 16, 32Kbytes são valores comuns
- Os registos são agrupados em blocos
 - O número de registos por bloco depende do tamanho das suas colunas
- Se os registos têm comprimentos diferentes, usa-se uma tabela de deslocamentos:
 - Os registos guardam-se a partir do início do bloco, os deslocamentos a partir do fim
 - Quando um registo é removido, deixa-se uma marca (“pedra tumular”) no lugar do deslocamento

Ficheiros

- Um ficheiro organiza os registos por páginas no disco
- Tipos de ficheiros:
 - Não-ordenados (*heap file*)
 - Ordenados num ou mais atributos
 - Indexados
 - *Hashing*
- Geralmente é guardada uma relação por ficheiro
 - Excetuam-se os ficheiros agrupados (*clustered*) nos quais se guardam diferentes relações com alguma “proximidade”: por exemplo as que são usadas frequentemente em junções. Nem todos os SGBD permitem ficheiros com relações diferentes.

Ficheiros não-ordenados (*heap*)

- Os registos são guardados por ordem de chegada, no primeiro espaço livre
 - Simples e rápido
- Não há ordenação
- Encontrar um registo implica percorrer todos os registos
- Eficazes para consultas de varrimento sequencial

Ficheiros ordenados

- Chamados sequenciais, são ordenados por um atributo (que pode ser a chave primária)
- A inserção/remoção de registos pode implicar reorganização do ficheiro
 - Quando não há um lugar livre, os registos vão para uma área especial
 - Logo, ficam fora de ordem e tornam a pesquisa mais complexa
 - Periodicamente é necessário ordenar o ficheiro
- A pesquisa na coluna ordenada é baseada na pesquisa binária
- As pesquisas nas colunas não-ordenadas implicam varrimento sequencial

Ficheiros indexados

- Utilizam um ou mais índices para localizar registos de forma eficaz sem ter de percorrer todos os blocos
- O índice que define a ordenação do ficheiro chama-se **índice agrupado** (ou primário)
- Os outros índices são **secundários**
- Um ficheiro só pode ter um índice agrupado
- *Exemplo:* Lista Telefónica, ordena por apelido (índice agrupado); no fim da lista pode ter outro índice (por exemplo por setor de atividade), mas este não determina a listagem nas páginas

Ficheiros *hashing*

- Usam uma função de *hashing* para distribuir os registos pelos blocos
- Pode existir enviesamento, com preenchimento rápido de apenas alguns blocos
- Não é possível ler a chave de forma ordenada, valores consecutivos podem estar em blocos diferentes
- As técnicas mais utilizadas usam hashing dinâmico

Operações comuns na base de dados

- Percorrer sequencialmente uma tabela
- Pesquisar com testes de igualdade
- Pesquisar com intervalos de valor
- Inserir tuplos
- Remover tuplos
- Ordenar tuplos

Estruturas de indexação

- Uma estrutura de indexação deve receber uma chave e devolver rapidamente os registos a ela associados
- Existem essencialmente duas técnicas:
 - Índices com chaves ordenadas
 - Hashing
- Note que **chave** de um índice não significa o mesmo que no modelo relacional

Índices

- Uma entrada do índice pode conter:
 - A chave e o registo
 - A chave e um apontador para o registo
 - A chave e uma lista de apontadores para os registos correspondentes
- Na opção 1 o índice é o próprio ficheiro de registos
- As opções 2 e 3 são independentes da organização dos ficheiros

Utilizações dos índices

- Devem permitir rápido acesso aos registos de interesse
- Esta vantagem não deve ser eliminada pelo custo de atualização do índice sempre que há inserções, remoções ou atualizações na tabela
- Algumas colunas não podem ser indexadas (Blobs)
- Nalguns casos basta encontrar a chave, não é necessário aceder ao registo, por exemplo:
 - Em verificações de integridade referencial, ou de unicidade da chave primária ou de outra coluna
 - Em contagens
 - Em consultas que usam apenas a chave de indexação (casos de junção com chave primária, por exemplo)

Índices únicos

- É possível definir um índice como sendo único, isto é não admitindo repetições na coluna indexada
 - Útil para definir restrições de unicidade envolvendo mais do que uma coluna
 - O índice dá erro se for inserida uma chave duplicada
- O índice associado a uma chave primária é sempre definido como único

Índices simples e compostos

- Um índice simples indexa apenas uma coluna
- Um índice composto indexa mais do que uma coluna
 - Deve consultar o manual do SGBD para restrições ao número de colunas
 - Na prática a chave é a concatenação dos valores de cada coluna, na ordem em que foram especificadas.
- O índice composto (cidade, temperatura) pode ser usado para pesquisas por cidade, e por uma combinação de cidade e temperatura; mas não pode ser usado numa pesquisa apenas por temperatura, ou temperatura e cidade (a ordem é por isso importante)

Índices com funções

- Por vezes é útil indexar não o valor de uma coluna, mas o resultado de uma função aplicada a esse valor
- Um caso comum tem a ver com a comparação de maiúsculas e de minúsculas.
- Por exemplo, cria-se o índice com as chaves em minúsculas (*tolower(nome)*)
 - O índice pode ser usado com “nome = *tolower*(“João”)”, mas não é usado com “nome = “João””
- Outro exemplo “where year(data_envio) > 2016”

Índices parciais

- Um índice parcial não regista todas as chaves, apenas as que satisfazem a sua cláusula de restrição
- Um índice parcial usa-se quando se sabe que determinadas chaves não são interessantes para as consultas
- Um índice parcial é mais pequeno
- Por exemplo podemos querer indexar apenas encomendas cujo estado seja (“em curso”, “pendente”), excluindo as que têm estado “anulada”

Índices de cobertura

- Permitem responder a questões sem aceder aos registos no disco
- O índice contém todas as colunas necessárias
- Um índice de cobertura permite adicionar colunas aos nós folha do índice, evitando assim ir buscar essas colunas
- Especialmente importante para índices não-agrupados, onde as folhas contêm endereços disco (ou chaves do índice agrupado)
- Dependendo do SGBD, o número de colunas que é possível adicionar varia

Escolhas dos índices

- Geralmente as chaves primárias são indexadas
- Podem-se indexar colunas frequentemente utilizadas, e chaves estrangeiras
- Note-se que se um índice devolver a maioria dos registos, é preferível um acesso sequencial
 - Demora-se sensivelmente o mesmo tempo, e evita-se utilizar memória com o índice, perder tempo a lê-lo, e gastar tempo a organizá-lo e a percorrê-lo
 - A seletividade de um índice é o rácio entre Valores distintos / total de valores
 - Para uma chave primária a seletividade é 1

Escolhas dos índices

- Não fazem sentido em tabelas pequenas (exceto para as chaves primárias)
- Só fazem sentido se as consultas onde forem usados tiverem grande seletividade (se usarem a maioria dos registos, o índice não faz sentido)
- Deve-se consultar o plano de execução das consultas para decidir quais os índices a criar
- Note-se que a criação de índices depende apenas da utilização (isto é, das consultas), e não do modelo concetual (isto é, dos requisitos)

Alguns inconvenientes

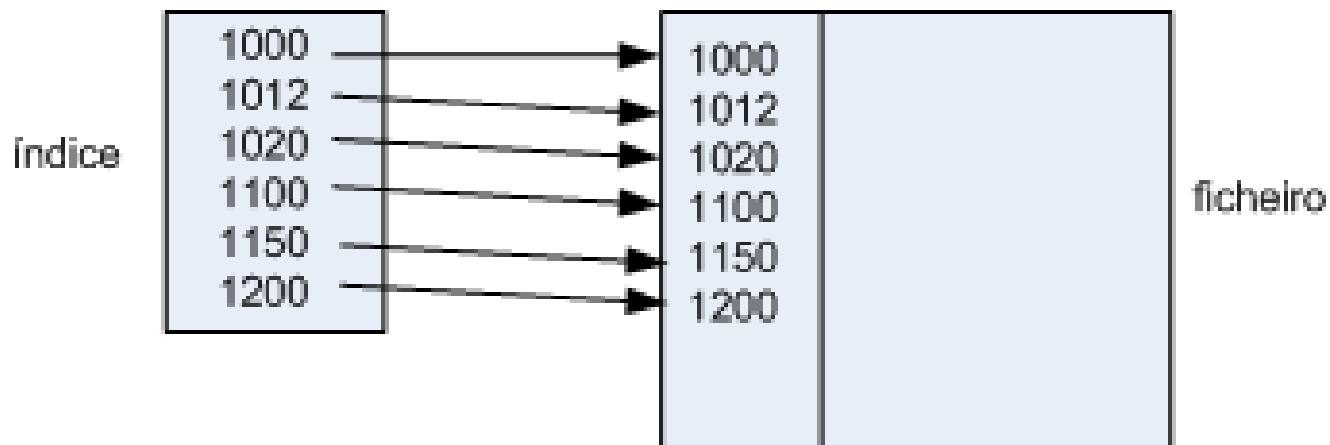
- Os índices ocupam espaço
- Como têm que ser atualizados, podem tornar mais lentas operações de inserção, remoção ou modificação das chaves
- Nalguns casos, a otimização de consultas pode ser penalizada pela existência de muitos índices (dificuldade de escolha e aumento das alternativas de execução)

Índices

- Um ficheiro pode ter vários índices
- O índice primário define a ordenação do ficheiro (geralmente corresponde à chave primária)
 - Também se diz **índice agrupado** (*clustered*), porque o ficheiro está ordenado pela chave indexada
 - Só pode existir um índice primário
- Os índices secundários usam uma chave que não define a ordem do ficheiro (e podem existir vários destes índices)

Estruturas de indexação

- Um ficheiro ordenado pela chave de indexação diz-se “sequencial indexado” (ISAM)
- Ficheiro “aluno” ordenado por “número”:



- A consulta “numero=1012” pede ao índice o número da página onde se encontra o respetivo registo

Tipos de índices

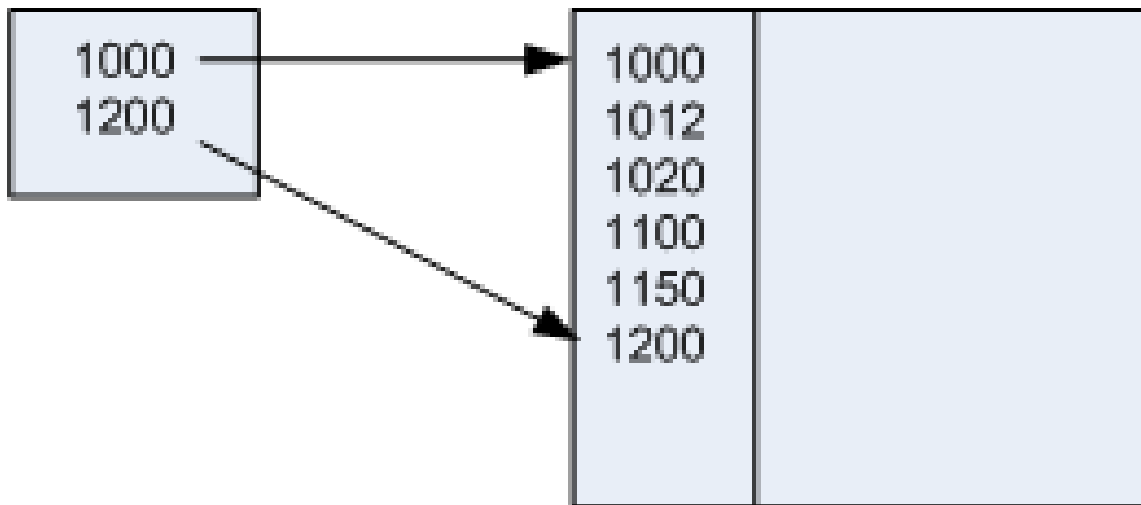
- Densos: há uma entrada para todas as chaves, e cada entrada do índice contém um apontador para o registo
- Esparsos: há entradas apenas para algumas chaves. As outras chaves acedem-se pela que tem maior ou igual valor à que procuramos
 - Só pode existir um índice esperso por ficheiro; como implica ordenação da chave, tem que ser primário (ou seja agrupado). os registos com a mesma chave estão guardados ordenados, basta ter um apontador para o primeiro
 - Os índices esparsos usam menos espaço e simplificam inserções e remoções; geralmente usam uma entrada por bloco

Tipos de índices

- Os índices secundários têm que ser densos (uma entrada por cada chave)
 - Não é possível um ficheiro sequencial indexado ser ordenado ao mesmo tempo pelo índice primário e por outro secundário
- É normal as entradas do índice secundário apontarem, não para os registos, mas terem uma chave do índice primário
 - Os acessos via índice secundário fazem-se em duas etapas: ler o índice secundário, aceder à chave no índice primário
 - Implica índice primário único (porquê?)

Índice esparsos

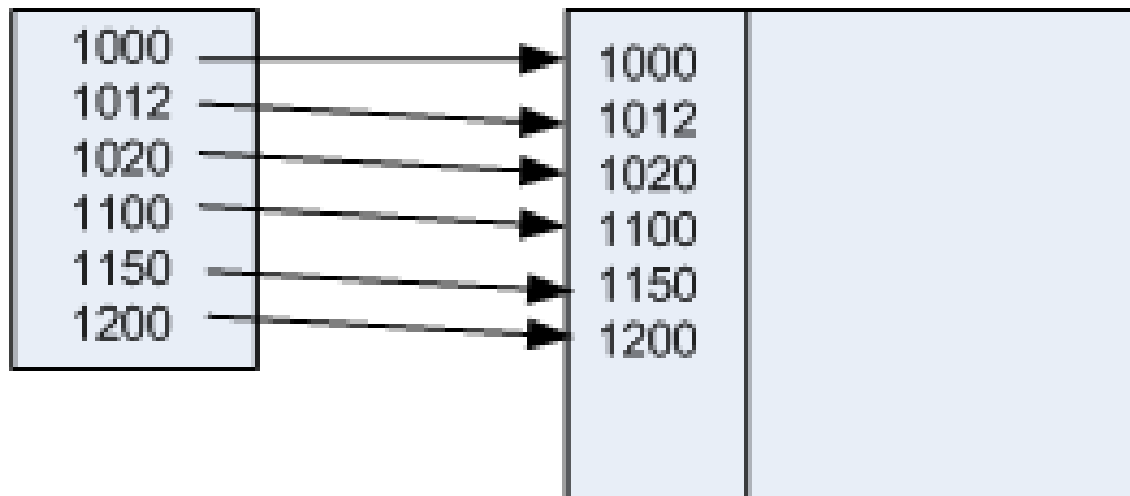
- Tem apenas uma entrada por cada página de valores do atributo indexado:



- A consulta “numero=1012” pede ao índice o número da página onde se encontra o respetivo registo. Está na mesma página que o registo com “numero=1000”

Índice denso

- Uma entrada por cada valor do atributo indexado

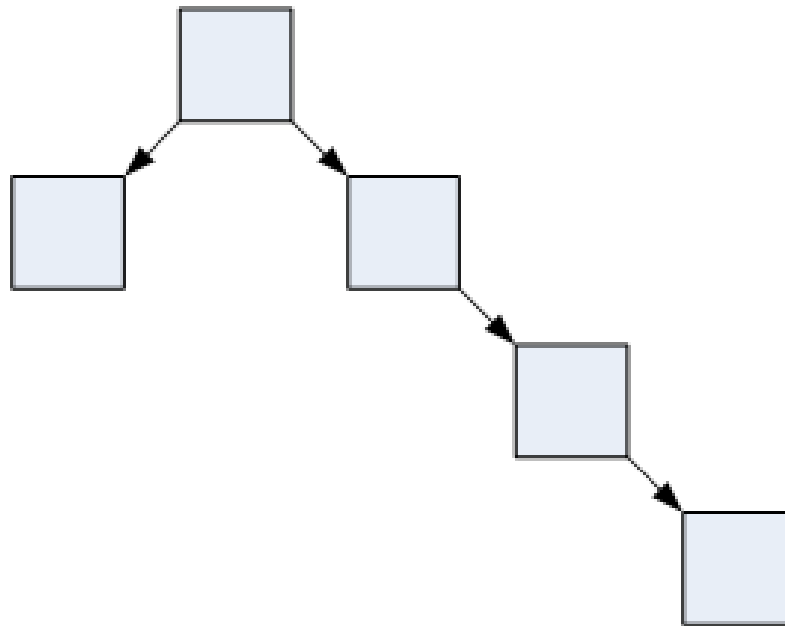


Inconvenientes do formato “sequencial indexado”

- Desempenho diminui com o aumento do ficheiro: registos e entradas do índice podem ficar fora de ordem (em blocos adicionais, não sequenciais)
- Compensado com reorganização do ficheiro, mas tem os seus custos
- Se uma tabela ocupar 50000 blocos, mesmo um índice esparsos teria que ter 50000 entradas
- Pode implicar 200 a 500 páginas de índice...
- Solução: indexar o índice (e se necessário indexar o índice do índice)
 - Obtém-se um “índice multi-nível”: a B-tree

Árvores binárias

- Fornecem uma estrutura de pesquisa eficiente
- No entanto, sucessivas inserções e remoções podem desequilibrar a árvore, degenerando numa lista

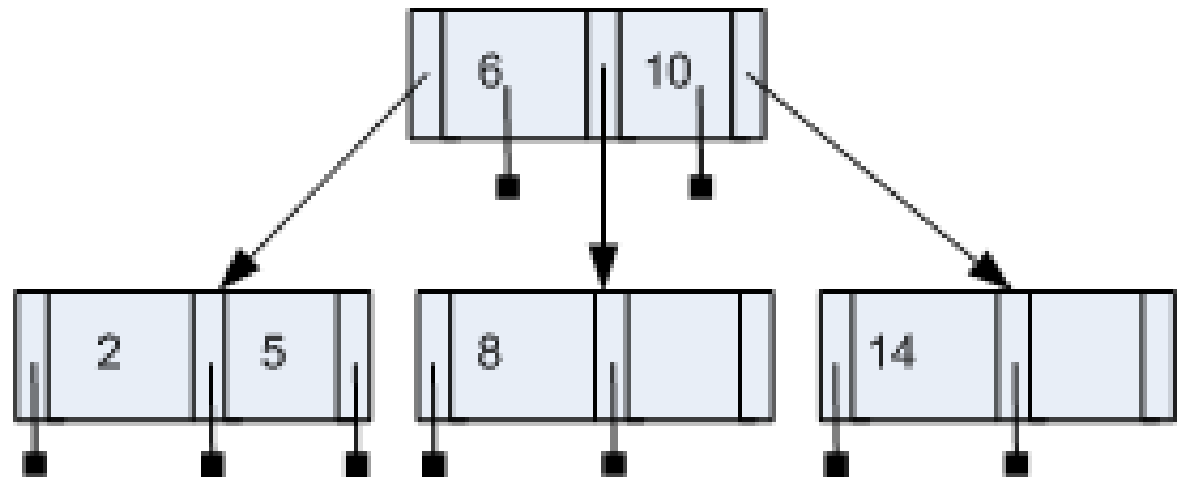


B-Tree

- Proposta por Bayer & McCreight (1969)
 - no mesmo ano do Modelo Relacional
- Generalização de árvore binária
- Numa B-Tree de ordem N cada nó interno tem entre N e $2N$ chaves
 - Apontadores = chaves + 1
 - Cada nó deve ficar pelo menos 50% cheio
 - A raiz tem $1 \leq \text{chaves} \leq 2N$
- Todas as folhas estão ao mesmo nível (a árvore é equilibrada)
- Todas as operações em $O(\log(N))$ onde N é o número total de chaves

B-Tree

- As chaves estão ordenadas
- Cada nó da árvore ocupa um bloco disco para simplificar o acesso (paginação)
- Os nós interiores formam um índice multinível esparsos
- Todos os nós apontam para registros
- Uma B-tree de ordem 1:



B-Tree

- A inserção de novas chaves pode provocar a divisão de nós existentes
 - A divisão vai dos descendentes para os ascendentes
 - Se a raiz tiver que ser dividida, acrescenta-se mais um nível
- Equilibrada, garante-se que qualquer chave é acedida num número máximo de comparações
- Para nível H , garante-se que H acessos ao disco são suficientes para encontrar uma chave
 - Por vezes menos, se a chave se encontrar num nó intermédio

Problemas da B-Tree

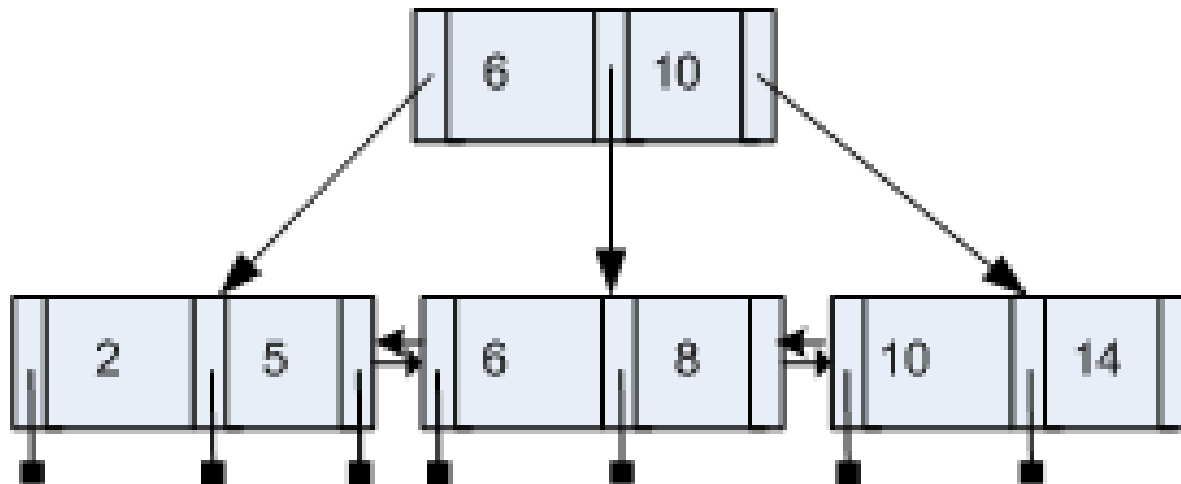
- Para ler chaves em folhas consecutivas tem que se “subir e voltar” a descer
 - Mais difícil responder a consultas com intervalos de valores
- Os nós interiores contêm apontadores para os registos, o que reduz espaço para mais chaves
- Em algumas situações podem no entanto ser mais eficientes (não é necessário descer às folhas)

