RELATÓRIO: 2º MINI-PROJETO

ADMINISTRAÇÃO E OPTIMIZAÇÃO DE BASES DE DADOS

70502: João Carlos Duarte Santos Oliveira Violante – METI **70599:** João Miguel Cordeiro Monteiro – METI **70627:** Pedro Luís Galvão Raminhas – MEIC	**Grupo 2**

1. Transaction Isolation Levels

Nota: Para as respostas às alíneas seguintes, admitiu-se que os procedimentos que podem estar a ser executados são apenas um dos três apresentados, e que apenas é executado um procedimento de cada tipo em cada momento.

1.1. Tendo em conta os procedimentos apresentados, um cenário de *dirty read* pode acontecer quando: o procedimento **insert_cheese** realiza uma escrita, utilizando a tabela CHEESE (escrita de um novo queijo, por exemplo). Caso não seja efetuado um commit, e o procedimento **update_production** de seguida efetue uma leitura sobre a tabela CHEESE, poderá acontecer então um problema de *reading uncommited data*.

1.2. Um cenário de *unrepeatable reads* não poderá acontecer tendo em conta os procedimentos apresentados, uma vez que não existe nenhum procedimento que efetue uma leitura sobre uma determinada tabela e depois efetue uma escrita sobre a mesma. Desta forma, mesmo que seja efetuada uma leitura sobre a tabela REGION ou sobre a tabela CHEESE (através do procedimento **update_production**), não existe nenhum procedimento que possa, de seguida, ler e alterar valores sobre essas tabelas.

1.3. Um cenário de *overwritting uncommited data* não poderá acontecer tendo em conta os procedimentos apresentados, uma vez que não existem dois procedimentos que efetuem a escrita nas mesmas tabelas.

1.4. a) **insert_cheese:** O nível de isolamento para executar este procedimento seria **read uncommited**. Como justificado nas alíneas 1.2 e 1.3, não podem acontecer cenários de *nonrepeatable reads* e de *overwritting uncommited data* e, por isso, não é necessário nenhum mecanismo para precaver os mesmos. Como este procedimento também não efetua nenhuma leitura sobre nenhuma tabela, também não poderá acontecer o cenário de *dirty read*. Por isso, utiliza-se o nível de isolamento menos restritivo.

b) **update_production:** O nível de isolamento para executar este procedimento seria **read commited**, pois este é um nível que permite cenários de *nonrepeatable read* e de *overwritting uncommited data*, mas impede cenários de *dirty read*. É importante evitar cenários de *dirty read* pois este procedimento executa leituras sobre as tabelas REGION e CHEESE e, como tal, se as escritas efetuadas por outros procedimentos nestas tabelas não efetuarem commit, serão lidos valores *dirty*. Os cenários de *nonrepeatable reads* e *overwritting uncommited data,* como justificado nas alíneas

1.2 e 1.3, não podem acontecer e, como tal, não é preciso nenhum mecanismo para precaver os mesmos.

c) **delete_region:** O nível de isolamento para executar este procedimento seria **read uncommited.** Como justificado nas alíneas 1.2 e 1.3, não podem acontecer cenários de *nonrepeatable reads* e de *overwritting uncommited data* e, por isso, não é necessário nenhum mecanismo para precaver os mesmos. Como este procedimento também não efetua nenhuma leitura sobre nenhuma tabela, também não poderá acontecer o cenário de *dirty read*. Por isso, utiliza-se o nível de isolamento menos restritivo.

2. Concurrency Control

2.1. a) O escalonamento apresentado não seria permitido em Strict 2-Phase Locking, uma vez que a transação T1 efetua uma escrita de A antes da transação T2 efetuar também ela uma escrita sobre A. Assim, para que não houvesse conflitos, T1 teria que fazer lock(A) antes de fazer write(A) e fazer unlock(A) depois disso. Como T1 faz uma escrita de B depois de T2 fazer uma escrita de A, isto significa que T1 teria que fazer lock(B) antes de fazer write(B) e fazer unlock(B) depois disso. Concluindo, como T1 fazia unlock(A) depois de fazer write(A) e antes de fazer lock(B), os unlocks dessa transação não seriam todos efetuados no final da mesma e, como tal, não cumpririam os requisitos de Strict 2-Phase Locking.

b) Admitindo que T1 entrou no sistema antes de T2, que entrou no sistema antes de T3, então sabe-se que o valor de TS(T1)-1 < TS(T2)-2 < TS(T3)-3. Quando T3 efetua write(B) com sucesso, então W-timestamp(B) passa a ser 3. De seguida, quando T1 deseja efetuar write(B), como TS(T1), que é 1, é menor do que W-timestamp(B), que é 3, a operação é rejeitada e é efetuado um rollback sobre T1. Como tal, o escalonamento não é permitido no protocolo timestamp-based.

2.2 a) Este escalonamento também não seria permitido em Strict 2-Phase Locking, uma vez que um dos requisitos é que todos os unlocks sejam feitos no fim da transação. Como a transação T1 realiza unlock(A) antes de fazer lock-S(B), este requisito não é cumprido.

b) Admitindo que T1 entrou no sistema antes de T2, então sabe-se que o valor de TS(T1)-1 < TS(T2)-2. Quando T2 efetua write(B) com sucesso, então W-timestamp(B) passa a ser 2. De seguida, quando T1 deseja efetuar read(B), como TS(T1), que é 1, é menor ou igual do que W-timestamp(B), que é 2, a operação é rejeitada e é efetuado um rollback sobre T1. Como tal, o escalonamento não é permitido no protocolo timestamp-based.

3. Recovery System

3. a) Fase de análise:

» Ficheiro de Log:

LSN	Type	Transaction	Page
10	Update	T1	P1
20	Update	T2	P2
30	Begin_checkpoint	-	-
40	End_checkpoint	-	-
50	Commit	T1	
60	Update	T3	P3
70	Commit	T2	-
80	Update	T3	P2
90	Update	T3	P5

» Tabela das Transações Activas:

Transaction	LastLSN
T1	Commited
T2	Commited
T3	90

» DPT (Dirty Page Table):

Page ID	RecLSN
P1	10
P2	20
P3	60
P5	90

b) Fase de redo:

» Ficheiro de Log:

LSN	Type	Transaction	Page
10	Update	T1	P1
20	Update	T2	P2
30	Begin_checkpoint	-	-
40	End_checkpoint	-	-
50	Commit	T1	
60	Update	T3	P3
70	Commit	T2	-
80	Update	T3	P2
90	Update	T3	P5

» Tabela das Transações Activas:

Transaction	LastLSN
T1	Commited
T2	Commited
T3	90

» DPT (Dirty Page Table):

Page ID	RecLSN
P1	10
P2	80
P3	60
P5	90

c) Fase de undo:

» Ficheiro de log:

LSN	Type	Transaction	Page	undoNextLSN
10	Update	T1	P1	
20	Update	T2	P2	
30	Begin_checkpoint	-	-	
40	End_checkpoint	-	-	
50	Commit	T1		
60	Update	T3	P3	
70	Commit	T2	-	
80	Update	T3	P2	
90	Update	T3	P5	
100	CLR	T3	P5	90
110	CLR	T3	P2	80
120	CLR	T3	P3	null
130	END	T3		

De realçar que, após a fase de undo, as tabelas de transações ativas e de *dirty pages* são eliminadas por outro mecanismo (que é executado em paralelo com o de recuperação da informação) e, por isso, deixam de existir após esta fase.

4. Schema and Index Tuning

4.1.
A query Q1 é apresentada de seguida:

```
Select type, AVG(calories) as avgcalories
From CHEESE
Group By type
```

Por outro lado, a query Q2 é apresentada de seguida:

```
Select cheeseID
From CHEESE
Where Proteins = MAX(Select Proteins From CHEESE)
```

4.2.

Q1 – A solução seria criar um índice *clustered* do tipo *Hash,* sobre o atributo type. Como se pretende fazer uma média de calorias por tipo, a procura torna-se muito mais eficiente caso os queijos do mesmo tipo se encontrem seguidos, evitando percorrer toda a tabela. O fato de ser do tipo Hash é importante, pois pode-se facilmente aceder a todos os queijos do mesmo tipo.