B+-Tree

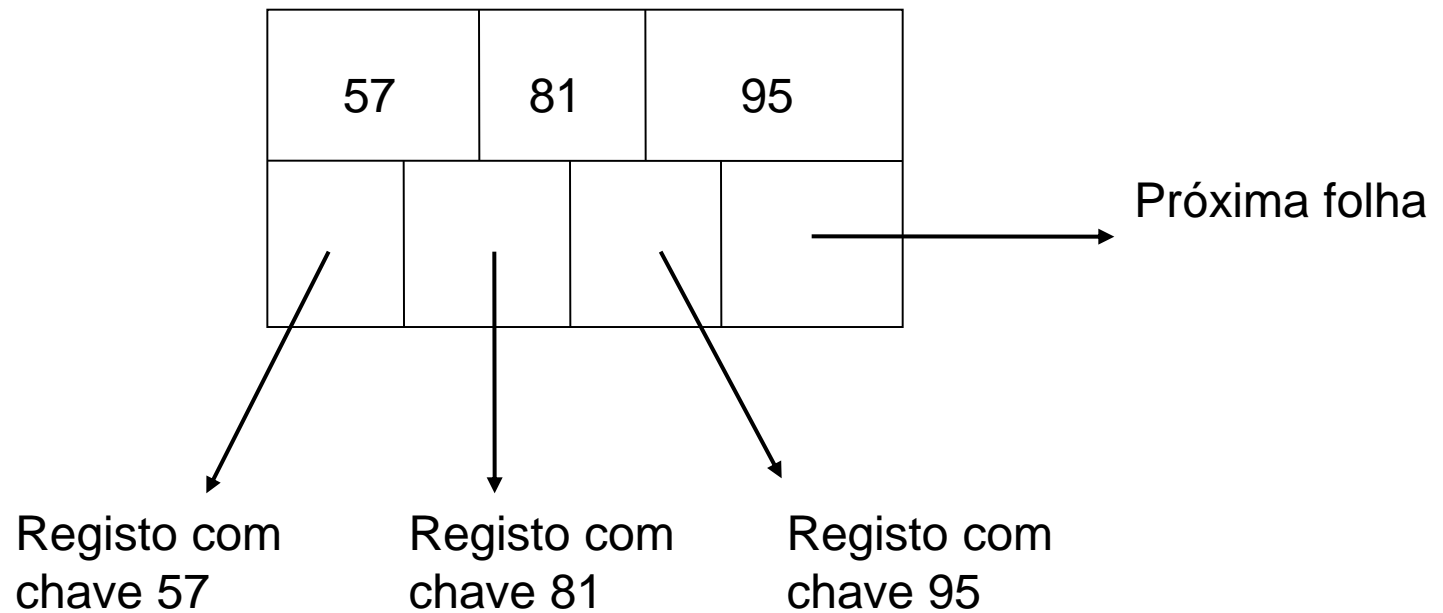
- Proposta para resolver os problemas da B-Tree
- Os nós internos só contêm chaves, as folhas contêm os endereços dos registros
- As chaves são replicadas nas folhas:
 - Numa B-Tree não são
 - Os nós internos podem conter apenas “separadores” que não são chaves. Os separadores são mais pequenos, permitindo guardar mais (ex Prefix B+-Tree)
- As folhas estão ligadas entre si (favorece o acesso sequencial)
- Dentro de cada nó usa-se pesquisa binária para testar chaves

B+-Tree

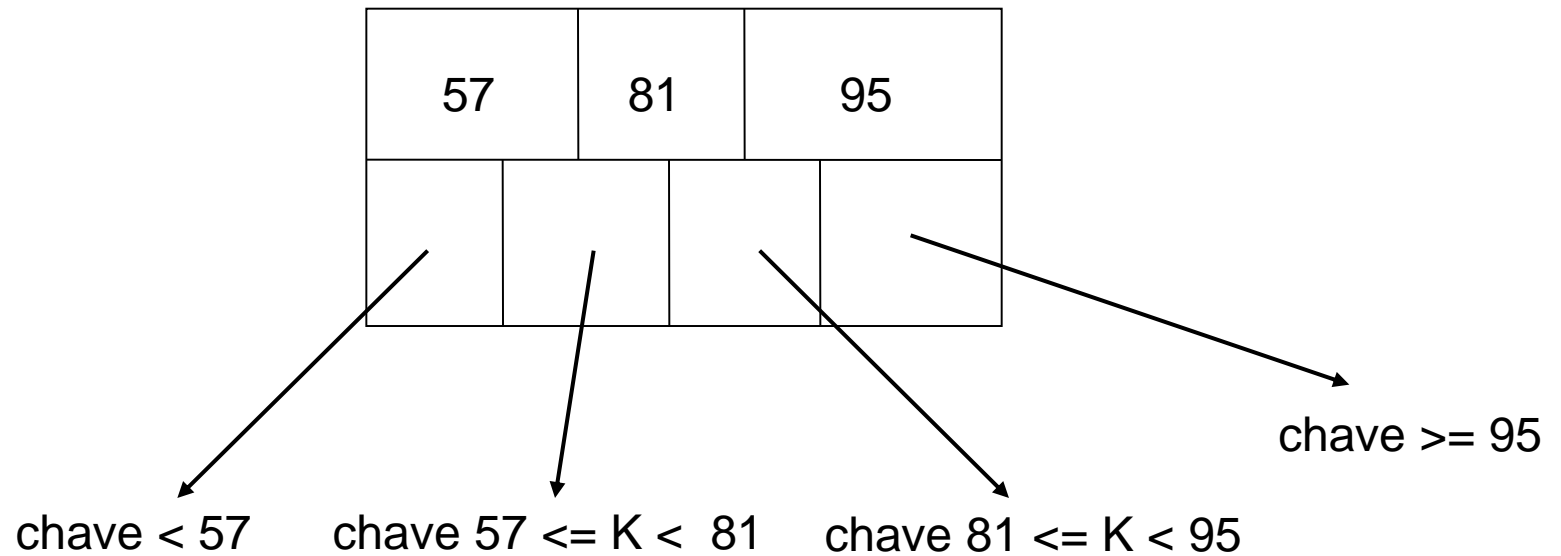
- As folhas podem conter os registos, em vez de apontadores para esses registos
- Os nós intermédios servem apenas para navegação, e não para aceder a registos como na B-Tree



Um nó folha B+-Tree

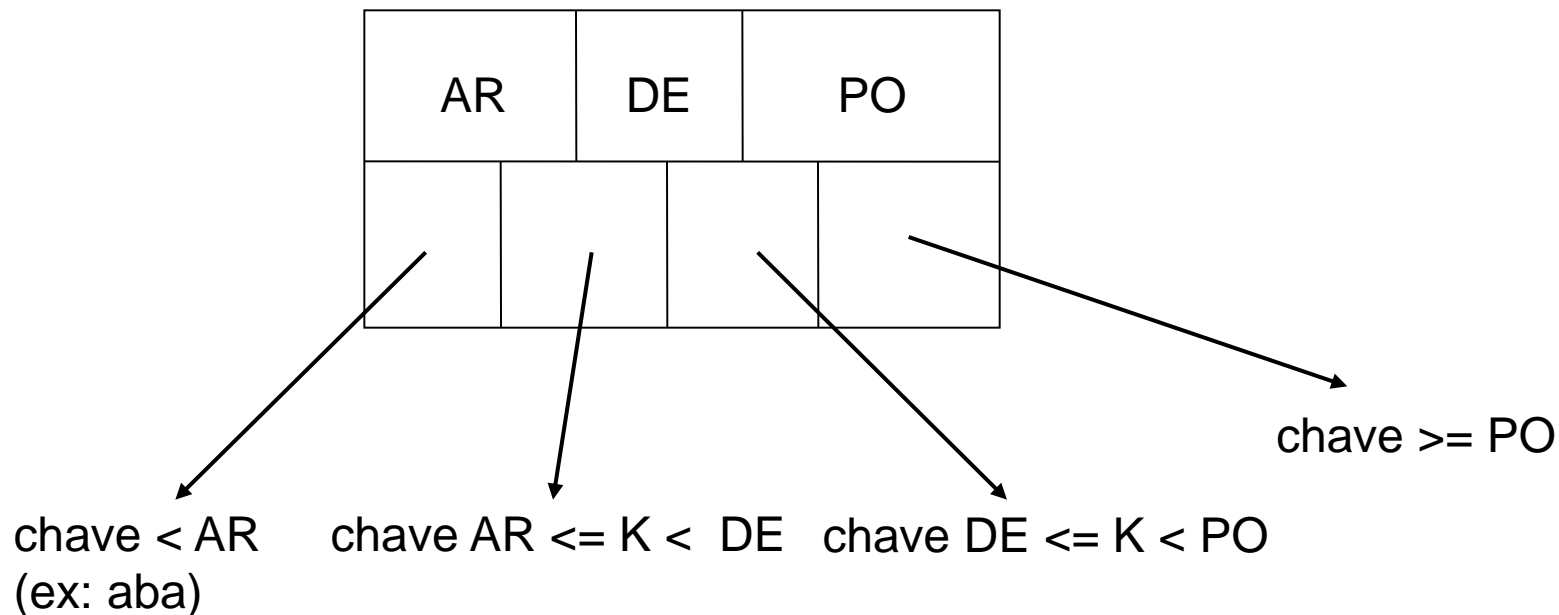


Um nó interno B+-Tree



Um nó interno com prefixos de chaves

- Quando as chaves são longas podem-se usar apenas prefixos para discriminar, poupando espaço



Alguns valores...

- Um bloco no disco tem 4096 bytes
- Uma chave tem 4 bytes, um apontador tem 8 bytes
- Um nó B+-Tree por bloco: $4 \cdot C + 8 \cdot (C + 1) \leq 4096$
- Desta forma um nó é lido em cada acesso ao disco
- $C = 340$, cada nó pode ter 340 chaves e 341 apontadores
 - Um nó pode não estar totalmente preenchido por precaução
- O número de folhas N é $\lceil (\text{total de chaves}) / C \rceil$
- A altura da B+-Tree é $\lceil \log_F(N) \rceil$, em que N é o total de chaves (folhas) e F é o número médio de descendentes de cada nó intermédio (por exemplo $0,6C$)

Árvore com $c=340$, $h=3$?

- Consideremos os nós parcialmente preenchidos, com 255 apontadores em vez de 341
- A raiz tem 1 nó e 255 apontadores
- 2º nível tem 255 nós e 255×255 apontadores
- 3º nível tem $255 \times 255 = 65025$ folhas
- Cada folha tem 255 apontadores, um total de 16.6 milhões de registos (255^3)
- Tudo numa B+-Tree de nível 3! Com a raiz em cache, fazem-se apenas 2 acessos ao disco
- Com os tamanhos atuais da memória, os índices cabem geralmente em memória
- O problema é a pesquisa binária dentro de cada nó

Os resultados são interessantes

- As B+-Trees adaptam-se bem (crescem/encolhem) mantendo-se equilibradas
- São eficientes para consultas por intervalos:
 -**where** 1900 < ano **and** ano < 1950
- Existem muitas variantes: B*-tree, Prefix B+-tree, B-link Tree,...

Comparadas com árvores binárias

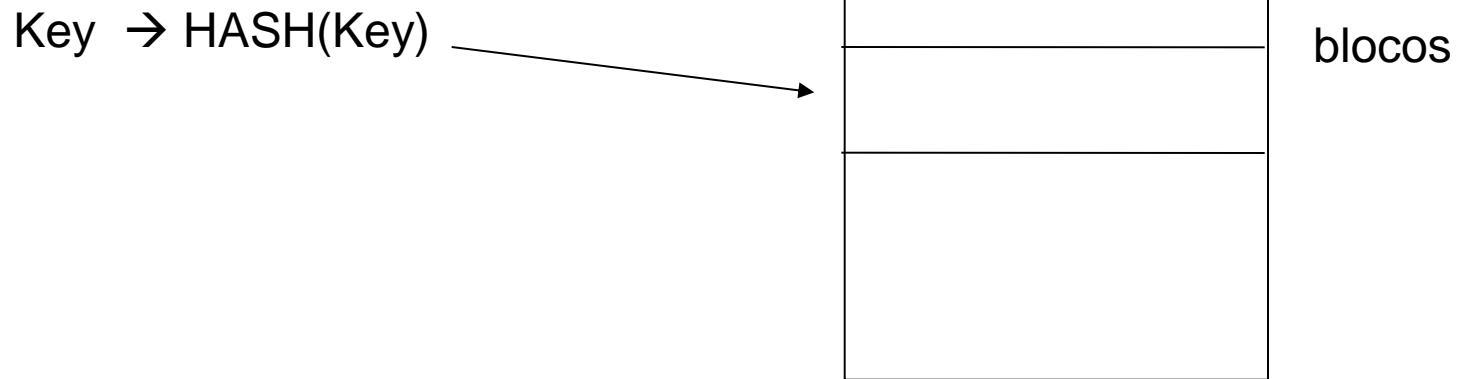
- As árvores binárias são “altas”, e os nós pequenos em tamanho
- As B+-Trees são “baixas” e “largas”
- Para K chaves:
 - numa árvore binária, o pior caso pesquisa $\log_2(K)$ nós
 - Numa B+-Tree o pior caso pesquisa é $\sim \log_{N/2}(K)$, em que N é o número de chaves por nó (N/2 significa que os nós estão preenchidos a metade)

Índices de Hashing

- Evita-se ler a estrutura arborescente
- A função de hashing devolve um valor entre 0 e $B-1$ onde B é o número de blocos do ficheiro
- Usam-se blocos de transbordamento
- Tente a seguinte função: $K \bmod B$, onde K é a chave (se K é texto, pode-se somar o valor *int* de cada character)

Como funciona

- Geralmente lê o registro em um único acesso disco
- A função de hashing deve distribuir uniformemente os valores pelos blocos
- Tem colisões



Características

- Como os valores das chaves não são conhecidos, uma função de hashing deve produzir uma distribuição:
 - Uniforme: cada bloco deverá conter aproximadamente o mesmo número de chaves
 - Aleatória: o *hash* de cada chave não terá nenhuma relação com ordenação ou outra características das chaves
- Podem apenas ser usados para comparações de igualdade, o que limita a sua utilização

Tipos de hashing

- Estático
 - Essencialmente o que vimos
- Dinâmico
 - Adapta-se ao número de chaves
 - Dois tipos: extensível e linear

Crescimento do espaço

- A utilização do espaço deve situar-se entre 50% e 80%
 - se $< 50\%$ desperdiça-se espaço
 - se $> 80\%$ aumenta o risco de transbordar
- Quando cresce, podem-se usar as técnicas:
 - Overflow
 - Dynamic hashing (o número de blocos muda)

GiST

- Generalized Search Tree
- Índices genéricos, são definidos pelo utilizador
 - Árvores binárias equilibradas, com funções de pesquisa, inserção e remoção específicas a um determinado tipo de dados
- Nem todos os SGBD os usam. Existem em PostgreSQL

Índices em SQL

```
CREATE INDEX Index_Name ON STUDENT(ADDRESS)
```

```
CREATE UNIQUE INDEX Index_Nber ON  
STUDENT(STUDENT_ID)
```

- mesmo que PK

Column	Type	Modifiers
--------	------	-----------

-----+	-----+	-----
--------	--------	-------

id	smallint	not null
----	----------	----------

Índices:

```
"ttt_pkey" PRIMARY KEY, btree (id)
```

Índices em SQL

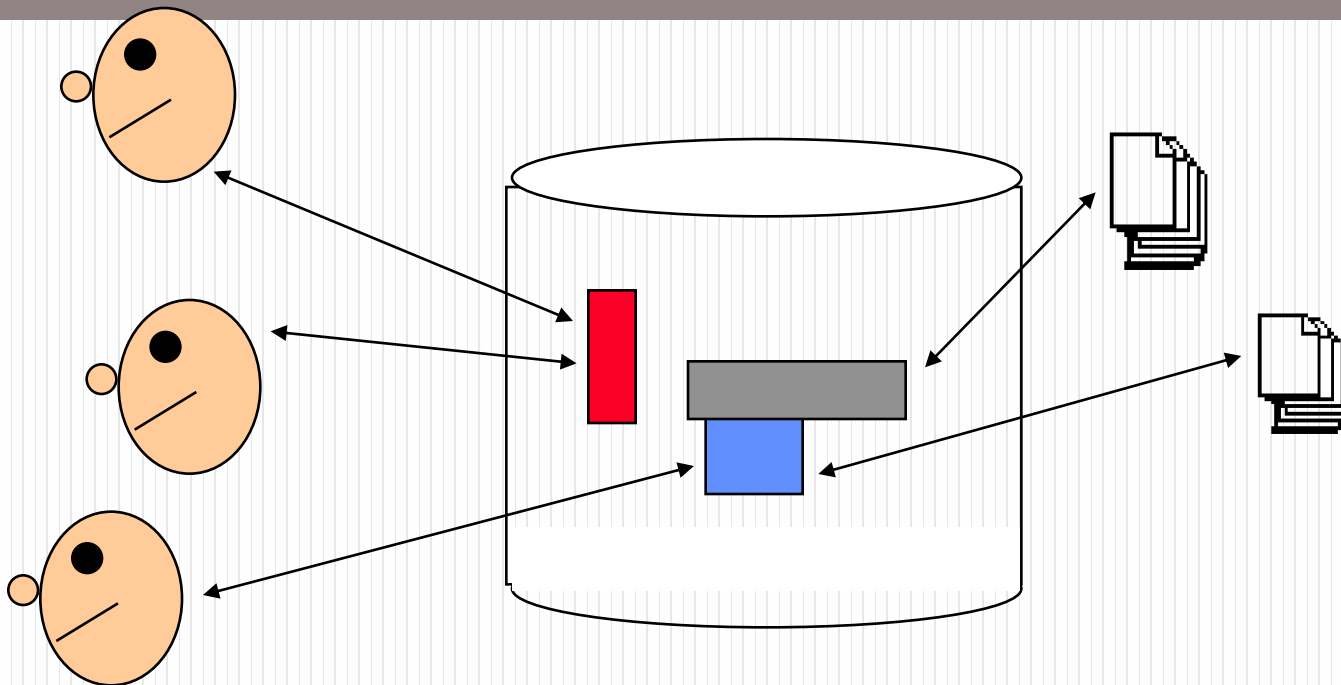
- `CREATE INDEX ON STUDENT (lower(nome));`
 - Permite pesquisas indiferentemente de se usar maiúsculas ou minúsculas
- `CREATE INDEX ON STUDENT(morada) USING hash;`
- `DROP INDEX index_name`

Utilização de índices

- A otimização de consultas decide da utilização de índices ou de varrimento sequencial (use EXPLAIN para verificar)
- Restrição em chave primária {aluno_id}
 - “aluno_id = 1234” o índice é usado porque só devolve no máximo um registo
- Restrição em chave primária {aluno_id, disciplina_id}
 - “aluno_id = 1234 and disciplina_id=88” usa o índice
 - “disciplina_id = 99” não usa o índice
 - “aluno_id = 1234” usa o índice, porque é a primeira coluna

Transações

Feliz Gouveia, UFP



Controlo de Concorrência

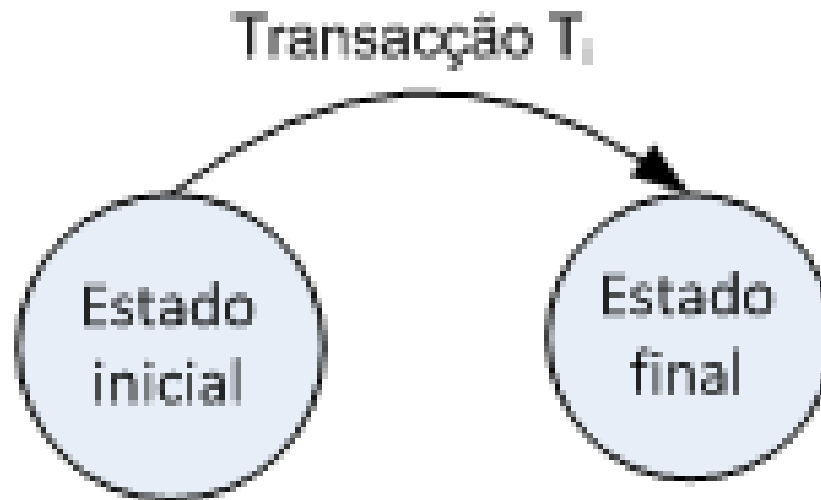
- Componente do SGBD responsável por garantir a integridade dos dados na presença de múltiplos acessos concorrentes aos mesmos dados
- Recebe as interrogações dos utilizadores e é responsável por escalonar a sua execução
- É invisível, cada utilizador pensa que tem a base de dados “só para si”
- Baseia o seu funcionamento no conceito de transação

Gestor de Transações

- É responsável por:
 - Aceitar a transação
 - Atribuir-lhe um identificador
 - Gerir as suas transições de estado
 - Terminar a transação

Transações

- Uma transação transforma a BD de um estado coerente noutro estado coerente (além das restrições de integridade deve respeitar também as restrições do negócio)



Transações (1)

- Uma transação é uma unidade lógica de trabalho: é uma operação “tudo ou nada”
- Uma transação consiste numa sequência de leituras e de escritas (*r* e *w*), terminadas por *c* (confirmação) ou *a* (interrupção)
- Uma transação é terminada por:
 - COMMIT (*c*): em caso de sucesso, todas as modificações na BD tornam-se permanentes
 - ROLLBACK (*a*): em caso de falha, todas as modificações são desfeitas

Transações (2)

- Vamos supor que se atualiza o custo em 10%
 - `update produto set custo = custo*1.1;`
- A meio da atualização ocorre uma falha; metade dos tuplos foram atualizados, a outra metade não
- A base de dados está num estado incoerente. A noção de transação impede que isso aconteça: como a transação foi interrompida, todos os tuplos voltam ao estado (valor) que tinham antes do início da transação

Transações (3)

- As transações não comunicam entre si
- No entanto, podem ler e escrever valores que são usados noutras transações
 - Por exemplo, numa livraria o valor QUANTIDADE da tabela PRODUTO é consultado e eventualmente atualizado por centenas ou milhares de transações a correr em simultâneo
- Significa que o SGBD não pode ignorar os efeitos de uma transação noutras transações (mas o utilizador pode!)

Transações (4)

- Uma transação T_i contém operações de leitura, $l_i(x)$, e de escrita, $e_i(x)$, sobre um elemento da base de dados x
 - Na prática podem existir operações como “incrementar” que implica ler e escrever
- Uma dessas operações diz-se **suja** se operar sobre valores não confirmados por outra transação
- Uma escrita diz-se **cega** se não efetuou uma leitura antes

Transações (5)

- Os comandos SQL também devem ser atômicos, uma vez que operam em conjuntos de linhas, e deveriam ser “tudo ou nada”
- As modificações podem ser refeitas ou desfeitas utilizando um “log”, ou um diário (“journal”)
- O log deve ser escrito fisicamente antes de o COMMIT ser efetuado (Write-Ahead Log: WAL) — em caso de acidente, a BD pode ser recuperada para um estado coerente

Transações – o log

- T_i : update ALUNO set numero=1234 where ID=25
- O registo de *log* pode conter a operação, ou os valores antes e depois:
- T_i antes: numero=123 depois: numero=1234
- Desfazer a transação consiste neste caso em repor o valor antes (dito “imagem antes”)

Memória de trabalho

- As páginas residem no disco
- Quando uma transação pede um elemento, o Gestor de Memória verifica se a página está em memória
 - Se não estiver, lê-a do disco
 - O Gestor de Memória usa geralmente LRU e variantes
- Uma página em memória pode estar a ser utilizada por várias transações
 - Algumas das quais a podem querer modificar

Transações: propriedades ACID

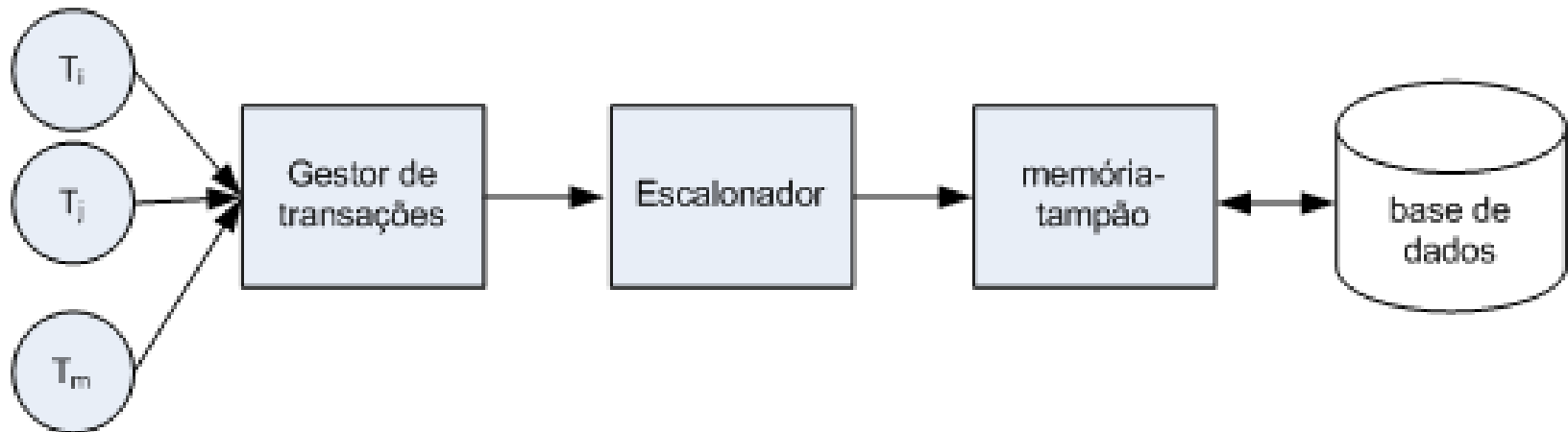
- **Atomicidade:** as transações são atômicas (“tudo ou nada”)
- **Consistência:** a BD passa de um estado consistente para outro estado consistente
- **Isolamento:** transações concorrentes “escondem-se” umas das outras, não interferem
- **Durabilidade:** em caso de sucesso, os seus efeitos são permanentes

Execução de transações

- Em série, umas após as outras: o resultado é sempre correto, não há interferência entre transações (mas não otimiza recursos)
- Concorrencialmente, as transações podem interferir
- Como garantir que o resultado de um conjunto de transações concorrentes é correto?

Componentes

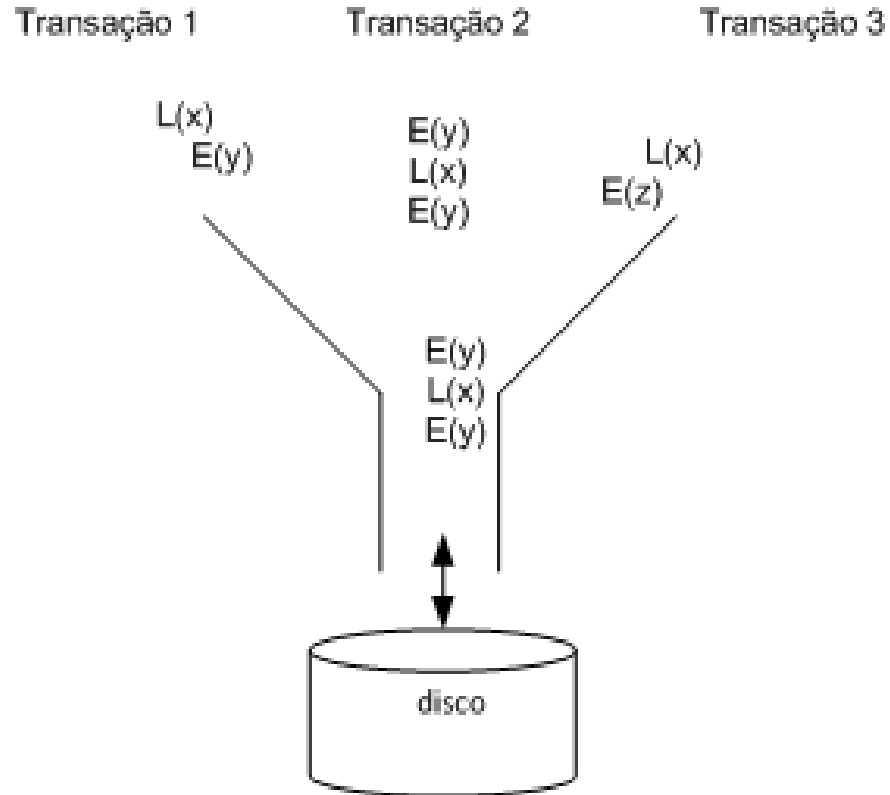
- O Gestor de Transações deve escalonar as operações que recebe das transações:



Escalonamento

- O Gestor de Transações recebe ações e produz um escalonamento das diferentes transações
- O escalonamento intercala as ações das diferentes transações, respeitando apenas a ordem dentro de cada transação
- Diferentes escalonamentos podem resultar no mesmo estado final
- Como garantir que um dado escalonamento é correto?

Execução concorrente



Critério de serialização

- É o critério de garantia de correção no controlo de concorrência
- Se a execução de um conjunto de transações for serializável, então é correto — isto é, produz o mesmo resultado que uma execução em série das mesmas transações
- Pretendemos então gerar escalonamentos que não sendo série, são “equivalentes” a um escalonamento série

Escalonamentos equivalentes

- Um escalonamento equivalente a um escalonamento série é serializável
- Dois escalonamentos são equivalentes se produzem o mesmo resultado final e:
 - O conjunto de transações é o mesmo
 - Garantem a sincronização “ler-escrever”, e “escrever-escrever”

Noção de conflito

- Um par de operações está em conflito se:
 - forem de transações diferentes,
 - operarem sobre o mesmo elemento e
 - pelo menos uma delas for uma escrita
- A existência de pares de operações em conflito não é por si um problema
 - Implica que a ordem das operações tenha que ser mantida (fazer uma escrita antes ou depois de outra operação dá resultados diferentes...)

Padrões de conflito

- Um conflito entre duas operações envolve o mesmo elemento e pelo menos uma delas é uma escrita:
- Leitura irrepetível: $I_1(x) \dots e_2(x) \dots I_1(x)$
- Leitura suja: $e_1(x) \dots I_2(x)$
- Atualização perdida: $e_1(x) \dots e_2(x)$

Equivalência a conflitos

- Garante que os pares de operações em conflito são ordenados da mesma forma que num escalonamento série
- É condição suficiente, mas não é necessária (é muito restritiva)
 - Muitos escalonamentos serializáveis são rejeitados por este critério
- Mas é a mais fácil de testar, pelo que é a usada na prática
 - Usam-se grafos de precedências

Transformar num escalonamento série

- A ideia é transformar um escalonamento num escalonamento série equivalente, trocando apenas de posição operações adjacentes que não estejam em conflito:
- $I_1(x) I_2(y) e_2(y) e_1(x)$
- $I_1(x) I_2(y) e_1(x) e_2(y)$
- $I_1(x) e_1(x) I_2(y) e_2(y)$
- É o escalonamento série $T_1 T_2$
- Logo, o escalonamento inicial é serializável

Grafos de precedências

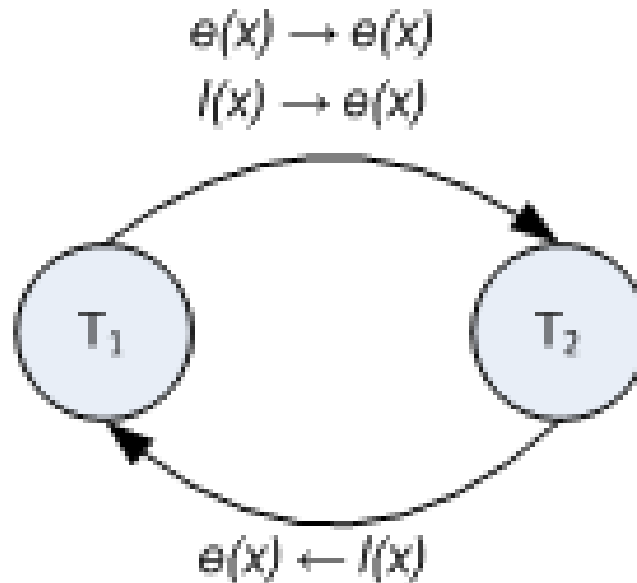
- Permitem testar a equivalência a conflitos
- Os nós são as transações
- Um arco entre T_i e T_j significa que uma operação de T_i precede e entra em conflito com uma operação de T_j
- Como vimos, um par de operações está em conflito se forem de transações diferentes, se operarem sobre o mesmo elemento e pelo menos uma delas for uma escrita

Teorema da serialização

- Um escalonamento E é serializável se o seu grafo de precedências não contiver ciclos
- Se não há ciclos, é sempre possível “ordenar” o grafo (logo, as transações)
- Se há um ciclo, não é serializável (e o ciclo tem que ser desfeito)
 - Repare-se que em grafos com centenas de transações detetar ciclos pode ser uma tarefa demorada (mas sempre realizável em tempo útil)

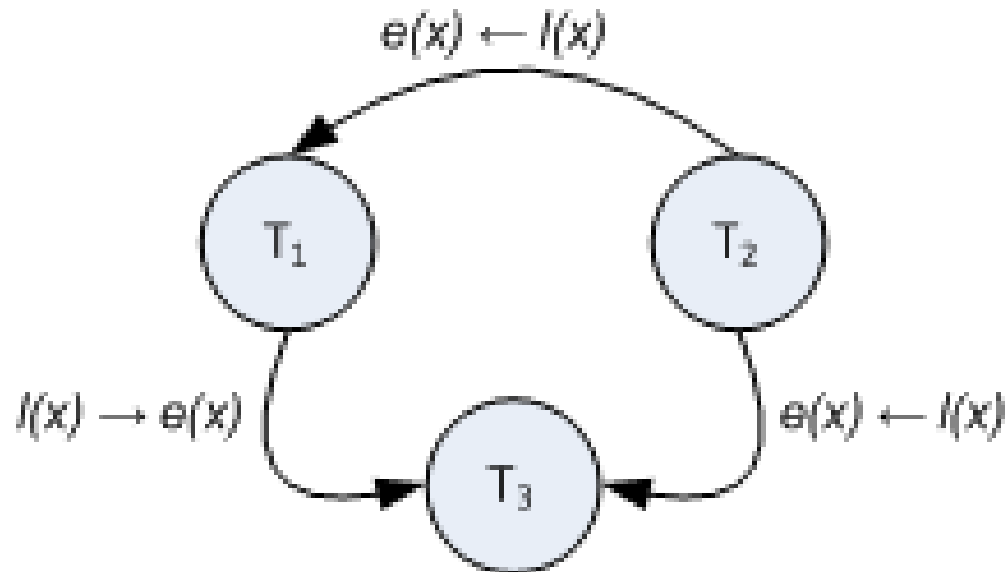
Escalonamento não-serIALIZÁVEL

- $E = I_1(x), I_2(x), e_1(x), c_1, e_2(x), c_2$



Escalonamento serializável

- $E = I_1(x), I_2(x), e_1(x), I_3(x), e_2(y), c_2, e_1(y), c_1, e_3(x), c_3$



Como resolver um ciclo?

- Na prática, retardar a execução de uma das operações para alterar a ordem
- Se necessário, interromper a transação causadora do ciclo, e recomencá-la mais tarde
- Esta é a tarefa principal do Gestor de Concorrência
- Note-se que o Escalonador tem conhecimento das operações apenas à medida que elas vão aparecendo, e decide o que fazer nessa altura, em tempo real (não pode esperar pelo fim das transações)

Outros critérios de serialização

- A serialização de vistas permite mais escalonamentos mas é mais difícil de testar



Serialização de vistas

- E_1 e E_2 são equivalentes a vistas se:
 - Todos os elementos têm os mesmos valores iniciais em ambos (mesmas leituras iniciais)
 - Se T_i lê um elemento x após T_j em E_1 , então T_i lê um elemento x após T_j em E_2
 - Para cada elemento x , a transação que o escreve por último em E_1 e E_2 é a mesma
- Testar a equivalência a vistas é em geral um problema NP-completo

Serializável a vistas mas não a conflitos

- Todo o escalonamento serializável a conflitos é serializável a vistas; o inverso não é verdade
- Um escalonamento serializável a vistas mas não a conflitos contém escritas cegas:
 - $E = I_2(y), e_1(y), e_1(x), e_2(x), e_3(x)$
 - Tem um ciclo entre T_1 e T_2
 - Mas é equivalente na prática a $T_2 T_1 T_3$

Mas...

- Não basta um escalonamento ser serializável
- Tem que respeitar outros critérios:
 - Por exemplo, em caso de falha de uma ou mais transações deve-se manter a correção do escalonamento

Qual é o problema?

- $I_1(x) \ e_1(x) \ I_2(x) \ e_2(x) \ c_2 \ a_1$
- O critério de serialização é respeitado
- A transação 1 é interrompida
- O que deve fazer a transação 2?
 - Afinal leu um valor que nunca existiu...
 - Mas já confirmou (não pode voltar atrás)

Recuperação

- Em caso de falha, deve ser possível recuperar o estado da base de dados
- Ou seja, desfazer transações em curso, e refazer transações confirmadas
- Se uma transação que leu valores sujos confirma, o que fazer se esses valores são desfeitos pela interrupção da outra transação?
 - O escalonamento não é recuperável...

Condição para recuperação

- Nenhuma transação pode confirmar se leu valores não confirmados por outras transações (valores sujos) antes de estas terminarem
- Dito de outra forma, se $e_i(x) < l_j(x)$ então temos que ter $c_i < c_j$
- Assim evita-se violar o contrato da transação
 - Uma transação que já confirmou não pode ser desfeita pelo efeito cascata

Interrupções em cascata

- Podem ser necessárias para recuperar valores lidos por uma transação
- Se uma transação lê valores de uma transação que é interrompida, então também tem que ser interrompida
- Por sua vez pode interromper outras (efeito cascata)

Evitar interrupções em cascata

- Evita desperdiçar recursos
- Uma transação não deve efetuar leituras sujas
- Se só ler valores confirmados, não é necessário desfazer nenhuma leitura devido a interrupção da transação que escreveu
- Dito de outra forma, temos que ter $e_i(x) < c_i < l_j(x)$

Recuperação com “imagens antes”

- $X = 5, e_1(x, 3) e_2(x, 1) a_1$
 - Não há leituras sujas, mas a recuperação com “imagem antes” implica que x volte a 5, perdendo-se a escrita de T_2
- $X = 5, e_1(x, 3) e_2(x, 1) a_1 a_2$
 - x volta a 3 (antes da escrita de T_2) mas devia voltar a 5
- Podem-se evitar estas situações impedindo escritas sujas. Ou seja, só se podem escrever valores confirmados por outras transações

Escalonamento Estrito

- Um escalonamento estrito adia todas as escritas e leituras de elementos não-confirmados (sujeitos) até que as outras transações que os escreveram terminem

Escalonamento rigoroso

- Um escalonamento é rigoroso se for estrito e se adicionalmente nenhuma transação escrever elementos lidos por outras transações até que estas terminem
- Tem a propriedade da ordem de serialização de duas transações em conflito ser a sua ordem de confirmação
- rigoroso \subset estrito \subset evita cascata \subset recuperável

Na prática

- Pode-se optar por não usar o critério de serialização (por este ser muito restritivo)
- Na prática, as transações podem apresentar diversos graus de isolamento entre elas
- Esses graus de isolamento são definidos em termos das anomalias que vamos ver a seguir
- Isto significa que o programador está consciente dos problemas que a sua transação poderá gerar

Porquê evitar serialização?

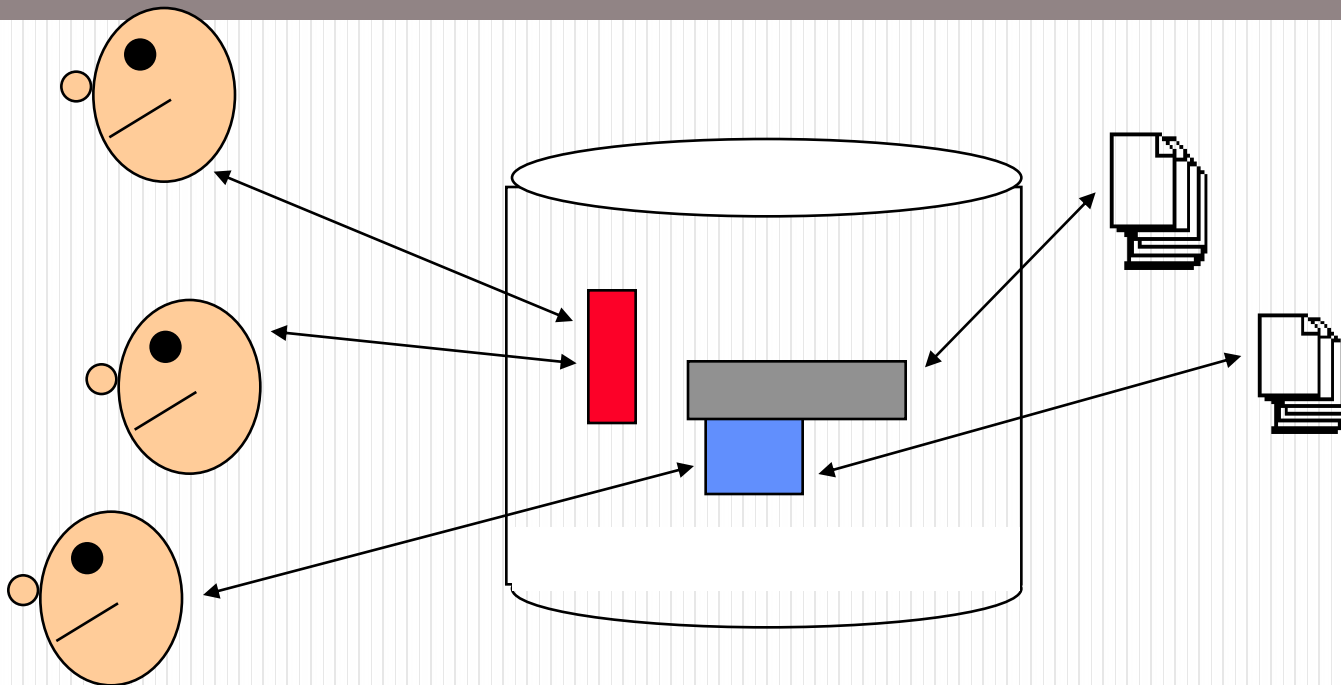
- Algumas situações não precisam de serialização; exemplo, não há problema se um valor estatístico tiver uma certa margem de erro
- Muitas transações podem ser de leitura, não dando origem a conflitos escrita-escrita
- Nalgumas situações podem aceder à mesma linha, mas modificar colunas diferentes (não há conflito)
- O ganho de desempenho pode compensar a eventual perda de consistência de alguns dados

A seguir

- Algoritmos do Escalonador

Controlo de Concorrência

Feliz Gouveia

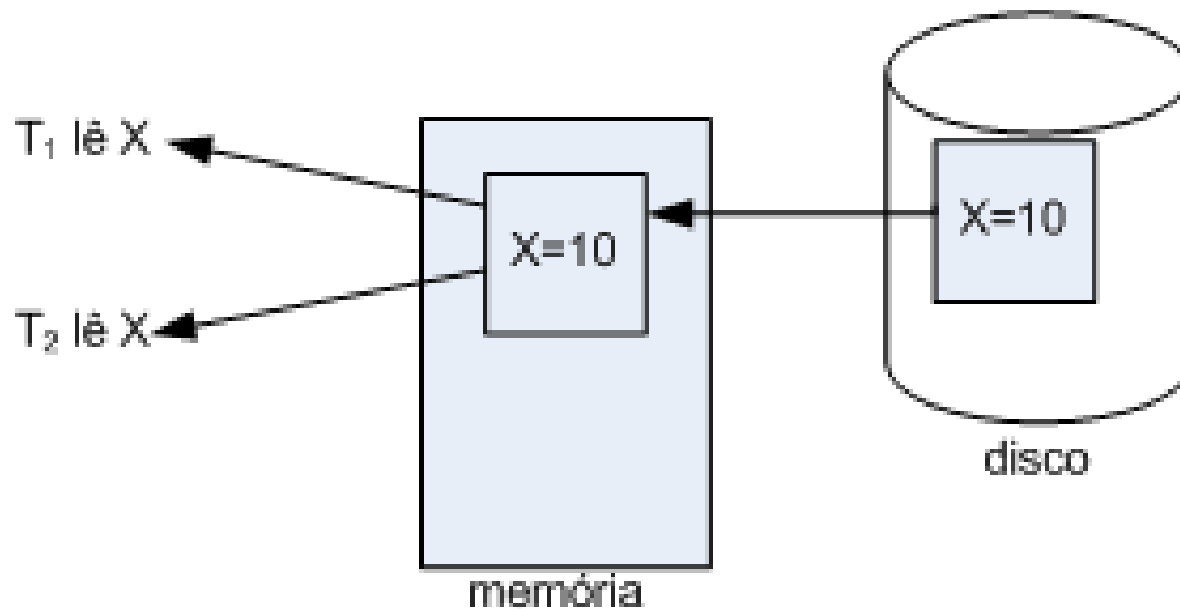


Controlo de Concorrência

- Componente do SGBD responsável por garantir que as transações não têm interferências indesejáveis quando manipulam os mesmos dados
- Essas interferências são definidas em termos das anomalias que devem ser evitadas
- Curiosamente, identificaram-se poucas anomalias

Transações

- Do ponto de vista de concorrência interessam apenas as Leituras e Escritas



Anomalias de concorrência

- Com vários utilizadores em simultâneo a trabalhar na BD três problemas podem ocorrer:
 - Atualização perdida
 - Dependência não realizada
 - Análise inconsistente
- Estes problemas implicam um estado final da base de dados que não seria possível se as transações fossem executadas em série
 - Logo o estado final é inconsistente

Atualização perdida

- T1 lê o elemento x
- T2 lê o elemento x
- T1 modifica o elemento x
- T2 modifica o elemento x (sobre a modificação de T1, que se perde)
- T1 confirma, T2 confirma
- T1 pode confirmar após modificar x, o efeito é o mesmo
- Conflito escrita-escrita

Atualização perdida

- T1 modifica x
- T2 modifica x
- T1 é interrompida (x volta ao valor inicial, a modificação de T2 é perdida)

Dependência não realizada (leitura suja)

- T1 lê e modifica x
- T2 lê (o novo) x
- T1 é interrompida (T2 trabalha com um valor errado, que “nunca existiu”)
- Conflito escrita-leitura

Análise inconsistente

- Uma transação está a somar saldos, outra está a transferir dinheiro: a soma dos 2 saldos é 120 ($A=80$, $B=40$).
- T1 lê conta A ($= 80$), e retira 20
- T2 lê conta A ($= 60$) e B ($=40$)
- T1 lê conta B e soma 20
- T2 lê um total de 100, valor incorreto uma vez que o saldo conjunto (120) permanece inalterado
- Conflito escrita-leitura

Leitura irrepetível

- T1 lê x, T2 escreve x, T1 lê x modificado
- `SELECT * FROM PAUTA WHERE NOTA > (SELECT AVG(NOTA) FROM PAUTA WHERE DISC_ID=23) AND DISC_ID=23;`
- Entre os dois SELECT pode ser atualizado um valor de NOTA para a disc_id=23, e o resultado pode não ser o esperado
- Conflito leitura-escrita

Recapitular

- O teste de correção de um escalonamento pode ser feito com um grafo de precedências
- O grafo de precedências tem um nó para cada transação, e um arco para cada par de operações em conflito pertencentes aos dois nós (transações)
- Um grafo acíclico garante que o escalonamento é correto (serializável)