Q2 – A solução seria criar um índice *clustered* do tipo B-Tree, sobre o atributo Proteins, porque como o que é pretendido é devolver os queijos com o valor de proteínas máximo, torna-se mais eficiente caso a tabela esteja ordenada por proteínas, pois evita-se percorrer toda a tabela. O índice B-tree é adequado para queries que contenham a keyword MAX pois executa uma serie de comparações, tendo em conta o valor de Proteins, disponibilizando rapidamente o valor máximo da mesma tendo em conta a página em que está localizada (encontrando rapidamente o nó mais à direita da árvore para o efeito).

4.3.

Q1 – A solução seria utilizar uma *materialized view*, com as seguintes características:

Relação: CHEESE(cheeseID, Type, Producer, Calories, Proteins)
Materialized view: AVGCHEESECALORIES(Type, Avgcalories)

O atributo Avgcalories, como o próprio nome indica, consiste na média de calorias de cada tipo, atualizada automaticamente pelo SGBD à medida que os registos são inseridos e/ou modificados.

Q2 – A solução seria construir duas relações (em vez de uma) utilizando *vertical partitioning*. Assim, teria-se:

CHEESE(cheeseID, Type, Producer, Calories)
CHEESEP(cheeseID, Proteins)

Desta forma, em vez de se ter em memória todos os registos do tipo CHEESE (que contêm vários atributos, por isso gastam muita memória, podendo ser separados por várias páginas), tem-se duas relações nas quais se dividem os atributos pelas duas. Quando se pretende procurar o máximo das proteínas, verificar-se-á apenas a tabela CHEESEP (com menos atributos, logo menos memória usada), aumentando-se portanto a *hit ratio*.

5. Query Tuning

5.1.
A razão para o otimizador não encontrar um bom plano de execução prende-se com o fato de o mesmo não utilizar os índices presentes em expressões aritméticas, como o caso da expressão *(calories * 100 /800).* A solução passa por simplificar a expressão, ficando apenas o atributo *calories* do lado esquerdo da desigualdade.

```
SELECT name
FROM   CHEESE
WHERE  calories < 8
```

5.2.
O otimizador não irá tirar partido do índice na sub-expressão *calories < 90* porque, como os registos estão ordenados de acordo com o atributo *calories*, é apenas feito o scan a tabela enquanto se verificar a condição *calories < 90.*
A query sugerida é:

```
SELECT name
FROM   CHEESE
WHERE  calories < 40 AND calories <90
```

Isto deve-se ao fato de, na expressão *calories > 40,* ser utilizado o índice para aceder ao primeiro registo em que esta condição se verifique e depois ser feito um scan até a condição *calories < 90* já não se verificar.

5.3.
A query sugerida é:

```
SELECT type
FROM   CHEESE
WHERE  calories = 90

UNION ALL

SELECT type
FROM   CHEESE
WHERE  calories = 40
```

Na presença do operador OR, alguns otimizadores nunca utilizam os índices contidos nessas expressões mas, ao utilizar duas expressões unidas pela keyword UNION, o otimizador utiliza os índices e realiza uma união dos resultados, não eliminando os resultados repetidos por causa da utilização da keyword ALL seguida de UNION. Se não se utilizasse a keyword ALL, o SGBD procederia a uma eliminação dos resultados repetidos, mas visto que os conjuntos de resultados *calories = 90* e *calories = 40* são disjuntos, esta verificação apenas representaria um overhead para o sistema.