Usar o grafo de precedências

- O teste de equivalência a conflitos não pode ser feito quando o escalonamento estiver completo (é muito tarde...ver o problema da análise inconsistente)
- O tempo de detecção de ciclos é elevado: é proporcional a N^2 onde N é o número de transações no grafo
- O grafo não pode ser modificado enquanto é percorrido (transações a iniciar, terminar, operações a chegar)

Solução

- A solução deve aliviar o programador de escrever código para sincronizar as operações (o que é sempre complexo, e suscetível de introduzir erros)
- Deve ser o próprio componente de Controlo de Concorrência a tratar de tudo
- Foram propostos diversos algoritmos
 - Devem ser rápidos
 - Devem impedir violação da serialização do escalonamento

Algoritmos propostos

- Fechos
- Certificação (validação)
- Ordenação temporal (OT)
- Multiversão (MVCC)
- Certificação é otimista, os outros são pessimistas (baseiam-se no pior cenário, isto é, a existência de conflitos que verificam em cada operação)
- MVCC é otimista no uso de versões; pessimista no uso de fechos

Fechos (Locking)

- Geridos pelo componente de Controlo de Concorrência
 - O programador não interfere
- Colocados em tuplos (e também em páginas, tabelas e base de dados)
- Dois tipos básicos de fechos:
 - exclusivos (E, ou escrita)
 - partilhados (L, ou leitura)

Algoritmo de Fechos

- É o algoritmo que insere comandos específicos para colocar fechos (lock) e para os libertar (unlock)
- Estes comandos servem apenas para escalonar as operações, e não são enviados para a base de dados
- NOTA: é possível também um utilizador inserir explicitamente fechos em tabelas (LOCK TABLE tabela)

Matriz de compatibilidade

Transação B
detém fechos

Transação A
pede fechos

	E	L	-
E	n	n	S
L	n	S	S

n: pedido de fecho negado

s: pedido de fecho autorizado

Protocolo de acesso

- As transações são “bem comportadas”:
 - A leitura de um tuplo requer um fecho L
 - Uma modificação requer um fecho E
- Se um pedido é negado, a transação fica em espera até este ser concedido
- Este protocolo básico pode ocasionar conflitos, e produzir escalonamentos que não são serializáveis (que veremos a seguir)

Promoção de fechos

- Para permitir mais concorrência, podem-se promover fechos de leitura para escrita apenas quando necessário
- Podem-se despromover fechos de escrita para leitura
- Estas operações de promoção e despromoção implicam a atuação do utilizador, o que é um inconveniente

O Gestor de Fechos

- O Gestor de Fechos gere listas
- Quando chega um pedido de fecho para x :
 - Junta um registo no fim da lista de fechos do elemento x , ou cria uma lista só com este pedido
 - O primeiro pedido é sempre concedido, os outros dependem da compatibilidade do fecho em causa

O Gestor de Fechos

- Quando chega um pedido para libertar um fecho em x:
 - Apaga o registo na lista do elemento x
 - Testa o registo seguinte e se possível concede o fecho (e processa o registo seguinte)
- Se uma transação é interrompida, apaga os seus pedidos pendentes. Quando a recuperação for efectuada, liberta os fechos detidos pela transação

O Gestor de Fechos

- Este protocolo evita situações de mingua (interrupções sucessivas da mesma transação), porque os fechos são processados por ordem de chegada
- Eventualmente qualquer pedido acaba por ver chegar a sua vez

Os 3 problemas revisitados

- T1 lê o tuplo x fecho L
 - T2 lê o tuplo x fecho L
 - T1 modifica x fecho E – espera por T2
 - T2 modifica x fecho E – espera por T1
-
- Resolve a “atualização perdida”, mas bloqueia eternamente numa espera circular (“deadlock”)

Dependência não realizada

- | | |
|---------------------|------------------|
| • T1 modifica x | fecho E |
| • T2 lê x | fecho L – espera |
| • T1 é interrompida | larga E |
| • T2 lê x | continua |
| | |
| • T1 modifica x | fecho E |
| • T2 modifica x | fecho E - espera |
| • T1 é interrompida | larga E |
| • T2 modifica x | continua |

Análise inconsistente

- Fica como exercício...

Problemas com o protocolo

- Libertar fechos antes do tempo pode provocar execuções não serializáveis
- $l_1(x)$, $l_2(x)$, $e_1(x)$, c_1 , $e_2(x)$, c_2

T2 liberta fecho

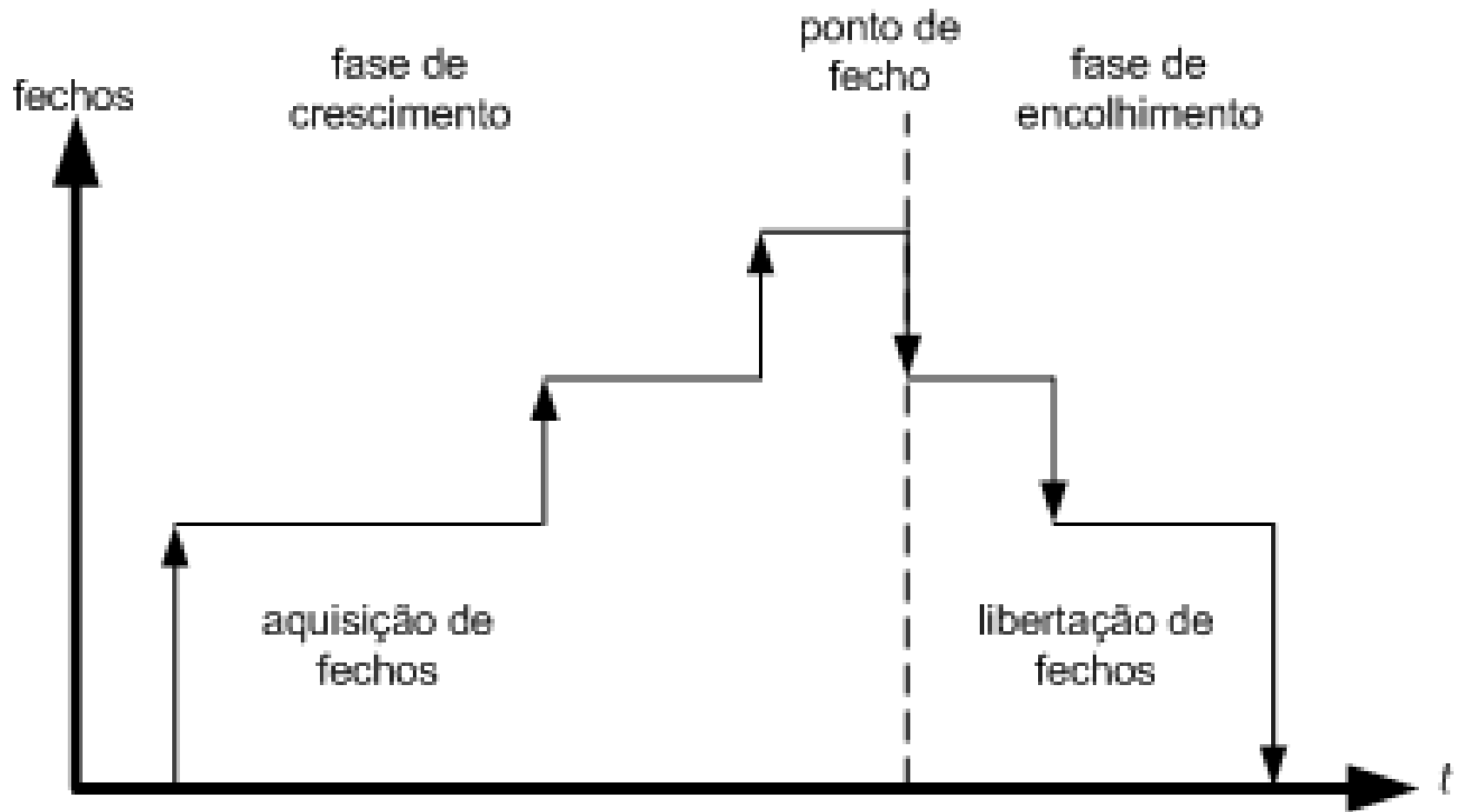
pedido de fecho
concedido a T1

- Na prática, os fechos podem ser de curta duração (a duração da operação), ou de longa duração (até ao fim da transação)

Teorema “Two-phase locking”

- “Se todas as transações obedecerem ao protocolo de fecho em 2 fases, todas as possíveis combinações temporais são serializáveis”
- Fase 1: aquisição de todos os fechos
- Fase 2: após libertar um fecho a transação não adquire mais nenhum
- A ordem de serialização é determinada pelo “ponto de fecho” (ver a seguir)
- Existindo promoção, só é permitida na primeira fase, e a despromoção na segunda fase

Fecho em 2 Fases (2PL)



2PL e Recuperação

- Como vimos, um escalonamento pode ser serializável, mas pode trazer problemas que impedem que seja recuperável
- A recuperação é uma propriedade importante de um escalonamento
- $fl1(x), l1(x), fe1(x), e1(x), -fe1(x), -fl1(x), fl2(x), l2(x), fe2(x), e2(x), c2, a1$
 - Este escalonamento é possível em 2PL, mas T2 leu um valor que nunca existiu
 - T1 não deveria ter libertado o fecho de escrita

2PL e Interrupções em cascata

- $fl1(x)$, $l1(x)$, $fe1(x)$, $e1(x)$, $-fe1(x)$, $-fl1(x)$, $fl2(x)$, $l2(x)$, $a1$
- A interrupção de T1 obriga a interromper T2
- 2PL foi respeitado porque T2 obteve um fecho em x (T1 já tinha libertado o fecho que detinha)
- Para permitir escalonamentos recuperáveis sem interrupções em cascata, os fechos exclusivos só devem ser libertados quando a transação termina (com ou sem sucesso)

Fase de libertação

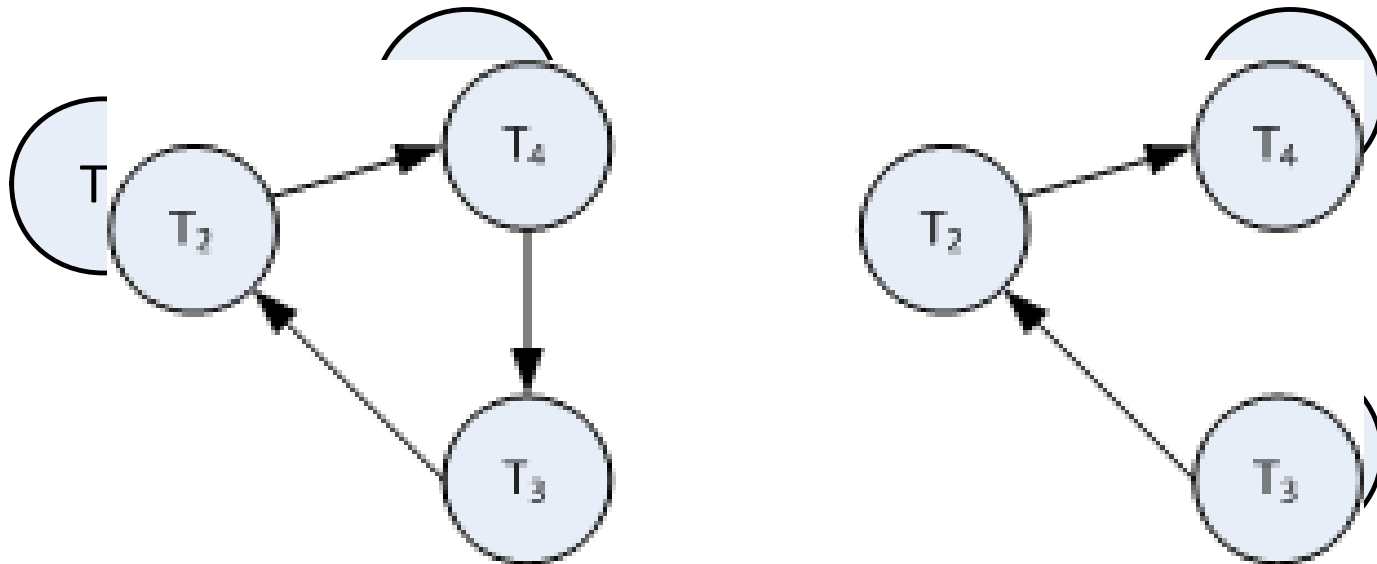
- 2PL Estrito: mantém os fechos exclusivos até ao fim; evita interrupções em cascata
- 2PL Rigoroso (SS2PL): mantém ambos os tipos de fechos até ao fim (Strong Strict 2PL)
 - A ordem de serialização é a ordem de confirmação
- Para a libertação ser feita de forma automática, SS2PL é a única opção
 - Caso contrário nunca se sabe se serão necessários mais fechos

Espera circular (Deadlock)

- Nenhuma transação consegue progredir, cria-se um impasse
- Pode ser detetada utilizando um grafo de dependências (quem espera por quem)
- Na prática usam-se temporizações para evitar esperas infinitas
- Uma das transações envolvidas na espera (a “vítima”) é interrompida e recomeçada mais tarde

Deteção de esperas circulares

- Usa-se um grafo de dependências; os nós são as transações, os arcos identificam quem espera por quem:



Resolução de ciclos

- Detecção: quando uma transação fica em espera verifica-se essa parte do grafo
- Prevenção: não se usa o grafo, nunca se permite espera, interrompe-se logo
- Temporização: uma transação colocada em espera é interrompida ao fim de um determinado tempo. Parametrizável, por exemplo:
 - -1, espera até ser concedido
 - 0, não espera, é interrompida
 - t , espera t segundos antes de ser interrompida

Escolha da vítima

- Na estratégia de detecção, pode-se escolher:
 - A transação que provocou a espera
 - Uma transação escolhida de forma aleatória
 - A transação detendo menos fechos
 - A transação mais recente
 - A transação que consumiu menos recursos
 - A transação envolvida em mais ciclos
- Os efeitos da vítima têm que ser desfeitos, o que pode ter custos elevados

Na prática

- São os algoritmos mais usados (DB2, SQL Server, MySQL/MariaDB)
- Atualmente apenas DB2 oferece apenas fechos, todos os outros oferecem também variantes de Multiversão (MVCC)
 - DB2 9 oferece *optimistic locking*
- Permitem um desempenho aceitável
- Todos os algoritmos implicam na prática uma certa dose de sintonização

Mais tipos de fechos

- Na prática todos os produtos propõem vários tipos de fechos
 - Fechos de “atualização” (update locks)
 - Fechos de “predicado” (next-key) que impedem a inserção de novos tuplos entre tuplos existentes
- Alguns produtos permitem “escalada” de fechos (fechos de “intenção”):
 - Um fecho pedido num tuplo pode ser escalado para a tabela se de repente forem necessários fechos em milhares de tuplos (5000 em SQL Server)

Fechos de Atualização (FA)

- Evitam esperas circulares devido a duas transações pedirem fechos de leitura e depois pedirem fechos de escrita (na ordem inversa)
- Só uma transação pode deter um FA o que resolve este problema
 - A transação promove depois o FA a um de escrita se pretender modificar o tuplo
- A nova matriz de compatibilidade é mostrada a seguir

Fechos de Atualização

		fechos existentes		
		FE	FA	FL
fechos pedidos	FE	recusado	recusado	recusado
	FA	recusado	recusado	concedido
	FL	recusado	concedido	concedido

Granularidade dos Fechos

- Coloca-se a questão a que nível é mais eficiente colocar fechos: tuplo, página, tabela
- Para manter um nível de concorrência aceitável, escolhe-se o tuplo mas...
- Ao colocar fechos em milhares de tuplos diminui-se a eficiência do Gestor de Fechos
- Pode ser conveniente neste caso colocar um fecho em páginas da tabela, ou na própria tabela: fechos de Intenção

Fechos de Intenção (FI)

- Hierarquia para colocar fechos: base de dados, tabela, página, linha
- Antes de obter um fecho, tem que deter um fecho FI nos ascendentes
- Para o libertar, liberta do nível mais específico para o mais geral
- Na mesma granularidade, FI só é compatível com FI
- Úteis para gerir escalada de fechos

Fechos de Intenção (FI)

- Sinalizam outras transações das intenções da transação
- Permitem detetar cedo possíveis situações de conflito
- Fechos de Intenção Partilhados (FIL)
- Fechos de Intenção Exclusivos (FIE)
- Fechos Partilhados de Intenção Exclusivos (FIEL)
 - Toleram pedidos de FIL

Matriz de compatibilidade

		fechos pedidos					
		FIL	FIE	FL	FIEL	FA	FE
fechos existentes	FIL	s	s	s	s		
	FIE	s	s				
	FL	s		s			
	FIEL	s					
	FA			s			
	FE						

Fechos de predicados

- Gerem os casos em que a base de dados é dinâmica (com remoção e inserção de tuplos)
- Nestes casos a serialização não pode ser garantida por nenhuma das variantes de 2PL
- Se o predicado não envolver um índice, tem de se obter um fecho de escrita em todas as páginas
- Se existir um índice, obtém-se um fecho para cada entrada do índice que satisfaz o predicado

Algoritmos baseados em grafos

- Alternativa a 2PL
- Define uma ordem parcial no conjunto de todos os itens (d_i) da base de dados
- Se $d_1 \rightarrow d_2$, então qualquer transação que aceda a ambos os itens deve fazê-lo nesta ordem
- O protocolo arborescente (Tree Protocol) é um exemplo de um protocolo baseado em grafos

Protocolo arborescente

- Só usa fechos exclusivos
- O primeiro fecho pedido por T_i pode ser em qualquer item; fechos subsequentes só são concedidos se T_i tiver um fecho no ascendente
- Os fechos podem ser libertados em qualquer altura, mas sobre esse item não pode ser pedido mais nenhum fecho por T_i
- Implicam padrões fixos de acesso aos dados (por exemplo, como no caso de uma B+-Tree)

Protocolo arborescente

- Produz escalonamentos serializáveis a conflitos e sem esperas circulares
- Fechos podem ser libertados mais cedo que em 2PL (mais concorrência)
- Não garante recuperação, nem é livre de interrupções em cascata
- Pode ser necessário obter fechos em itens que não são relevantes (diminuindo a concorrência)

Algoritmos de ordenação temporal

- Marca temporal: valor crescente (maior → mais recente) e único para cada transação
 - $TS(T)$, marca do início da transação T
- Cada elemento x tem duas marcas temporais: $R-TS(x)$ e $W-TS(x)$ — das transações que fizeram leitura e escrita mais recentes
- Na prática é equivalente a um escalonamento série pela ordem das marcas temporais das transações (ou seja, a ordem de serialização é determinada à partida)

OT: princípio

- Se p_i e q_j são duas operações em conflito,
 - Executa $p_i < q_j$ apenas se $TS(T_i) < TS(T_j)$
 - Caso contrário rejeita q_j e interrompe T_j
- A versão básica é agressiva: rejeita operações em conflito se não estiverem na ordem temporal das transações (e interrompe a transação que chegou “tarde”); senão executa-as imediatamente
- Respeitando esta regra, o grafo de precedências é sempre acíclico, e o escalonamento é serializável a conflitos

Regras da OT

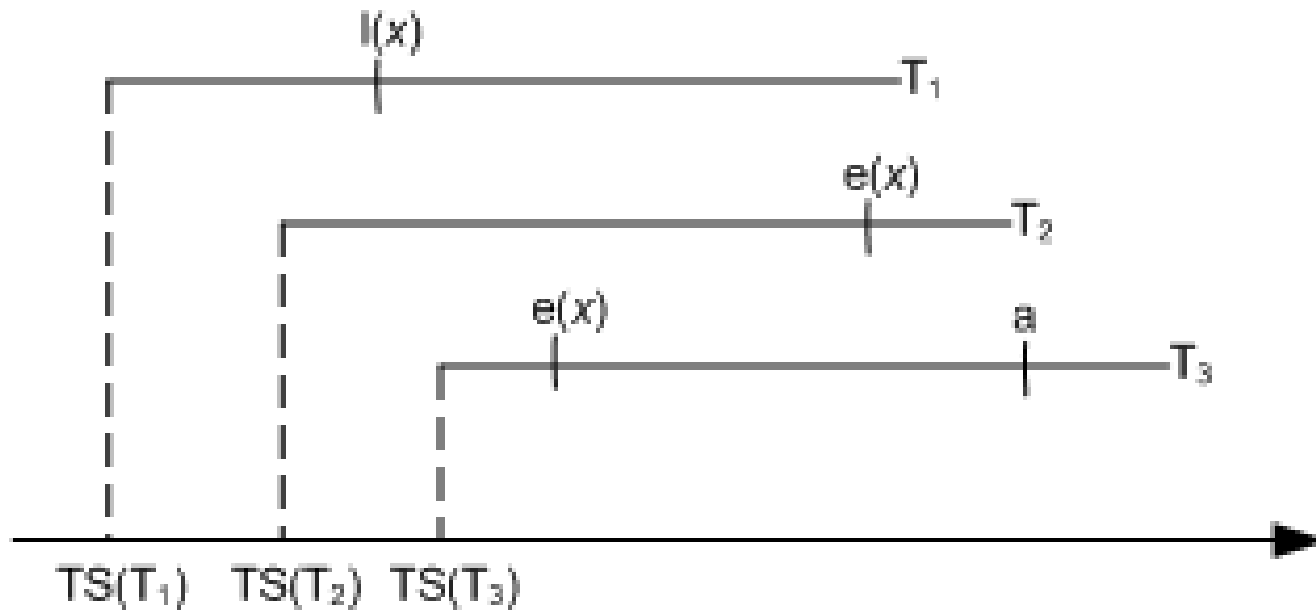
- T só pode ler x se $TS(T) \geq W-TS(x)$
 - e coloca $R-TS(x) \leftarrow \max(R-TS(x), TS(T))$
- T só pode escrever x se $TS(T) \geq W-TS(x)$ e $TS(T) \geq R-TS(x)$
 - e coloca $W-TS(x) \leftarrow TS(T)$
- Se a transação efectuar a operação, atualiza-se a marca temporal de x, caso contrário a transação T tem de ser interrompida e recebe uma nova marca
- Como as transações nunca ficam em espera, não há impasses (esperas circulares)

Regra de Escrita de Thomas

- Evita efetuar escritas que não vão ter efeito por já estarem desatualizadas
 - Outra transação que vai terminar depois vai escrever posteriormente o mesmo elemento
- Mantém-se a regra de leitura, a de escrita evita a interrupção no caso $TS(T) < W-TS(x)$ (ou seja, ignora a operação)
 - Se $TS(T) < R-TS(x)$, interrompe T
 - Se $TS(T) < W-TS(x)$ e $TS(T) \geq R-TS(x)$, ignora
 - Se $TS(T) \geq W-TS(x)$ e $TS(T) \geq R-TS(x)$, ok

Regra de Escrita de Thomas

- No entanto, a transação mais nova (T_3) pode ser interrompida, pelo que a escrita $e_2(x)$ ignorada passa a ser relevante (guarda-se)



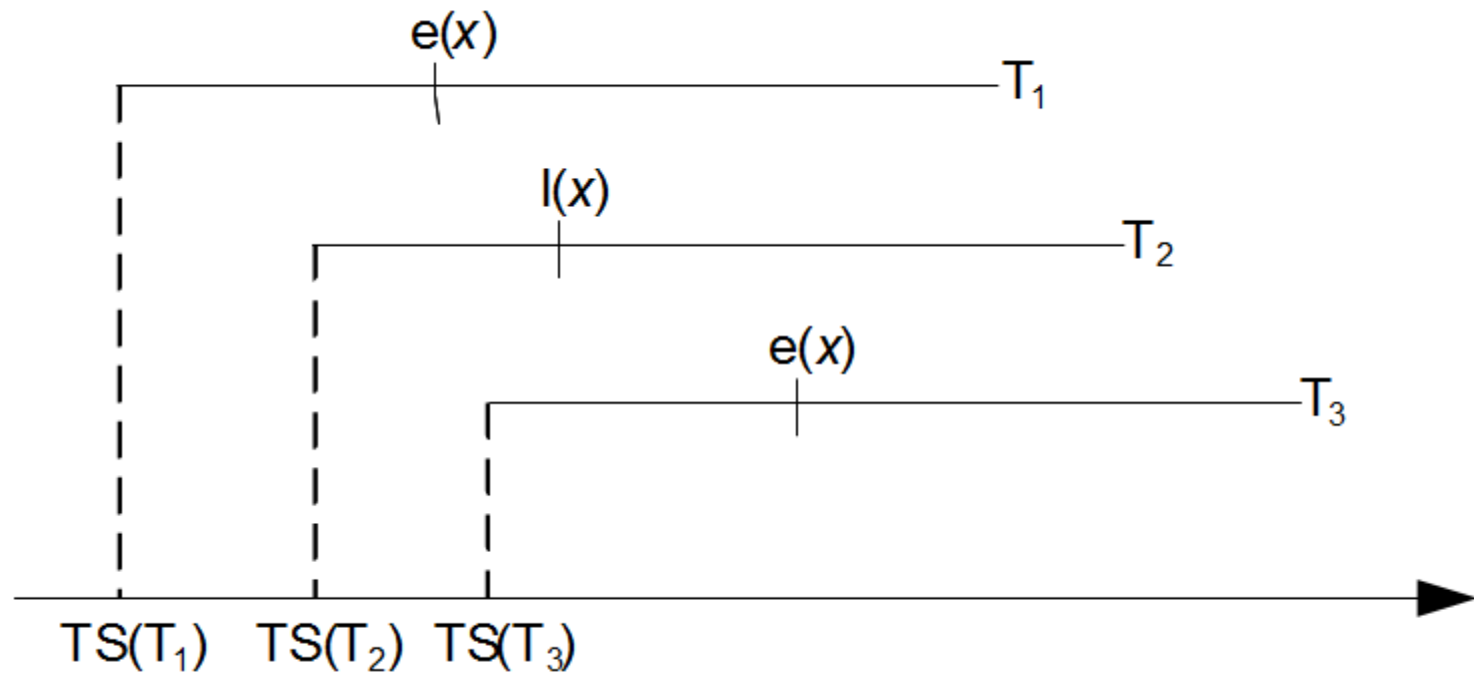
Exemplo de escalonamento

- $TS(T1) < TS(T2)$
- $I1(y) \ I2(y) \ e2(y) \ I1(x) \ I2(x) \ e2(x)$
- $TS(T1) < TS(T2) < TS(T3)$
- $e1(x) \ e3(x) \ I2(x)$
 - $I2(x)$ significa que T2 leu um valor que só será escrito mais tarde (T3 vai terminar depois) e por isso tem que ser interrompida

Na prática

- Estes algoritmos implicam muita gestão (duas marcas por linha, e atualização de marcas quando as operações são efetuadas)
- Pouco usados na prática, excepto em conjunto com outras técnicas
- Na versão básica o algoritmo admite escalonamentos não-recuperáveis em caso de falha (ver a seguir)
 - Chamado OTB (Ordenação Temporal Básica)

Exemplo



T_2 leu um valor não confirmado ($TS(T_2) \geq W-TS(x)$)

T_3 escreveu um valor não confirmado

Não-recuperável

- Escalonamento permitido, mas não recuperável:
 - $e_1(x) \mid e_2(x) \mid c_2 \mid a_1$

Solução

- Ordenação Temporal Estrita
- Produz escalonamentos recuperáveis
- Aos casos do algoritmo base junta-se a obrigação da transação esperar até que a transação que escreveu o elemento termine
 - Pode-se usar uma variável para cada elemento escrito, *escritas(x)*, que é colocada a 0 quando a transação que fez a escrita termina
- A seguir mostramos os problemas que um escalonamento estrito de OT evita

Ordenação Temporal Estrita

- Uma transação T não pode ler nem escrever elementos sujos.
- Leitura: se $W-TS(x) \leq TS(T)$, coloca T em espera até que a transação que escreveu x termine (evita leituras sujas)
 $esperas(x) \leftarrow T$
- Escrita: se $W-TS(x) \leq TS(T)$, coloca T em espera até que a transação que escreveu x termine (evita atualizações perdidas)
 $esperas(x) \leftarrow T$

Ordenação Temporal Estrita

- Adicionalmente queremos garantir que a ordem das operações, após serem enviadas para execução, não é trocada
- Por isso, além de testar as escritas feitas e confirmadas temos que testar também se todas as leituras que estavam em curso já foram feitas
 - Caso contrário $l_1(x)$ e $e_2(x)$ com $T_1 < T_2$ pode ser transformado em $e_2(x) l_1(x)$!!!

Algoritmo

- T quer ler x
 - se $TS(T) < W-TS(x)$ interrompe T
 - se $escritas(x) == 0 \ \&\& \ TS(T) < esperas(x)$
 - executa $l(x)$, $R-TS(x) \leftarrow \max(R-TS(x), TS(T))$
 - $leituras(x) \leftarrow leituras(x) + 1$
 - senão $esperas(x) \leftarrow T$, e depois aplica a regra
- T quer escrever x
 - se $TS(T) < R-TS(x)$ interrompe T
 - se $TS(T) < W-TS(x)$ ignora
 - se $escritas(x) == 0 \ \&\& \ leituras(x) == 0$
 - executa $e(x)$, $W-TS(x) \leftarrow TS(T)$, $escritas(x) \leftarrow 1$
 - senão $esperas(x) \leftarrow T$, e depois aplica a regra

Algoritmo (1)

- Quando uma transação T_i termina (confirma ou interrompe), coloca $escritas(x) \leftarrow 0$, $leituras(x) \leftarrow leituras(x) - 1$, $\forall x : w_i(x)$
- Se confirmou, as transações em espera podem prosseguir, reaplicando a regra de leitura ou de escrita
- Se interrompeu, as transações em espera devem testar as operações para determinar se ainda são válidas

OT Estrita e 2PL

- Comportamento é similar a 2PL
 - Elementos recebem um “fecho” até ao fim da transação
 - Mas produzem escalonamentos diferentes. O seguinte é OT mas não 2PL:
 - $l_2(x), e_3(x), c_3, e_1(y), c_1, l_2(y), e_2(z), c_2$ (T1 T2 T3)
 - $l_1(x), e_2(x), c_2, l_1(y), c_1$ (T1 T2)
 - Válido em 2PL mas não em OT
 - $l_2(x), e_3(x) c_3 e_2(x) c_2 l_1(x)$
 - $l_2(x), c_2, e_1(x), c_1$

Gestão das marcas

- É necessário guardar numa estrutura 2 marcas por cada elemento acedido
- Ao fim de algum tempo a tabela de marcas pode crescer e ser difícil de gerir
- Guarda a marca da transação mais antiga $TSa(T)$. Quando ela termina, podem-se remover as marcas nos elementos x tais que:
 - $TSa(T) > W-TS(x)$ e $TSa(T) > R-TS(x)$

Algoritmos Otimistas

- Só verificam conflitos no momento da confirmação, esperando que eles não existam
- Logo, são mais rápidos, porque evitam o trabalho de constantemente verificar, interromper e recomeçar transações
- Seguem o princípio de que “é mais fácil pedir desculpa mais tarde (se algo correr mal), do que pedir permissão agora”

Fases do controlo otimista

- Leitura dos elementos: são lidos pela transação, e as escritas são feitas nesta cópia local à transação (cópia privada)
- Validação: verifica se a transação que quer confirmar é serializável
- Escrita: se a fase anterior tem sucesso, a escrita é feita na base de dados, caso contrário a transação é recomeçada

Fases

- A fase de leitura prossegue sem verificação, de forma otimista
- As fases de validação e de escrita ocorrem numa secção crítica (sem interrupção)
- A fase de validação pode ocorrer de duas formas: direta e inversa
- Se a fase anterior tiver sucesso, a transação pode escrever e confirmar
- Como as escritas só acontecem quando as transações terminam, evitam-se interrupções em cascata

Validação direta (“futura”)

- Seja T_v a transação que quer terminar, e T_i o conjunto das transações ativas
- Há um conflito grave se $WS_v \cap RSi \neq \emptyset$
 - T_v ou uma das T_i devia ser interrompida
 - A serialização é determinada pela ordem de entrada na fase de Validação. Por isso não é necessário validar contra as escritas do conjunto de T_i
 - T_v precede necessariamente todas as T_i e como as escritas são locais não interferiram com T_i

Condição de serialização

- $T_v < T_i$ é serializável se:
 - Não há dependências de leitura: as escritas de T_v não afectam as leituras de T_i
 - Não há reescritas: as escritas de T_v não reescrevem as escritas de T_i
- A condição 2 é sempre garantida porque as escritas são feitas sequencialmente numa secção crítica: as de T_i só acontecem depois das de T_v

Validação direta

- Transações apenas de leitura nunca são interrompidas
- Resulta em serialização de conflitos
- Pode provocar interrupções desnecessárias
- Considere-se: $I_1(x)$ $e_1(x)$ $I_2(x)$ v_1 c_1 v_2
- Como T1 escreveu durante a fase de leitura de T2, T2 tem que ser interrompida
- No entanto este é o escalonamento T1 T2
- É simples mas “pessimista”

Validação direta

- Uma possibilidade consiste em “ajustar” a ordem das operações para que o conflito desapareça
- Os algoritmos otimistas com ajuste dinâmico da ordem de serialização têm as 3 seguintes situações de conflito a considerar
- O ajuste dinâmico usa uma marca temporal que determina a ordem de serialização: permite às transações saber se têm de repetir operações que necessitem ajustamento

Validação direta: casos

- Seja T_v a transação que quer terminar, e T_i o conjunto das transações activas
- $RS_v \cap WSi \neq \emptyset$, não tem problema desde que se garanta $T_v < T_i$
- $WS_v \cap RSi \neq \emptyset$, tem problema mas $T_i < T_v$
- $WS_v \cap WSi \neq \emptyset$, não tem problema: $T_v < T_i$
 - As únicas situações que podem provocar interrupção são aquelas em que se verifica 1 e 2, ou 3 e 2 (conflito sério).

Validação direta: casos

- Os casos 1 e 3 respeitam a ordem de serialização ($T_v T_i$), por isso não é necessário interromper transações
- O caso 2 pode ser resolvido se a leitura vier antes da escrita ($T_i T_v$), mas neste caso esta ordem é inversa da ordem de serialização (confirmação)
- Por isso, o caso 2 nunca pode ocorrer em simultâneo com o 1 ou com o 3; caso contrário o conflito é sério

Validação inversa

- Necessita de guardar os WS das transações que terminaram desde que T_v começou (na validação direta não é necessário)
- Há um conflito grave se $RS_v \cap WSi \neq \emptyset$
- Se houver conflitos apenas a transacção T_v em fase de validação é interrompida (na validação direta podia ser qualquer uma)
- Implica muita informação a guardar, e não permite escolha da transacção a interromper (é sempre T_v)

Míngua (*starvation*)

- Este problema ocorre quando uma transação é repetidamente interrompida, nunca conseguindo terminar
 - Exemplo, transações longas
- Solução: ao fim de N tentativas, bloqueia a BD só para si, ou
- Cria-se uma transação substituta que reserva os dados só para si, e faz as outras transações entrar em conflito

Utilização prática

- Bases de Dados Tempo Real (BDTR) devido à flexibilidade na escolha da vítima, e ao ajuste dinâmico
- IMS FastPath (IBM)
- GemStone, uma BDOO
- Phyrro, www.pyrrhodb.com

Algoritmos Multiversão

- Nestes casos, cada escrita de um elemento origina uma nova versão, em vez de substituir o valor anterior (evitam-se assim os conflitos escrever-escrever)
- Podem usar marcas temporais, ou podem ser baseados em fechos
- Particularmente adaptados a ambientes de leituras intensivas (web, análise de dados)
- Recentemente apareceu *Snapshot Isolation* (SI)

Multiversão: princípios

- Traduz as operações em operações sobre versões dos elementos
- Como vimos, uma leitura é normalmente rejeitada se o valor que quer ler já foi reescrito
- Com MV, permite-se a leitura do valor antigo e evita-se a interrupção
- As versões não estão visíveis para as transações, apenas para o escalonador
 - O SQL é o mesmo!

Algoritmos Multiversão

- Bastante populares:
 - Oracle (nativo)
 - PostgreSQL (nativo)
 - MySQL/MariaDB
 - SQL Server 2005 e 2008
- DB2, Sybase não têm Multiversão
- Podem ser implementados com diferentes técnicas (e variantes)

Implementações

- PostgreSQL: guarda todas as versões de uma linha na base de dados
 - Implica limpezas regulares das versões não necessárias (ver utilitário *vacuum*)
- Oracle/MySQL/MariaDB: usam a informação dos *logs* de recuperação para encontrar uma dada versão
 - Esta informação já era guardada para efeitos de recuperação pelo que não implica gastar recursos adicionais para fazer MVCC
- SQL Server usa *tempdb*

Multiversão por Ordenação Temporal

- $TS(T_i)$: marca temporal da transação, atribuída no seu início
- Cada elemento x tem associadas versões
- Cada versão x_k tem:
 - O seu valor
 - $W-TS(x_k)$: TS da transação que o escreveu
 - $R-TS(x_k)$: TS da última transação que o leu
 - Bit que indica se a transação já confirmou

Serialização MVOT

- Quando uma transação T_i quer:
 - $l(x)$, lê x_k onde a versão x_k é a versão escrita mais recente $W-TS(x_k) \leq TS(T_i)$
 - $e(x)$, se $W-TS(x_k) < TS(T_i) < R-TS(x_k)$, T_i é interrompida. Se $TS(T_i) = W-TS(x_k)$, reescreve-o. Caso contrário, cria uma nova versão
 - c , espera até que todas as T_j que escreveram versões lidas por T_i terminem
- Equivalente a uma execução por ordem das marcas temporais das transações

Análise MVOT

- As leituras nunca falham (vantagem)
- As escritas podem implicar interrupção, em vez de espera (inconveniente)
- A versão básica MVTO não permite recuperação (T não pode confirmar se leu valores sujos)
 - A solução é semelhante à da ordenação temporal (ver diapositivo anterior)

Multiversão por 2PL

- Tenta combinar o melhor de dois mundos
- As transações têm uma marca de início e outra de confirmação
- A marca de confirmação é um contador de versões (Tsc), incrementado nas confirmações
- Cada elemento tem o $W-TS = Tsc$ do seu criador (isto é, a marca da sua confirmação)
 - Existe uma versão confirmada e várias em atualização

MV2PL

- As leituras nunca bloqueiam, e seguem as regras MV; uma transação só de leitura tem um TS, e lê os valores cujo W-TS é o maior inferior a TS(Ti)
 - Como a marca de um elemento é a da confirmação do seu criador, só lê versões confirmadas (evita leituras sujas)
- As transações de escrita usam 2PL rigoroso (guardam todos os fechos até ao fim). São serializadas pela ordem de confirmação

2V2PL

- Duas versões: uma confirmada e no máximo uma não confirmada
- Para escrever pede um fecho de escrita, e depois escreve uma nova versão
- $W-TS(x)$ é inicialmente colocado a ∞
- Quando quer confirmar:
 - Coloca todos os $W-TS$ a $TSc+1$
 - Incrementa TSc
- Tx que comecem depois de Ti terminar já vêm as novas versões; as que começaram antes lêem a versão anterior

2V2PL (e MV2PL)

- Três tipos de fechos: leitura, escrita, e certificação
- Leituras e escritas não se bloqueiam
- Escrita é incompatível com escrita
- As leituras nunca esperam, leem valores mais antigos confirmados
- As escritas escrevem uma nova versão

Compatibilidade de Fechos

		fechos pedidos		
		FL	FE	FC
fechos existentes	FL	concedido	concedido	recusado
	FE	concedido	recusado	recusado
	FC	recusado	recusado	recusado

MV2PL

- Quando uma transação quer confirmar pede um fecho de certificação (exclusivo)
- Como o fecho é exclusivo, a transação espera até que todas as transações terminem a leitura desse elemento
- Produz escalonamentos recuperáveis e evita interrupções em cascata
- Em MV2PL os fechos de escrita não são exclusivos, e podem existir N versões. A certificação demora mais tempo

Snapshot Isolation

- Cada transação vê um instantâneo da BD, no instante em que foi iniciada (o que significa: quando a primeira operação for executada)
- As escritas são locais a cada transação
- Quando uma transação quer terminar, verifica-se a existência de conflitos (semelhante ao controlo otimista)
- Leitura não bloqueia escrita, e vice-versa
- Adotado por Oracle 7+, MS SQL Server 2005+, MySQL/MariaDB, PostgreSQL

SI: definição teórica

- Ti só é autorizada a terminar se nenhuma transação que tiver executado concorrentemente com Ti tiver escrito e confirmado elementos que Ti quer escrever
- Caso contrário Ti é interrompida
- Regra da “primeira transação a confirmar ganha” (semelhante ao controlo otimista)
- Há uma variante chamada “primeira transação a atualizar ganha” (usada em Oracle e PostgreSQL)

SI: Primeira Atualizar Ganha

- T recebe uma marca temporal TS no início
- I(x): se há uma versão criada por si, lê-a, senão lê a versão confirmada mais recente $\leq TS$
- e(x): se há uma versão criada por si, reescreve; se há uma versão criada após TS (uma transação concorrente confirmou), T é interrompida; caso contrário pede um fecho — se a transação que detém o fecho confirma, T é interrompida (evita a “atualização perdida”)

Anomalias de SI

- Desvio de escrita (write-skew)
 - Dependência subtil entre elementos: tabela com cores “preto” e “branco”. T1 troca branco por preto; T2 troca preto por branco; qualquer execução serializável resulta numa só cor; em SI pode não resultar porque cada transação vê um instantâneo diferente da outra
- Anomalia só de leitura
 - Ver exemplo

Solução: Serializable SI

- SI não deve ser usado como nível “serializable” (mas era em PostgreSQL antes da versão 9.1, e é em Oracle !)
- Modificação do algoritmo (SSI) adiciona verificações adicionais para detetar dependências entre as transações
- Corresponde a “serializável”
 - Obtém-se no nível “serializable” em PostgreSQL 9.1+

Níveis de isolamento SQL

- No nível máximo, não há interferência entre transações
- Na prática, e para aumentar a concorrência, baixa-se o nível de isolamento
- Na prática, poderão ocorrer raramente conflitos, ou estes poderão ser aceitáveis
 - Compete ao programador definir o nível de isolamento que pretende para uma determinada transação

Problemas a evitar

- dirty read
 - Ler dados modificados mas não confirmados por outra transação
- non-repeatable read
 - Rer ler dados entretanto confirmados por outra transação (que os confirmou desde a leitura inicial)
- phantom read
 - Refazer uma interrogação e verificar que o conjunto de tuplos devolvidos é diferente porque outra transação confirmou modificações (por exemplo fez um INSERT)

Níveis de isolamento SQL

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possível	Possível	Possível
Read committed	Impossível	Possível	Possível
Repeatable read	Impossível	Impossível	Possível
Serializable	Impossível	Impossível	Impossível

SQL99: este nível tem que ser equivalente a uma execução série

Níveis adicionais

	dirty read	cursor lost update	lost update	non-repeatable read	phantom read
read uncommitted	possível	possível	possível	possível	possível
read committed	impossível	possível	possível	possível	possível
cursor stability	impossível	impossível	por vezes	por vezes	possível
repeatable read	impossível	impossível	impossível	impossível	possível
snapshot	impossível	impossível	impossível	impossível	por vezes
serializable	impossível	impossível	impossível	impossível	impossível

Níveis em termos de fechos

- Read uncommitted: apenas fechos de escrita de longa duração
- Read committed: fechos de leitura de curta duração e fechos de escrita de longa duração
- Repeatable read: fechos de leitura de longa duração e fechos de escrita de longa duração
- Serializable: fechos de leitura e de escrita de longa duração, e fechos de predicados de longa duração

Níveis em termos de Snapshots

- Snapshot: neste nível o instantâneo é válido até ao fim da transação. Se existirem conflitos de escrita, a primeira a atualizar ganha e a outra transação é interrompida.
- Read Committed: se houver um conflito de escrita, pede um fecho — se a transação que detém o fecho confirma, volta a testar o predicado e se continuar aplicável, atualiza o instantâneo, e escreve. Resulta na utilização de linhas que não estavam no instantâneo inicial

Interpretações

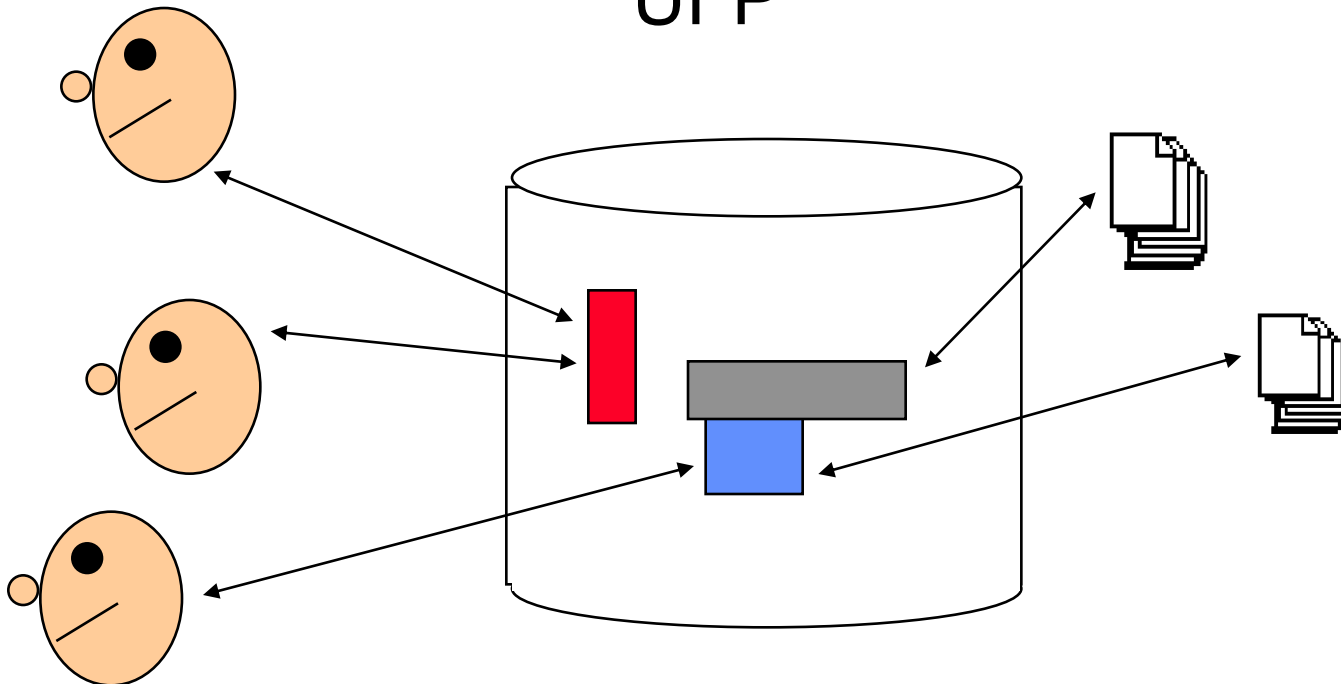
- Produtos baseados em MVCC têm comportamento diferente
- MySQL/MariaDB em Repeatable Read: as leituras são repetíveis, mas as atualizações são feitas sobre os dados confirmados (podem ser posteriores ao início da transação) – e passam a ser visíveis na transação embora não fizessem parte do instantâneo inicial....