5.4.
Como nos valores retornados pela query está contido o atributo-chave, então todos os registos retornados são diferentes, logo é desnecessária a utilização do DISTINCT, pois apenas acarretaria um overhead ao verificar se existem registos iguais quando não existem.
A query rescrita é:

```
SELECT cheeseID, type
FROM   CHEESE
```

5.5.
A keyword HAVING deve ser reservada para propriedades agregadas dos grupos de registos. Como não é o caso desta query, então é utilizado o WHERE para retornar os mesmos tuplos que retornaria a keyword HAVING.
A query sugerida é:

```
SELECT AVG(proteins)
FROM CHEESE
WHERE type = "Alverca"
GROUP BY  producer
```

6. Database Tuning

6.1. Através da análise das *queries* apresentadas, conclui-se que a *query* Q2 corresponde a um padrão típico de acesso a uma aplicação OLAP, uma vez que estas aplicações se caracterizam por uma grande capacidade de analisar e manipular uma grande quantidade de informação, através de várias perspetivas, muitas vezes utilizada para fins estatísticos. Desta forma, são analisados vários cenários, disponibilizando dados contidos em bases de dados operacionais, como *Data Warehouses*, identificados por dois ou mais atributos. Desta forma, na query Q2, observa-se uma agregação de vários tipos de queijo (agrupados por tipo através do GROUP BY), retornando-se assim o número de queijo de cada tipo (através do COUNT(*)). Trata-se por tipo, de um padrão observável nas aplicações OLAP. De realçar que as *querys* que interagem com este tipo de aplicações são tipicamente de leitura, como é o caso da *query* Q2.

Por outro lado, a query Q1 corresponde a um padrão típico de acesso a uma aplicação OLTP, pois estas são caracterizadas por transações curtas, em que o processamento da *query* é rápido. Têm normalmente como objetivo o registo de todas transações de uma determinada organização. Consequentemente, a *query* Q1 corresponde a uma interrogação relativamente rápida, não apresentando nenhum tipo de agregação (ao contrário da query Q2). É, por isso, um exemplo de acesso típico a uma aplicação OLTP.

6.2. Sabendo que a *query* Q2 corresponde a um acesso típico a uma aplicação OLAP, e que estas aplicações se caracterizam por apenas ser permitido a inserção e leitura de dados (sendo que, para um utilizador, apenas está disponível a leitura), faz mais sentido utilizar um nível de isolamento baixo (por exemplo, READ UNCOMMITED), uma vez que a probabilidade de existir um conflito entre transações é relativamente pequeno.

Por outro lado, relativamente à *query* Q1 que corresponde a um acesso a uma aplicação OLTP, e sabendo que estas aplicações permitem acessos de leitura, inserção, modificação e exclusão de dados muito frequentemente, faz sentido usar um nível de isolamento mais elevado (por exemplo, do tipo SERIALIZABLE), uma vez que a probabilidade de existirem conflitos entre transações é bastante elevado (devido à constante modificação dos dados).

6.3. Relativamente à *query* Q1, duas otimizações que podiam ser consideradas seriam:

- Do tipo index tuning: Criando um índice *clustering* (preferencialmente do tipo *hash*, mas um do tipo *BTree* seria também vantajoso) sobre o atributo de junção do *join* faria com que os dados estivessem fisicamente ordenados tendo em conta esse atributo. Dessa forma, quando fosse executada a interrogação, o processamento da mesma seria muito mais rápido.
- Do tipo query tuning: Como a cláusula FROM da *query* Q1 será executada antes da cláusula WHERE, e como nessa cláusula (FROM) estão presentes quatro *joins*, a tabela resultante seria de dimensões bastante elevadas, tendo em conta que nessa tabela apenas serão selecionados alguns tuplos tendo em conta a cláusula WHERE. Assim, a solução seria inserir em duas tabelas temporárias os dados filtrados (uma para a tabela REGION filtrada com os tuplos que têm apenas *country='Portugal'* e outra para PRODUCTION filtrada com os tuplos que têm *amount>10,000* e *season='Winter'*). Desta forma, não existiria uma tabela que contivesse todos os dados de todas as tabelas, mas sim os dados previamente filtrados adequadamente. Ou seja, a solução seria:

```
INSERT INTO TEMPR
SELECT *
FROM REGION
WHERE country = 'Portugal'

INSERT INTO TEMPP
SELECT *
FROM PRODUCTION
WHERE amount > 10000 AND season = 'Winter'

SELECT type, producer
FROM CHEESE NATURAL JOIN TEMPP NATURAL JOIN PROVENANCE NATURAL JOIN TEMPR
```