Interpretações

- MS SQL Server oferece MVCC de duas formas:
 - Read Committed with Row Versioning: cada leitura usa um novo instantâneo
 - Snapshot Isolation: o instantâneo é tirado no início da transação e mantém-se
- Oracle oferece Read Consistency (MV Read Committed): as escritas não são interrompidas mas bloqueadas
- PostgreSQL usa MVCC, SI em Read Committed e Repeatable Read, e SSI em Serializable
 - O primeiro produto a implementar SSI

Recuperação

Sistemas de Gestão de Bases de Dados
Feliz Gouveia
UFP



Introdução

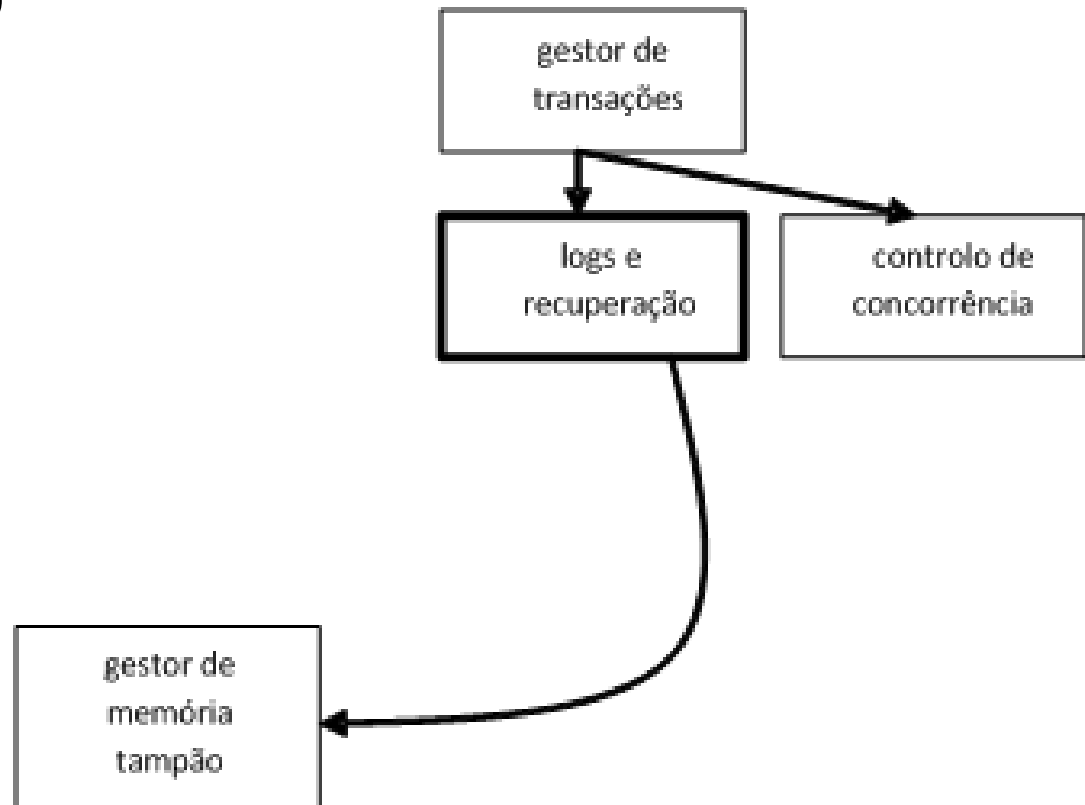
- ❑ Um SGBD tem que ser tolerante a falhas
 - ❑ Isso significa que uma base de dados não fica corrompida como resultado de uma falha
- ❑ A recuperação é a capacidade do SGBD de, após uma falha, reverter o seu estado, possivelmente corrompido, para um estado conhecido e consistente
- ❑ O programador não tem que se preocupar diretamente com concorrência ou recuperação
- ❑ Mas o administrador tem que se preocupar com as cópias de salvaguarda...

Objetivo da recuperação

- ❑ Garantir que em caso de falha se pode manter a Atomicidade e a Durabilidade
 - ❑ Atomicidade: garantida se se puderem desfazer os efeitos de transações não confirmadas (*rollback*)
 - ❑ Durabilidade: garantida se se puderem refazer os efeitos de transações confirmadas (*roll-forward*)
- ❑ As falhas podem ser acidentais, ou ser provocadas (o Gestor de Transações pode interromper transações: por exemplo em 2PL)

Componentes

A recuperação é uma função importante da Gestão de Transações e é garantida pelo Gestor de Recuperação



O Gestor de Recuperação

- ❑ Emite pedidos de leitura e de escrita de páginas para o Gestor de Memória
- ❑ *Fetch*: pede uma página. Se necessário o Gestor de Memória acede ao disco
- ❑ *Flush*: pede uma escrita de página.
- ❑ Processa as operações de Confirmação e de Interrupção de transações, e de Arranque da base de dados (após falha)

Fases da recuperação

- ❑ Qualquer algoritmo de recuperação trabalha a dois níveis:
 - ❑ O que precisa fazer durante o processamento normal das transações
 - ❑ O que precisa fazer após a falha para trazer a base de dados para um estado consistente

Concorrência e Recuperação

- ❑ Os escalonamentos devem prever a possibilidade de recuperação
- ❑ Mas como vimos um escalonamento serializável pode não ser recuperável
- ❑ Vamos rever estes conceitos a seguir

Serialização e recuperação

- ❑ Escalonamento serializável:

$$E = l_1(x), e_1(x), l_2(x), e_2(x), c_2, a_1$$

- ❑ Mas...
- ❑ Não há solução simples:
 - ❑ desfazer T2: viola o contrato de C2
 - ❑ Manter T2: viola a consistência das leituras
- ❑ Este escalonamento não é recuperável

Escalonamento “recuperável”

- ❑ Uma transação só pode confirmar depois de todas as transações de onde leu dados sujos terminarem
- ❑ Ou seja, para quaisquer T_i e T_j : se T_j leu dados escritos por T_i , T_j só pode confirmar depois de T_i confirmar

Interrupções em cascata

- ❑ Escalonamento serializável e recuperável:

$$E = I_1(x), e_1(x), I_2(x), e_2(y), a_1$$

- ❑ É recuperável, mas a interrupção de T_1 provoca a interrupção de T_2
- ❑ Desperdício de recursos, e de tempo
- ❑ T_2 pode provocar por sua vez interrupção de mais transações (efeito cascata)

Escalonamento “evita interrupções em cascata”

- ❑ Uma transação só lê elementos escritos por transações confirmadas
- ❑ Dito de outra forma, todas as operações de leitura ficam em espera até que as transações que modificaram os elementos terminem

Recuperação com “imagens antes”

- ❑ Escalonamento serializável, recuperável, sem interrupções em cascata:

$$E = e_1(x), e_2(x), a_1$$

- ❑ Ao repor o valor antes de $e_1(x)$, perde-se a escrita $e_2(x)$...
- ❑ Resolvido se uma transação só puder escrever elementos modificados por transações confirmadas
- ❑ Esta condição permite repor “imagens antes” de cada modificação em caso de interrupção

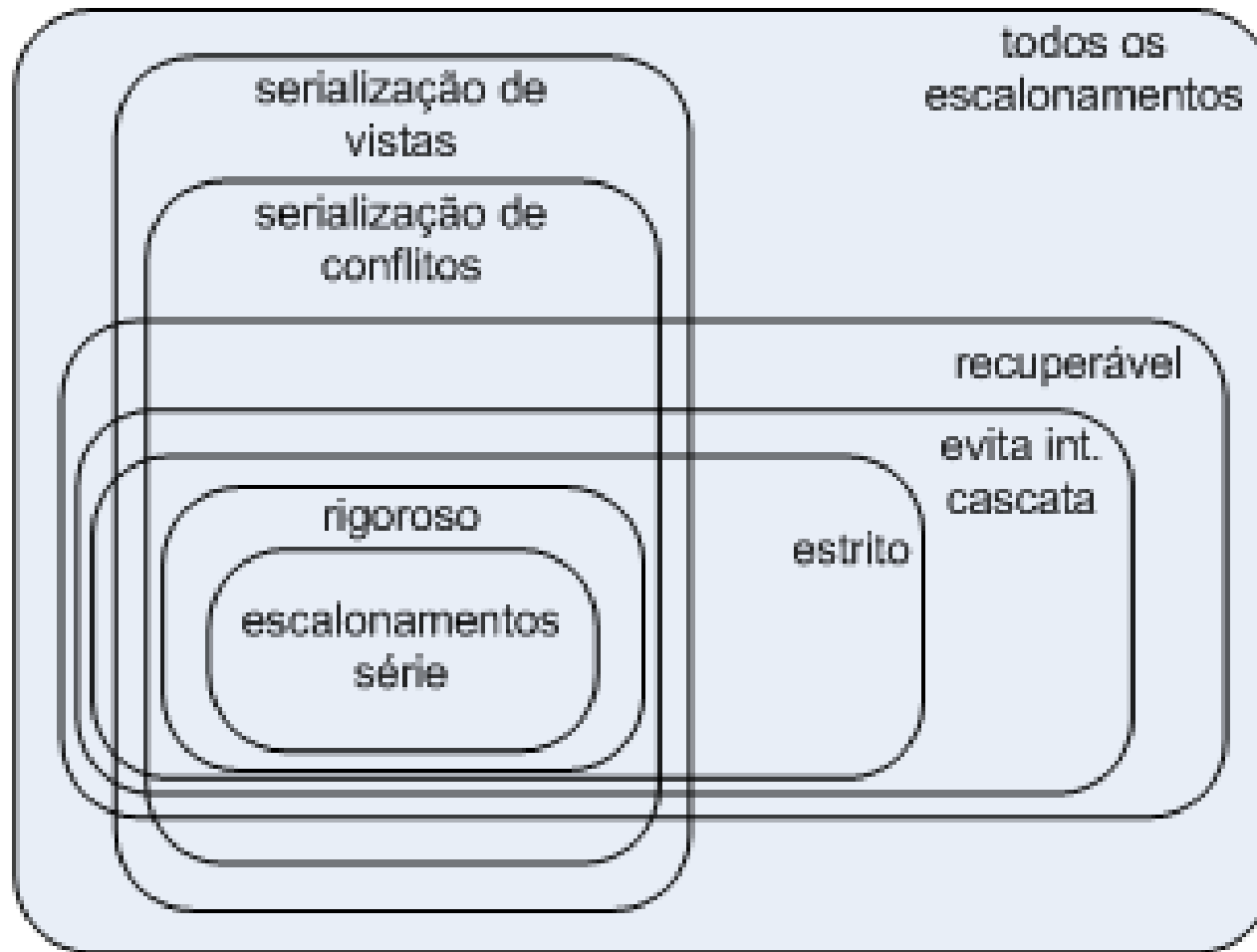
Escalonamento estrito

- ❑ Um escalonamento **estrito** adia todas as escritas e leituras de elementos não-confirmados (sujos) até que as outras transações que os escreveram terminem
- ❑ Um escalonamento estrito é recuperável e evita interrupções em cascata

Escalonamento rigoroso

- ❑ Um escalonamento é rigoroso se for estrito e se nenhuma transação escrever elementos lidos por outras transações até que estas terminem
 - ❑ Implica que a ordem de execução é igual à ordem de serialização
- ❑ rigoroso \subset estrito \subset evita cascata \subset recuperável

Classes de recuperação



Classes de recuperação

- ❑ A recuperação é ortogonal à serialização
- ❑ Um escalonamento deve-se sempre situar na intersecção das classes serializáveis (por exemplo a conflitos) e recuperáveis (por exemplo estrito)

Que tipos de falhas?

- ❑ Um SGBD usa diversos níveis de memória
 - ❑ Em qualquer instante, há dados nos diferentes níveis, de forma provisória ou permanente
- ❑ Em caso de falha, esses níveis são afetados de forma diferente
- ❑ Estudar as formas de recuperação implica conhecer esses níveis, e como o seu conteúdo resiste às falhas
- ❑ Qualquer sistema falha...
 - ❑ Ver MTTF: *Mean Time To Failure*

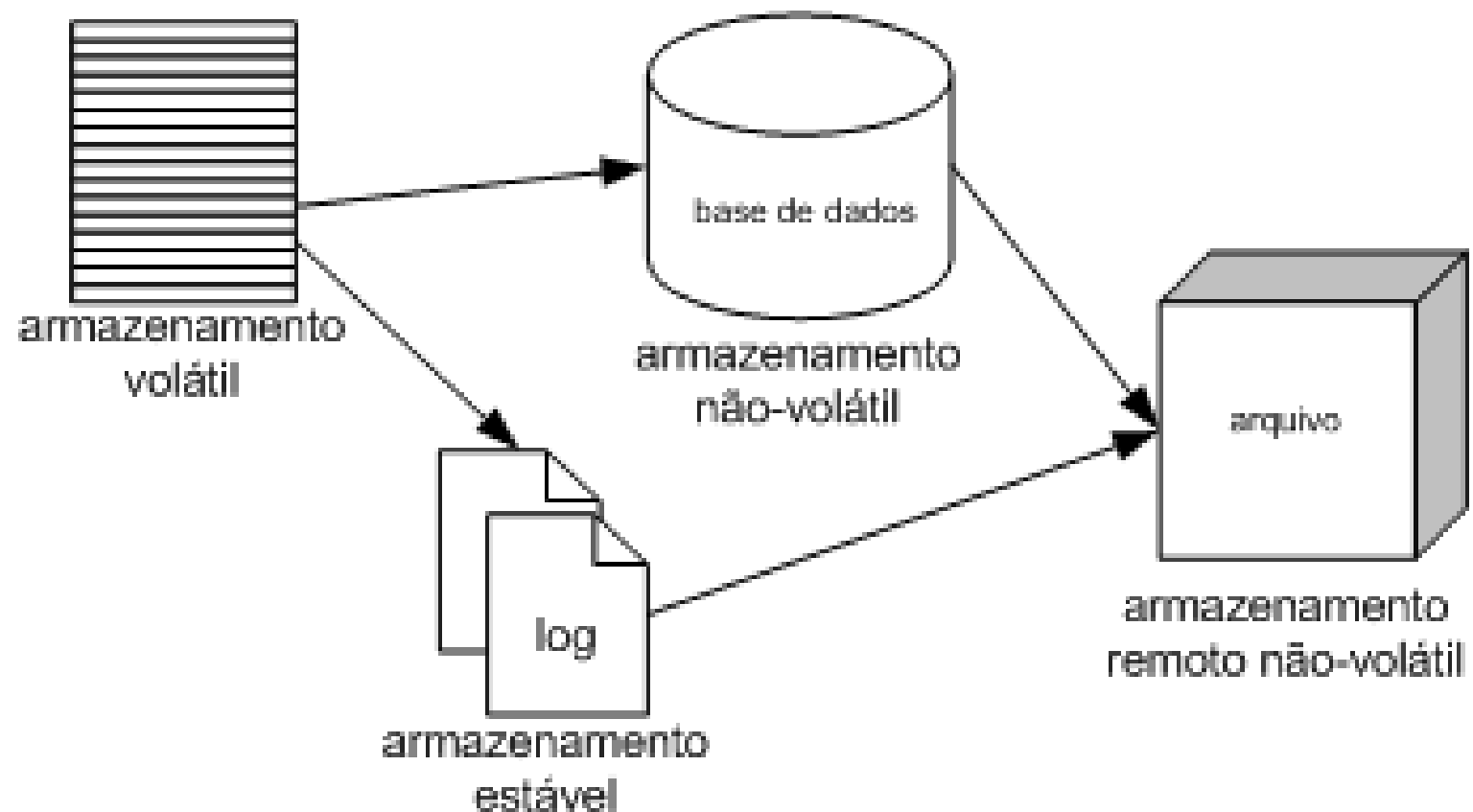
Falhas não-intencionais

- ❑ Para aumentar a fiabilidade do sistema tem que se aumentar o grau de redundância
- ❑ 100% de fiabilidade é impossível
 - ❑ Exemplo: discos de qualidade têm um MTTF de 1-1,5 Mhoras (em 1000 discos, em média um falha a cada 1500h)
- ❑ Uma opção consiste em manter um histórico de todas as modificações na base de dados (sob a forma de listas FAZ e DESFAZ)

Níveis de armazenamento

- ❑ Volátil: o seu conteúdo desaparece totalmente em caso de falha (exemplo: RAM, cache)
- ❑ Não-volátil: o seu conteúdo resiste a falhas sistema, mas pode ter problemas nos suportes, ou nos dispositivos de leitura (exemplo: discos magnéticos)
- ❑ Estável: é imune à maioria das falhas, devido a usar redundância (exemplo: RAID)
- ❑ Remoto não-volátil: arquivo em discos ou bandas

Níveis de armazenamento



Tipos de falhas

- ❑ **Nas Transações:** as transações podem ser interrompidas pelo Gestor de Transações, pelo utilizador (*rollback*), por erros de programação ou da base de dados
- ❑ **No Sistema:** podem ser devidas a falha de energia, erros de programação, erros no Sistema Operativo,...
- ❑ **Nos Suportes de armazenamento:** podem ser devidas a erros de leitura ou de escrita, a falhas nos suportes magnéticos, erros mecânicos,...

O que existe em cada falha

- ❑ Transação: para desfazer ou refazer uma transação pode-se usar o conteúdo da memória volátil e dos *logs* em memória estável.
- ❑ Sistema: só se pode usar o conteúdo da memória não-volátil e dos *logs* em memória estável.
- ❑ Suporte: só se pode usar o conteúdo da memória remota não-volátil (arquivo)

Organização da memória

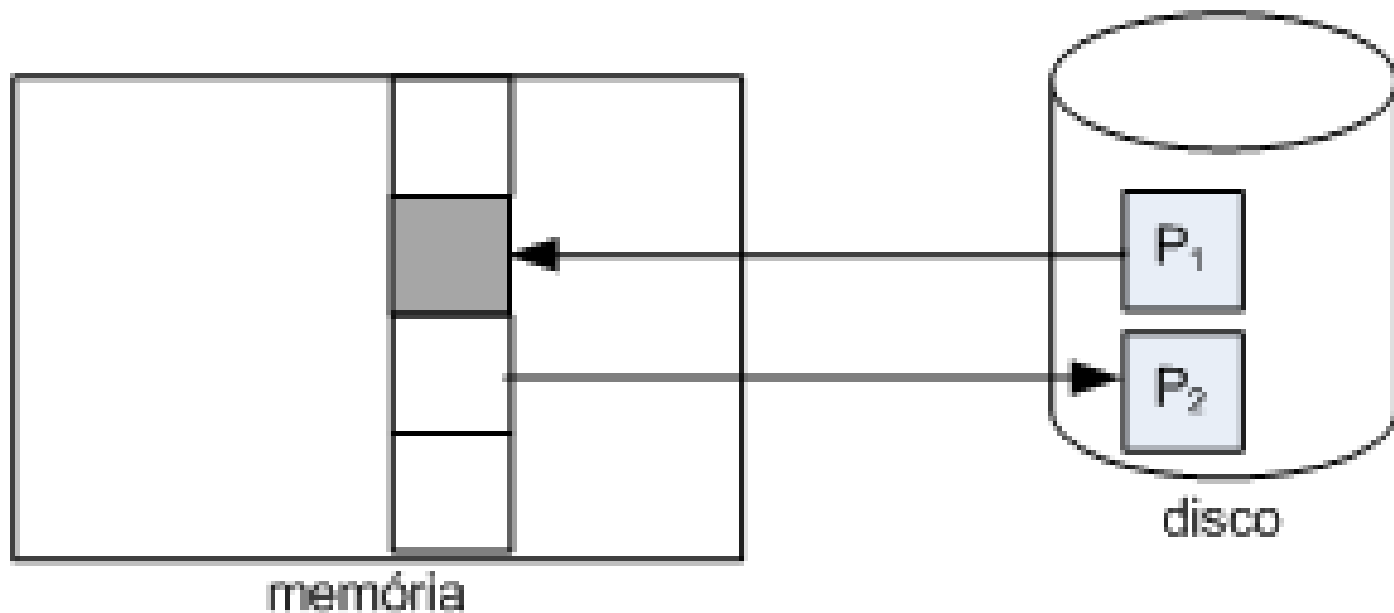
- ❑ O SGBD reserva uma parte da memória para trabalhar (um “tampão”)
- ❑ A memória principal está dividida em blocos de tamanho fixo (*frames*). Cada bloco pode conter geralmente uma página do disco
- ❑ A página é a unidade de transferência de e para o disco; o tamanho típico é 4k ou 8k.
- ❑ Quando o SGBD quer aceder a um elemento e ele não está em memória, tem que lê-lo do disco (tem de “paginar”)

Parametrização

- ❑ Maioria dos SGBD permite definir o tamanho das páginas
 - ❑ PostgreSQL: 8k (“Database block size”)
 - ❑ Faça: pg_controldata D:\PostgreSQL\data
- ❑ Mas se o tamanho for um múltiplo da página do S.O., as operações podem não ser “verdadeiramente” atômicas
 - ❑ As escritas para o disco não são atômicas

Gestor de Memória

Faz a paginação



Gestor de Memória

- ❑ Tem por objetivo:
 - ❑ Minimizar o número de páginas pedidas e que não se encontram em memória (faltas de páginas)
- ❑ Permite ao SGBD não utilizar a gestão de memória do S.O., e portanto ser mais eficaz (mas não deve duplicar trabalho!)

As frames

- ❑ Cada *frame* de memória tem 2 variáveis:
 - ❑ Pin Count: que indica quantas operações estão em curso sobre a página que contém
 - ❑ Dirty: que indica se a frame foi modificada desde que a página que contém foi lida em memória
- ❑ Guarda o `pageID`, e informação sobre trincos
- ❑ Estas variáveis são usadas na gestão da memória, e guardadas num BCB (Buffer Control Block)

Como usar as variáveis

- ❑ Quando uma transação pede um elemento numa dada página:
 - ❑ Se uma frame contém a página, incrementa o seu Pin Count
 - ❑ Se a página não está em memória
 - ❑ Escolhe uma frame com Pin Count = 0, incrementa o seu Pin Count
 - ❑ Lê a página para esta frame em memória
 - ❑ Devolve o endereço da frame que contém a página pedida

Como usar as variáveis

- ❑ Quando a operação sobre a página termina, a transação informa o Gestor de Memória para decrementar o Pin Count (operação *unfix*)
- ❑ Se a operação modificou a página, o Gestor de Memória coloca o bit Dirty a 1
- ❑ A operação deve reter a página o menor tempo possível

Escolha das frames

- ❑ Nalgumas situações podem não existir frames livres, ou com Pin Count a 0
 - ❑ Nesse caso, a transação fica em espera, e poderá ter que vir a ser interrompida
 - ❑ Razão para se libertar a página mal seja possível (decrementando o Pin Count)
- ❑ Nos casos em que há várias frames com Pin Count a 0, podem-se utilizar várias políticas de substituição
- ❑ Se a página escolhida foi modificada (Dirty=1), tem que ser escrita no disco primeiro

Políticas de substituição

- ❑ Tentam “prever” a sequência de acessos a uma página, escolhendo para substituir a que não vai ser usada tão cedo
- ❑ Seja $d_t(p)$ a distância direta de p (a distância à próxima referência de p)
- ❑ Os algoritmos usam as referências (distâncias) passadas para estimar as referências futuras
- ❑ Modelo Referências Independentes: assume-se que referências consecutivas à mesma página são independentes, ou seja a probabilidade de acontecerem é constante

Algumas políticas

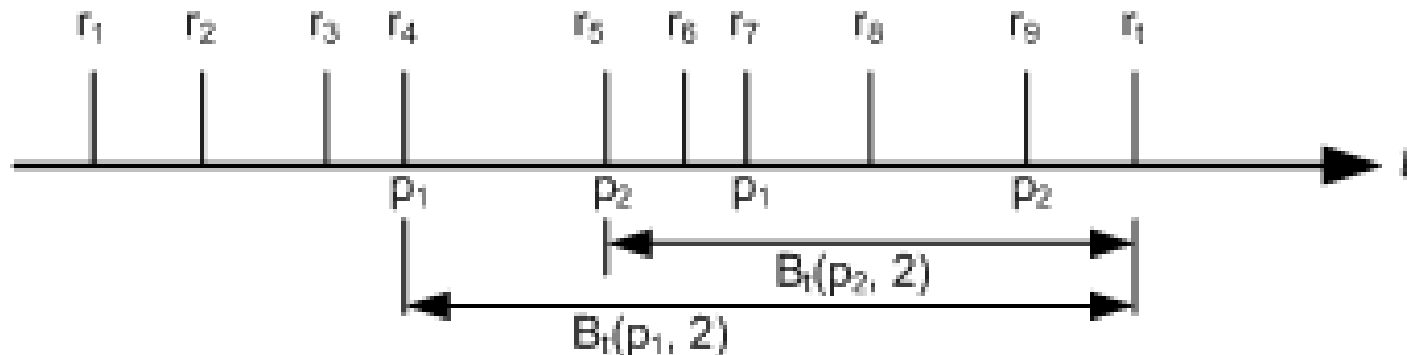
- ❑ Menos Recentemente Utilizada (**LRU**: Least Recently Used)
 - ❑ A lógica é que se não é usada há muito tempo é pouco provável que o venha a ser tão cedo
- ❑ Mais Recentemente Utilizada (**MRU**: Most Recently Used)
 - ❑ A lógica é que se foi usada agora, não vai voltar a sê-lo tão cedo
- ❑ Outros: FIFO, CLOCK, GCLOCK, LIRS, ARC, 2Q

Na prática

- ❑ Menos Recentemente Utilizada k-último acesso (**LRU-K**: *Least Recently Used k^{th} Last Request*), escolhe para substituir a página que tenha sido utilizada pela **k** última vez há mais tempo.
- ❑ A maioria dos produtos utiliza variantes de LRU-2
- ❑ MySQL usa LIRS, PostgreSQL usa CLOCK

LRU-2

- ❑ p_i páginas, r_i referências, B distância
- ❑ LRU-2 escolhe a página com maior B na altura da referência r_t : p_1
- ❑ Imune a varrimentos, substitui as páginas menos frequentes (segunda referência mais antiga)

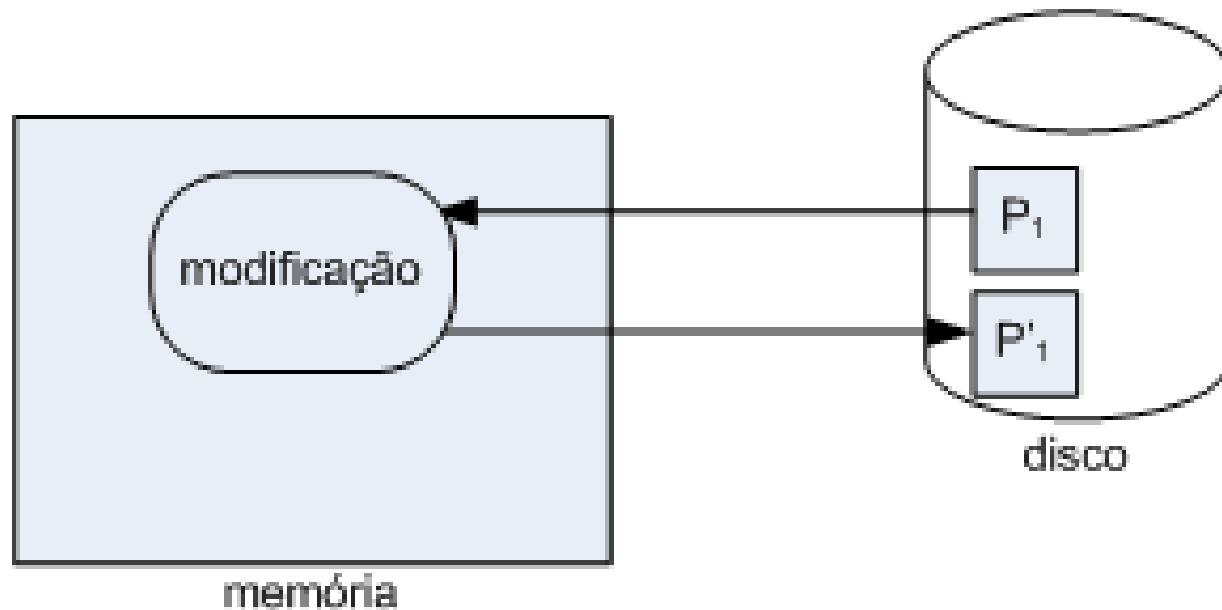


Modificação de elementos

- ❑ Duas formas de propagar as alterações para o disco
 - ❑ Escrever as páginas sempre para o seu espaço físico no disco, isto é, de onde foram lidas, reescrevendo assim o anterior conteúdo.
 - ❑ Escrever as páginas para uma nova localização, mantendo a página originalmente lida, que passa assim a ser uma “sombra” da nova página.

Páginas-sombra

- ❑ *Shadow-Page*: mantém-se a página atual e a página-sombra no disco. Um diretório aponta sempre para a página atual



Páginas-sombra (1)

- ❑ Apenas de interesse histórico (System R)
 - ❑ PostgreSQL mantém o histórico de modificações por linha, até ser desnecessário
 - ❑ GemStone usava “páginas sombra”
- ❑ As “páginas sombra” mantêm-se até ao ponto de controlo
- ❑ Recuperar consiste em repor as páginas “anteriores”
- ❑ Implica mais gestão das páginas e pontos de restauro demorados

Implicações na técnica de recuperação

- ❑ A primeira opção implica que tenha que se guardar um histórico (*log*), uma vez que os valores anteriores são reescritos
 - ❑ Técnica mais usada, e que vamos ver
- ❑ A segunda opção (“*shadow page*”) é o próprio histórico: as páginas anteriores mantêm-se no disco

Políticas do Gestor de Memória

- ❑ O método de recuperação depende da relação entre o Gestor de Recuperação e o Gestor de Memória
- ❑ Essa relação é caracterizada pelas opções:
 - ❑ FORCE / ~FORCE
 - ❑ STEAL / ~STEAL
- ❑ Decide essencialmente o que fazer:
 - ❑ Quando uma transação confirma
 - ❑ Quando é necessário mais memória

Confirmação de transações

- ❑ Depende se o Gestor de Memória torna permanentes alterações quando da confirmação da transação vinda do Gestor de Recuperação
- ❑ FORCE: as páginas modificadas por transações são escritas para o disco sempre que estas confirmam (atualização diferida)
- ❑ ~FORCE: as páginas modificadas por transações confirmadas são escritas para o disco sempre que for considerado adequado

Escrita de páginas para o disco

- ❑ Depende se o Gestor de Memória espera pela ordem do Gestor de Recuperação
- ❑ STEAL: as páginas modificadas por transações não-confirmadas podem ser escritas para o disco se for necessário reclamar o espaço que estas ocupam (ou otimizar tráfego)
- ❑ ~ STEAL: as páginas modificadas por transações não-confirmadas não podem ser escritas para o disco, sendo guardadas em memória pelo menos até ao fim da transação

Implicações em caso de falha

- ❑ Podem existir transações confirmadas sem os dados no disco, ou transações ativas com dados no disco
- ❑ O trabalho de recuperação é diferente consoante os casos

Utilizar apenas REFAZER

- ☐ Quando não é preciso “roubar” páginas
- ☐ Para transações curtas (a memória não é problema)
- ☐ A operação de confirmação é rápida
- ☐ A operação de interrupção é rápida: só precisa de libertar as páginas presas

Utilizar apenas DESFAZER

- ☐ Quando é preciso “roubar” páginas, mas não é necessário “forçar” para o disco
- ☐ Para transações longas
- ☐ Pouca memória disponível

Combinações

□ As combinações possíveis são:

	~STEAL	STEAL
FORCE	trivial	
~FORCE		preferível

Opção mais usada

- ❑ A combinação ~FORCE/STEAL significa que o Gestor de Memória pode escrever no disco qualquer página modificada sempre que o decidir, independentemente da transação que a modificou já ter confirmado.
- ❑ Este comportamento é o desejável porque permite otimizar o tráfego entre a memória e o disco, e evita utilização de muita memória (por transações que acedem a muitos dados, ou muito longas)

~FORCE/STEAL

- ❑ Interrupção
 - ❑ Gestor de Recuperação **desfaz** efeitos
- ❑ Confirmação
 - ❑ Gestor de Recuperação escreve uma entrada de confirmação
- ❑ Recuperação
 - ❑ Transações confirmadas são recuperadas
 - ❑ Se só tiverem uma entrada BEGIN, são desfeitas

Opção mais simples

- ❑ Para recuperar é mais simples FORCE / ~STEAL
 - ❑ O disco só contém dados confirmados (não é necessário desfazer nada)
 - ❑ Todas as transações confirmadas já têm os dados no disco (não é necessário refazer nada)
- ❑ É no entanto ineficiente (FORCE), e inflexível na gestão da memória (~STEAL)
- ❑ Pode não ser possível interromper uma transação porque não há memória...

FORCE / ~STEAL

- ❑ Interrupção
 - ❑ Gestor de Memória liberta as páginas “presas”
- ❑ Confirmação (faz de forma atômica:)
 - ❑ Gestor de Recuperação envia um *flush* ao Gestor de Memória para as páginas sujas
 - ❑ Gestor de Memória liberta as páginas “presas”
 - ❑ Gestor de Recuperação escreve uma entrada de confirmação
- ❑ Recuperação
 - ❑ Nada a fazer se a granularidade for a página

Arranque da base de dados após falha

- ❑ Transações em curso, outras terminadas
- ❑ Usa-se o log para colocar a base de dados num estado consistente
- ❑ Opção REFAZER:
 - ❑ Percorre-se o log desde a entrada mais antiga para identificar transações confirmadas
 - ❑ Segunda fase refazem-se as suas operações
- ❑ Opção DESFAZER:
 - ❑ Percorre-se o log desde a entrada mais recente, desfaz operações de transações sem entrada de confirmação

Comparação

☐ REFAZER

- ☐ + escritas rápidas, operação de confirmação rápida (só usam o log)
- ☐ - Arranque demorado em caso de falha

☐ DESFAZER

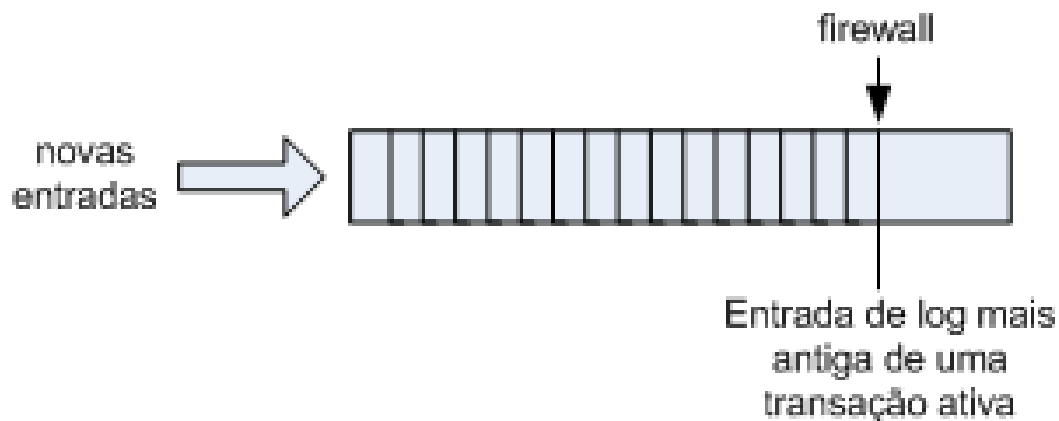
- ☐ + gere bem a memória, arranque rápido
- ☐ - implica forçar escritas na confirmação
 - ☐ Se várias transações modificam o mesmo elemento, este é escrito várias vezes....
 - ☐ Se muitas transações confirmam ao mesmo tempo, temos um estrangulamento...

Técnicas de log

- ❑ Um log nunca diminui, e nunca nada é apagado. Cada entrada no log tem um LSN (Log Sequence Number)
- ❑ Um log é uma lista FIFO
- ❑ Escrita sequencial
 - ❑ Na cauda do log
 - ❑ Sequencial evita posicionamento do braço (já lá está, mas rápido)
 - ❑ Disco dedicado (evita que o braço seja movido)
- ❑ Imagine $1000 \text{ TPS} \times 500 \text{ octetos} = 500 \text{ Ko} / \text{s}$

Gestão do log

- ❑ Transações longas ou TPS elevado podem utilizar todo o espaço de log
- ❑ O log é gerido como uma lista circular, reescrevendo entradas de transações terminadas: a firewall impede de reescrever entradas ativas

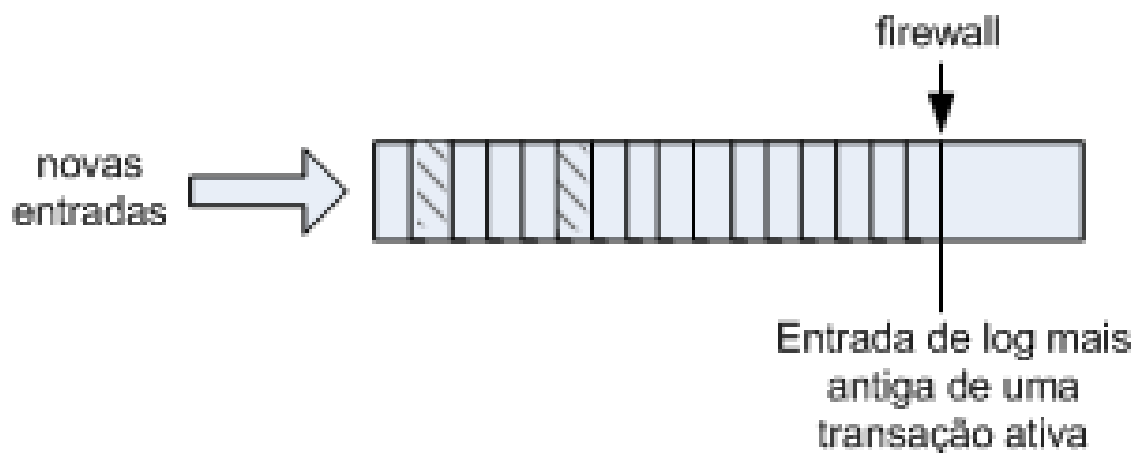


Gestão do log (1)

- ❑ Se for necessário mais espaço, o Gestor de Log tem duas opções:
 - ❑ Interrompe uma transação antiga (a evitar, e tem que ter espaço para escrever entradas UNDO, ou usar STEAL)
 - ❑ Inicia um ponto de controlo (preferível, mas a evitar porque é demorado)
- ❑ Outros autores propuseram deslocar a entrada de uma transação antiga para o início do log (permitindo assim libertar mais espaço de transações confirmadas na cabeça do log)

Gestão do log (2)

- ❑ Outras soluções optam por comprimir o log
- ❑ Numa situação desfavorável (mas normal), o log tem “buracos” (entradas não necessárias) que não consegue aproveitar:
- ❑ Enquanto a firewall não avançar, as entradas são perdidas



Gestão do log: entradas

- ❑ Dois tipos para representar modificações:
 - ❑ Entrada física: regista-se a localização do tuplo na página, e os valores antes e depois. Simples para recuperar (usadas para REFAZER)
 - ❑ Entrada lógica: representam a operação. São mais simples de criar, mas mais complexas para recuperar (usadas para DESFAZER)
- ❑ Mistura dos dois: entrada **fisiológica**
 - ❑ Refere-se apenas a uma página
 - ❑ Representa operações lógicas dentro dessa página

Pontos de Controlo

- ❑ Evitam ter que percorrer o log todo em caso de falha
 - ❑ A partir de uma dada entrada garante-se que os dados já estão em memória estável
- ❑ Escrevem uma entrada BEGIN CHECKPOINT + Transações ativas neste ponto
- ❑ Escrevem as páginas para memória estável
- ❑ Escrevem uma entrada END CHECKPOINT
- ❑ Pode ou não aceitar novas transações enquanto faz o ponto de controlo: “transaction-consistent” (não aceita) ou “fuzzy” (aceita)

Pontos de Controlo (1)

- ❑ No método firewall de gestão do log, a firewall é a entrada mais antiga entre a entrada do início do ponto de controlo mais recente e a mais antiga das transações ativas

Pontos de Controlo Difusos (fuzzy)

- ☐ Não aceita novas modificações
- ☐ Não escreve todas as páginas no disco
- ☐ Faz duas listas:
 1. Páginas sujas em memória
 2. Transações ativas e entrada de log mais recente
- ☐ Aceita novas modificações exceto na lista 1
- ☐ Cria uma entrada no log com a lista 2
- ☐ Retoma processamento normal

Pontos de Controlo Difusos (1)

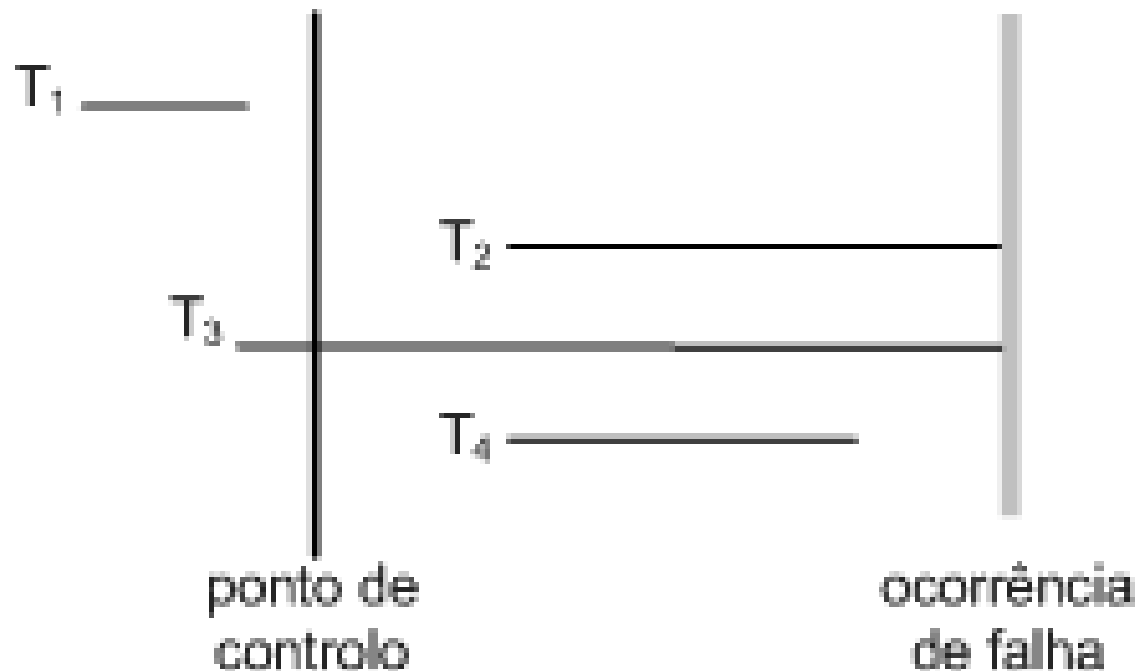
- ❑ Faz a escrita das páginas sujas quando pode
- ❑ Não faz mais nenhum ponto de controlo sem terminar de escrever as páginas sujas na altura do último
- ❑ A recuperação só precisa de ir até ao penúltimo ponto de controlo (representa um estado consistente)
- ❑ É um método rápido, não implica a paragem de transações durante toda a sua duração

Recuperação

- ❑ REFAZ = {}, DESFAZ = {}
- ❑ Percorre o log até ao Ponto de Controlo mais recente. Se encontra:
 - ❑ $\langle T_i, \text{commit} \rangle$ junta T_i a REFAZ
 - ❑ $\langle T_i, \text{start} \rangle$ e $T_i \notin \text{REFAZ}$, junta T_i a DESFAZ
- ❑ Para cada $T_i \in \text{ATIVA}$
 - ❑ Se $T_i \notin \text{REFAZ}$, junta T_i a DESFAZ
- ❑ Percorre o log desde o início até encontrar $\langle T_i, \text{start} \rangle$ para cada $T_i \in \text{DESFAZ}$, desfazendo
- ❑ Desde o ponto de controlo mais recente até ao início do log, refaz as entradas de $T_i \in \text{REFAZ}$

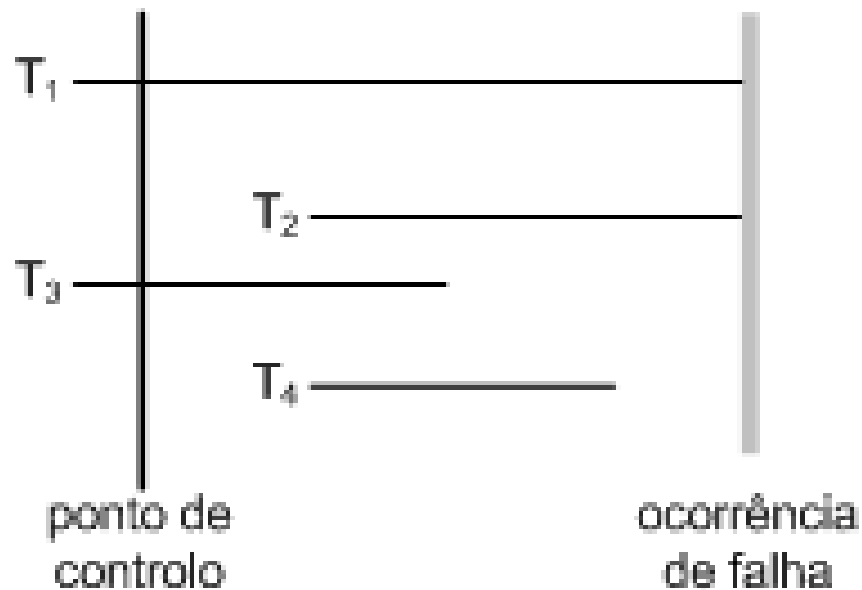
Exemplo

- ❑ Identifique os passos da recuperação neste caso:



Exemplo (1)

- ❑ T_1 e T_2 : desfazer
- ❑ T_3 e T_4 : refazer



Cenários

- ❑ t_1 T escreve página P
- ❑ t_2 Gestor de Memória decide enviar P para o disco
- ❑ t_3 Falha !!!
- ❑ Não se pode desfazer a ação de T em P porque o log também foi perdido!!
- ❑ É necessário garantir que o log está a salvo

Técnicas de log: WAL

- ❑ Regra WAL (Write-Ahead Logging):
 - ❑ Escrita de página: tem que existir uma entrada no *log* em memória estável antes das modificações serem escritas no disco
 - ❑ Confirmação: antes de uma transação confirmar, as suas modificações têm que estar nos *logs* em memória estável
 - ❑ Uma transação só está **confirmada** quando as suas entradas de log estiverem em memória estável
- ❑ Idempotência: pode-se recomeçar após uma falha tantas vezes quantas as que se quiser

WAL

- ❑ Ou seja:
 - ❑ A informação necessária para desfazer uma operação deve ser registada no log em memória estável **antes** da atualização ser aplicada na cópia não-volátil da base de dados
 - ❑ A informação necessária para refazer uma operação deve ser registada em memória estável **antes** de se escrever a confirmação da transação no log
- ❑ Permite ~FORCE / STEAL

Mas...

- ❑ Dependendo dos discos usados, as escritas podem ser feitas apenas para as suas caches
- ❑ A durabilidade fica comprometida....
- ❑ Necessário ter baterias para os discos (mas não salvaguarda outro tipo de falhas)
- ❑ Alguns SGBD tentam forçar a escrita nos discos, evitando as suas caches

WAL em PostgreSQL

- Escrito para \$PGDATA/pg_xlog
- A localização pode ser mudada com um atalho
- Segmentos de 16MB

Trincos e fechos

- ❑ Os trincos são usados nas páginas durante processamento normal e de desfazer
 - ❑ Trincos de leitura e de escrita
 - ❑ Garantem consistência física das páginas
- ❑ Um trinco é semelhante a um semáforo
- ❑ As esperas circulares são evitadas da seguinte forma:
 - ❑ Uma transação nunca detém mais do que 2 trincos simultaneamente

Cópias de salvaguarda

- ❑ Adicionalmente deve-se copiar integralmente a base de dados
- ❑ Em PostgreSQL: pg_dump
- ❑ Nenhum modificação pode estar em curso durante um “dump”
- ❑ Ficheiro resultante deve ir para arquivo (diferente local, múltiplas cópias)

Técnicas de Recuperação

- ❑ Várias possíveis
- ❑ ARIES (IBM, 1992): **A**lgorithms for **R**ecovery and **I**solation **E**xploiting **S**emantics
 - ❑ Baseado em WAL
 - ❑ Suporta as estratégias ~FORCE / STEAL
 - ❑ Elemento básico de recuperação é a página
 - ❑ Cada entrada no log tem um LSN (Log Sequence Number)

ARIES: granularidade

- ❑ Registo de modificações nas páginas, para a operação de REFAZER (log físico)
- ❑ Registo de modificações em cada linha, para a operação de DESFAZER (log lógico)
- ❑ Log ***fisiológico***
- ❑ Porquê?
 - ❑ Refazer tem que repor páginas (id físico) porque não se sabe em que estado está a BD
 - ❑ Desfazer deve ser registo a registo, ARIES só desfaz as transações perdedoras (não confirmadas na altura da falha)

Log fisiológico

- ❑ Reduz o tamanho do log
- ❑ Exemplo: uma inserção obriga a deslocação de registos dentro da página
 - ❑ Com log físico, grande parte da página tinha que ficar no log
 - ❑ Com log lógico, só a inserção fica, o que permite REFAZER a operação

ARIES: entrada de log

- ❑ Vários tipos de entradas:
 - ❑ Commit
 - ❑ Rollback
 - ❑ BEGIN/END Checkpoint
 - ❑ Update/Insert/Delete
 - ❑ Start (opcional)
 - ❑ End
 - ❑ CLR

ARIES: entrada de log

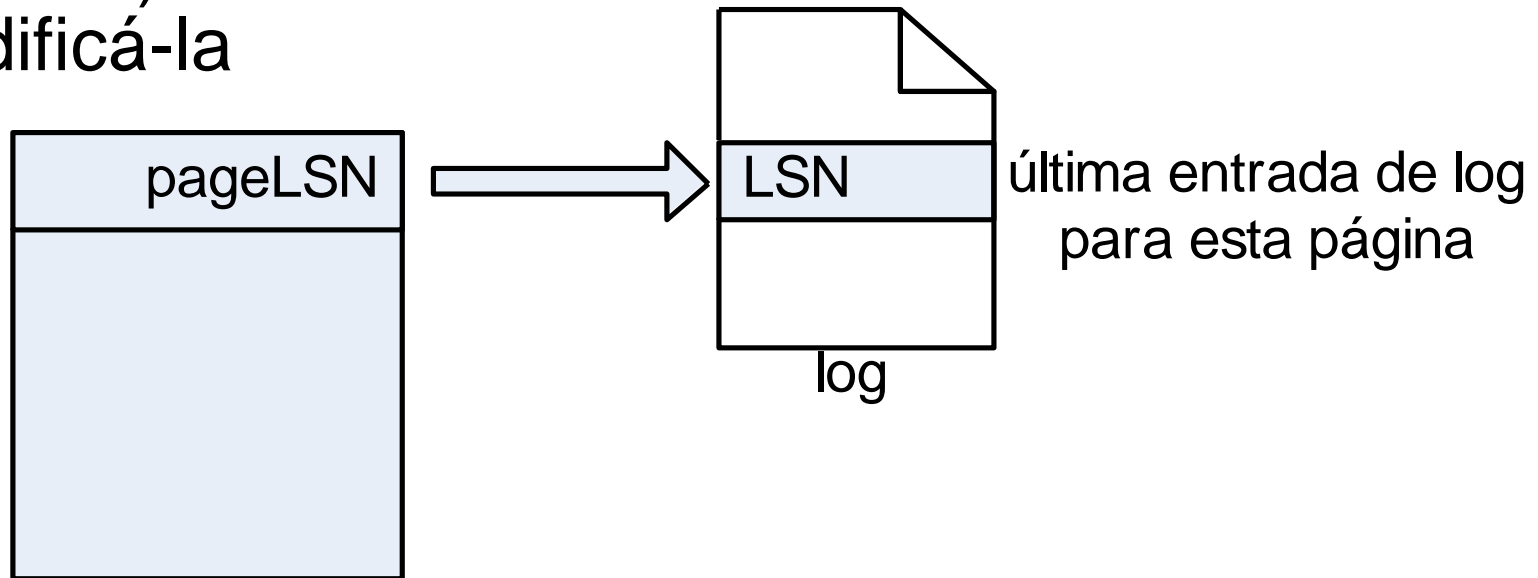
- ❑ LSN (Log Sequence Number)
- ❑ ID do registo modificado / Página
- ❑ Antigo valor (usado para desfazer)
- ❑ Novo valor (usado para refazer)
- ❑ ID da transação
- ❑ Tipo de operação (insert, delete, update,...)
- ❑ Tamanho da entrada
- ❑ Entrada anterior criada por esta transação (prevLSN)

ARIES: entrada de log

- ❑ Compensation Log Record (CLR)
 - ❑ Permite registrar o desfazer de uma ação
 - ❑ Tanto numa interrupção normal como na fase de Desfazer
- ❑ O LSN do CLR vai para pageLSN
- ❑ Um CLR nunca é desfeito, mas pode ser refeito
- ❑ Tem um undoNxtLSN que é o prevLSN da entrada que desfaz

ARIES

- ❑ Cada página tem um pageLSN (*Log Sequence Number*): LSN da entrada mais recente a modificá-la



- ❑ O pageLSN evita repetir operações múltiplas vezes em caso de falha consecutiva (idempotência)

ARIES: estruturas

- ❑ Tabela de transações: tem uma entrada para cada transação activa, guardando também o LSN da última entrada de log gerada por esta transação (lastLSN)
 - ❑ Mantida pelo Gestor de Transações
- ❑ Tabela de Páginas Sujas (*Dirty Page Table*, ou DPT), tem entradas para todas as páginas sujas residentes em memória guardando o LSN da primeira entrada que as sujou (recLSN)
 - ❑ Mantida pelo Gestor de Memória

Tabela de Transações

- ❑ Mantida pelo Gestor de Transações
- ❑ Tem uma entrada para cada transação ativa, guardando também o LSN da última entrada de log gerada por esta transação (lastLSN)
- ❑ É atualizada em modo normal, contendo o estado da transação (ativa, confirmada, interrompida)
- ❑ Tem uma entrada undoNextLSN usada apenas durante a fase de Desfazer (ou interrupção normal)

Tabela de Páginas Sujas

- ❑ Mantida pelo Gestor de Memória
- ❑ A DPT (*Dirty Page Table*), tem entradas para todas as páginas sujas residentes em memória. Cada entrada tem um recLSN (recoveryLSN), que é a primeira entrada de log que a sujou
- ❑ Em modo normal, regista-se uma página apenas quando é trazida para memória com intenção de modificar
- ❑ Quando uma página é escrita para o disco, remove-se da DPT

Tabela de Páginas Sujas (1)

- ❑ Em funcionamento normal:
 - ❑ Quando uma página é modificada, guarda-se em DPT, e em recLSN o LSN da entrada a ser gerada
 - ❑ Se a página já estiver na DPT não faz nada
 - ❑ Assim, recLSN é o primeiro LSN cuja modificação não está em disco
- ❑ Durante a recuperação:
 - ❑ É atualizada sempre que uma entrada de modificação é encontrada na fase de Análise

Estratégias possíveis

- ❑ A operação de ARIES é independente da estratégia de confirmação (FORCE) e da de substituição (STEAL)
- ❑ A única restrição é usar WAL

O que faz uma transação

- ☐ Pede uma página, marca-a como “pinned”
- ☐ Pede um trinco exclusivo
- ☐ Escreve a entrada LSN para o log
- ☐ Modifica a página
- ☐ Atualiza o lastLSN na tabela de transações
- ☐ Atualiza a DTP
- ☐ Atualiza o pageLSN da página
- ☐ Liberta o trinco
- ☐ “Desprende” a página (“unpin”)

Quando se escreve para o disco

- ❑ flushedLSN: é o maior LSN já escrito em disco
- ❑ As páginas de log são escritas
 - ❑ Nos pontos de controlo
 - ❑ Quando uma transação confirma
 - ❑ Quando se quer escrever uma página com $\text{pageLSN} > \text{flushedLSN}$
- ❑ As páginas da memória são escritas
 - ❑ Dependendo da política do Gestor de Memória
 - ❑ Pede primeiro um trinco partilhado na página (assegura consistência)

As estruturas

❑ DPT, Tabela de Transações e Log

pageID	recLSN
100	10
150	20

transID	lastLSN
1	20
2	30

LSN	transID	type	pageID	before	after	prevLSN
10	1	update	100	xyz	ab	0
20	1	update	150	kwf	ik	10
30	2	update	100	ab	qwe	0

Os Pontos de Controlo

- ❑ Guardam a Tabela de Transações e a DPT
- ❑ O LSN desta entrada é escrito para memória estável quando o ponto de controlo termina
- ❑ São rápidos e não obrigam a paragem das transações
 - ❑ As páginas sujas vão sendo escritas para o disco em tarefa de fundo
 - ❑ As tabelas escritas só são válidas à altura do início do ponto de controlo
- ❑ Devem ser feitos periodicamente

Interrupção de uma transação (total ou parcial)

- ❑ Obtém o lastLSN da Tabela de Transações
- ❑ Escreve uma entrada ABORT
- ❑ Desfaz os seus efeitos até START ou saveLSN
- ❑ Antes de desfazer, escreve um CLR
 - ❑ undoNextLSN dessa entrada é o LSN seguinte a desfazer (ou seja, o prevLSN da entrada que se está a desfazer)
- ❑ A próxima entrada a desfazer é undoNextLSN se a entrada é um CLR ou prevLSN senão
- ❑ Quando termina escreve uma entrada END

ARIES: fases

- ❑ ARIES tem 3 fases:
 - ❑ Análise
 - ❑ Refazer (REDO)
 - ❑ Desfazer (UNDO)

Fase de Análise

- ❑ destina-se a recolher informação sobre páginas sujas e transações perdedoras (isto é, não confirmadas à altura da falha)
- ❑ Percorre o log desde o ponto de controlo mais recente (cuja entrada estava em memória estável), lê as tabelas (Transações e DPT) e atualiza-as à medida que prossegue
- ❑ Se encontrar um FIM (INÍCIO ou outra entrada) de transação remove-a (inclui-a) da tabela de transações
- ❑ Para cada modificação verifica se a página está na DPT, senão junta-a e regista o recLSN como sendo o da entrada atual

Fim da Fase de Análise

- ❑ Para outras entradas, atualiza o lastLSN
- ❑ No fim, a tabela de transações contém as transações perdedoras (têm que ser desfeitas)
- ❑ A DPT contém uma estimativa conservadora das páginas provavelmente sujas (porque algumas poderão ter sido escritas para o disco)
- ❑ O valor de recLSN mais antigo na DPT é o ponto de início da fase REDO (redoLSN, ou firstLSN)
- ❑ Se DPT vazia, redoLSN \leftarrow checkpointLSN

Fase de Refazer

- ❑ Se a DPT estiver vazia no fim da análise, a fase de REDO é desnecessária
- ❑ Refaz todas as transações (ganhadoras e perdedoras), e refaz também CLR's
- ❑ Percorre o log da entrada mais antiga (redoLSN) para a mais recente
- ❑ O objetivo desta fase é “**repetir a história**”, para garantir que **todas** as operações no log (incluindo UNDO e ABORT) são aplicadas antes da fase seguinte (fase de “Desfazer”)
- ❑ Não escreve entradas de log (ver à frente)

Fase de Refazer: otimização

- ❑ Se a página não estiver na DPT, ignora
- ❑ Refaz apenas as modificações de uma página na DPT se $\text{recLSN} \leq \text{LSN}$ da entrada em curso e se $\text{pageLSN} < \text{LSN}$: ou seja, a entrada de log que modificou a página (pageLSN) é anterior (mais antiga) que a entrada em processamento (nota: se $\text{recLSN} > \text{LSN}$ nem precisa ler do disco a página para testar o pageLSN)

Fase de Refazer: resultado

- ❑ Para um REDO atualiza o pageLSN para o LSN atual mas não cria uma entrada no log
 - ❑ O pageLSN garante idempotência
- ❑ No fim desta fase a base de dados fica no estado em que estava quando a falha ocorreu
- ❑ Escreve entradas END para as transações que confirmaram (isto é que tinham um COMMIT)
- ❑ Remove da tabela de transações as que completaram a confirmação no ponto anterior

Fase de Desfazer

- ❑ A tarefa principal desta fase é a de garantir a atomicidade das transações, removendo incondicionalmente da base de dados **todos** os efeitos de transações não confirmadas.
- ❑ A DPT não é usada, apenas a TT (que apenas contém perdedoras, com o seu lastLSN)
- ❑ Percorre o log da entrada mais recente para a mais antiga que for necessário

Fase de Desfazer (1)

- ❑ Para cada transação na Tabela de Transações, vai utilizar o undoNxtLSN encontrado na fase de Análise
- ❑ Escolhe o $\max(T_i.\text{undoNxtLSN})$
 - ❑ Desfaz a entrada
 - ❑ Atualiza $T_i.\text{undoNxtLSN}$
 - ❑ Se for uma entrada normal, coloca $T_i.\text{undoNxtLSN} = \text{log.prevLSN}$
 - ❑ Se for um CLR, coloca $T_i.\text{undoNxtLSN} = \text{log.undoNxtLSN}$
 - ❑ Se $T_i.\text{undoNxtLSN}$ é nulo termina a transação (escreve END), senão coloca-a na lista para processamento

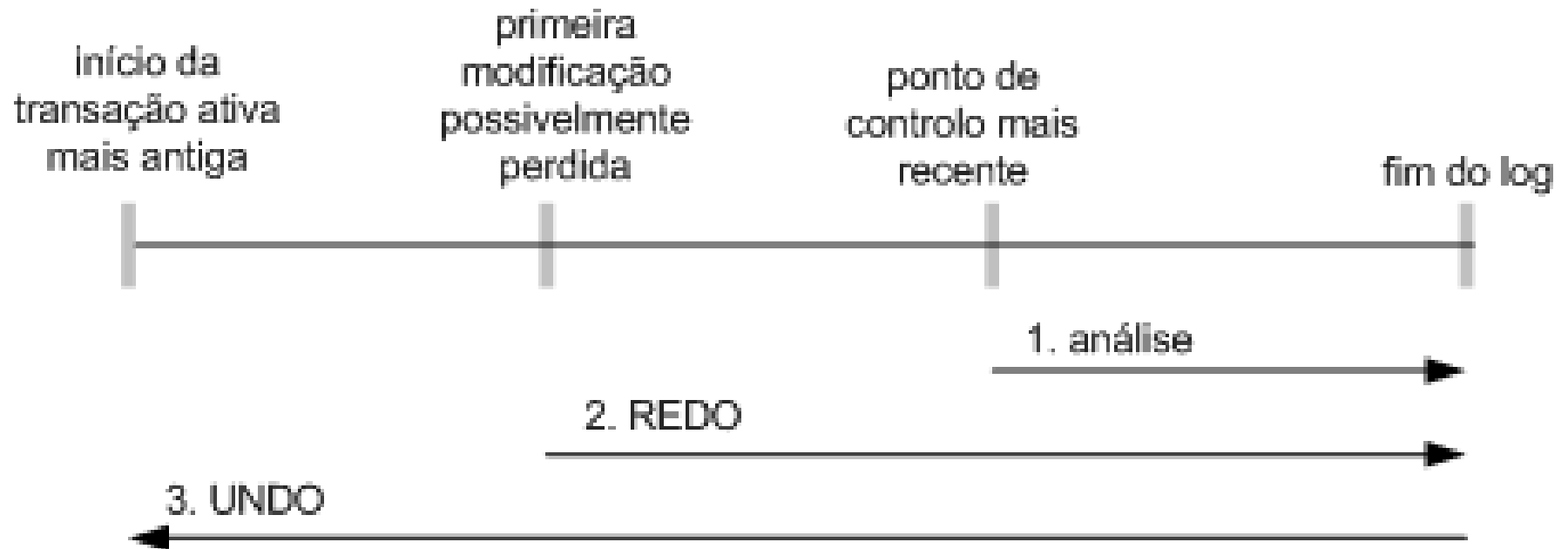
Fase de Desfazer (2)

- ❑ Escreve uma entrada de compensação (Compensation Log Record, CLR) para cada UNDO
 - ❑ Em caso de nova falha, evita desfazer a operação de desfazer!
 - ❑ Coloca em undoNxtLSN o prevLSN desta entrada
- ❑ Desfaz a modificação (coloca valor anterior)
- ❑ Atualiza o pageLSN para este LSN

Fase de Desfazer (3)

- ❑ Se a entrada é um CLR:
 - ❑ Se undoNxtLSN é nulo, termina a transação
 - ❑ Senão coloca undoNxtLSN na fila para processamento
- ❑ Se estamos a processar uma entrada destas significa que se estava a recuperar quando outra falha ocorreu
- ❑ Ao refazê-lo na fase II já se desfez o seu efeito
- ❑ Não se **desfaz** o que um CLR **desfaz**!

Funcionamento



Exemplo

10 BEGIN CHCK

20 END CHCK

30 T1 START

40 T1, P5, UPDATE

50 T2 START

60 T2, P3, UPDATE

70 T1 ABORT prevLSN = 40 ($= T_1.\text{lastLSN}$)

80 CLR UNDO LSN 40 undoNextLSN = 30

90 T1 END prevLSN = 80

P5 recLSN=40 pageLSN=40

P3 recLSN=60 pageLSN=60

Exemplo (1)

100 T3 START
110 T3, P1, UPDATE
120 T2, P5, UPDATE prevLSN = 60

P1 recLSN=110 pageLSN=110
P5 recLSN=40 pageLSN=120

☹ FALHA !!!!!

❑ Análise: reconstruir TT e DPT a partir do ponto de controlo (LSN 10). Resulta em:

$TT = \{(T_2, 120) (T_3, 110)\} (T_i, lastLSN)$

T_1 não entra, tem uma entrada END

$DPT = \{(P1, 110) (P3, 60) (P5, 40)\} (P_i, recLSN)$

$redoLSN = \min(recLSN) = 40$

Exemplo (2)

❑ Refazer: começar em redoLSN e refaz tudo

REDO 40 refaz, pq $p5.recLSN = LSN(40)$
 $p5.pageLSN \leftarrow 40$

REDO 50 passa

REDO 60 refaz, pq $p3.recLSN = LSN(60)$
 $p3.pageLSN \leftarrow 60$

REDO 70 passa

REDO 80 refaz pq $p5.recLSN(40) < LSN(80)$ e
 $p5.pageLSN(40) < LSN(80)$; $p5.pageLSN \leftarrow 80$

Exemplo (3)

REDO 90 *passa*

REDO 100 *passa*

REDO 110 *refaz, pq* $p1.recLSN = 110$,
 $p1.pageLSN \leftarrow 110$

REDO 120 *refaz pq* $p5.recLSN (40) < LSN (120)$ e
 $p5.pageLSN (80) < LSN (120)$); $p5.pageLSN \leftarrow 120$

Exemplo (4)

- ❑ Desfazer: inicializar o undoLSN de cada transação perdutora a lastLSN

- ❑ Enquanto $TT \neq \emptyset$

- ❑ Escolher o $\max(\text{undoLSN}) = 120$

130 CLR UNDO 120, $\text{undoNextLSN} \leftarrow 60$,
 $T_2.\text{undoLSN} \leftarrow 60$

140 CLR UNDO 110, $\text{undoNextLSN} \leftarrow 100$,
 $T_3.\text{undoLSN} \leftarrow 100$

150 CLR UNDO 100, remove T_3

Exemplo (5)

160 CLR UNDO 60, undoNextLSN \leftarrow 50,
 T_2 .undoLSN \leftarrow 50
170 CLR UNDO 50, remove T_2

Outro exemplo

```
10 T1, P3, UPDATE
20 T2, P2, UPDATE
30 T1 END
40 BEGIN CHECK
50 END CHCK
60 T3, P1, UPDATE
70 T2, P3, UPDATE
80 T2 END
☹ FALHA !!!!!
```

Outro exemplo (1)

□ Análise: reconstruir TT e DPT a partir do ponto de controlo (LSN 40). Resulta em:

$TT = \{(T_2, 20)\} (T_i, lastLSN)$

T_1 não entra, tem uma entrada END

$DPT = \{(P2, 20)\} (P_i, recLSN)$

P3 foi escrita no Ponto de Controlo com
pageLSN=10; e $P2.pageLSN = 20$

$redoLSN = \min(recLSN) = 20$

Outro exemplo (2)

- ❑ Refazer: começar em redoLSN e refaz tudo
- REDO 20 *passa*, pq $p2.recLSN = LSN(20)$
 $p2.pageLSN \geq LSN(20)$
- REDO 60 *refaz*, $p1.pageLSN \leftarrow 60$
- REDO 70 *refaz*, $p3.pageLSN \leftarrow 70$

Outro exemplo (3)

- ❑ Desfazer: inicializar o undoLSN de cada transação perdutora a lastLSN
 - ❑ Enquanto $TT = \{T_3\} \neq \emptyset$
 - ❑ Escolher o $\max(\text{undoLSN}) = 60$
- 90 CLR UNDO 60 *p1.pageLSN* \leftarrow 90