Fpt University

Chapter 7: Practical issues of database application

Objectives

- Understand transactions and their properties (ACID)
- Apply transaction in database application programming
- Understand the roles of indexing techniques
- Implement indexes for query optimization
- Understand what views are for and how to use them
- Understand query execution plan for query optimization analysis

Contents

1. Transaction in SQL
2. Indexes in SQL & Query optimization
3. Views

Fpt University

Introduction

- DB User operates on database by querying or modifying the database
- Operations on database are executed one at a time
- Output of one operation is input of the next operation
- So, how the DBMS treats simultaneous operations?

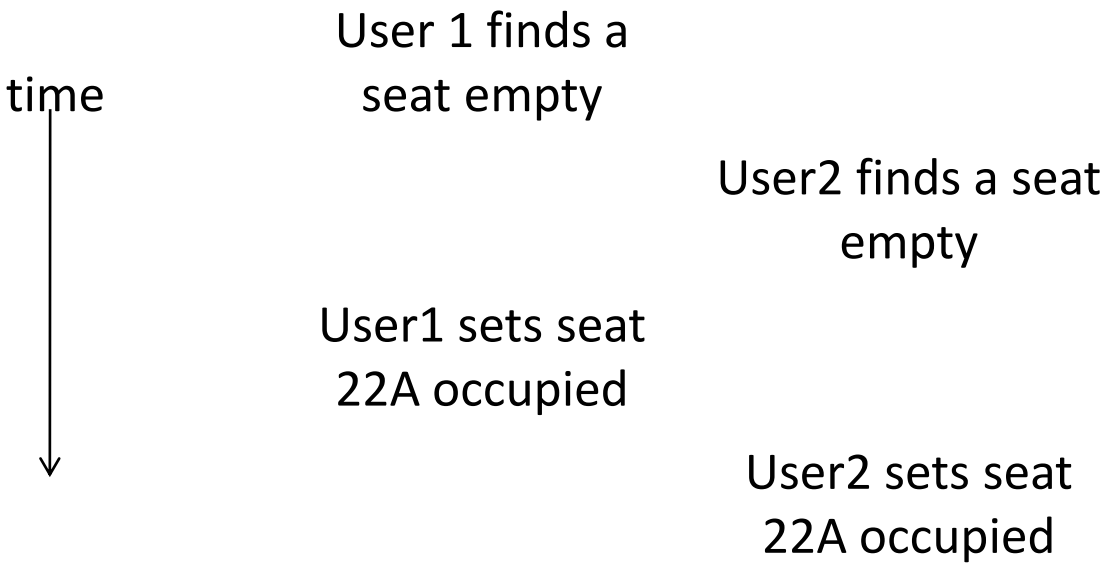
Fpt University

Serializability

- In applications, many operations per second may be performed on database
- These may operate on the same data
- We'll get unexpected results

Serializability

Example: Two users book the same seat of the flight



Serializability

- Transaction is a group of operations that need to be performed together
- A certain transaction must be serializable with respect to other transactions, that is, the transactions run serially – one at a time, no overlap

Fpt University

Atomicity

A certain combinations of database operations need to be done atomically, that is, either they are all done or neither is done

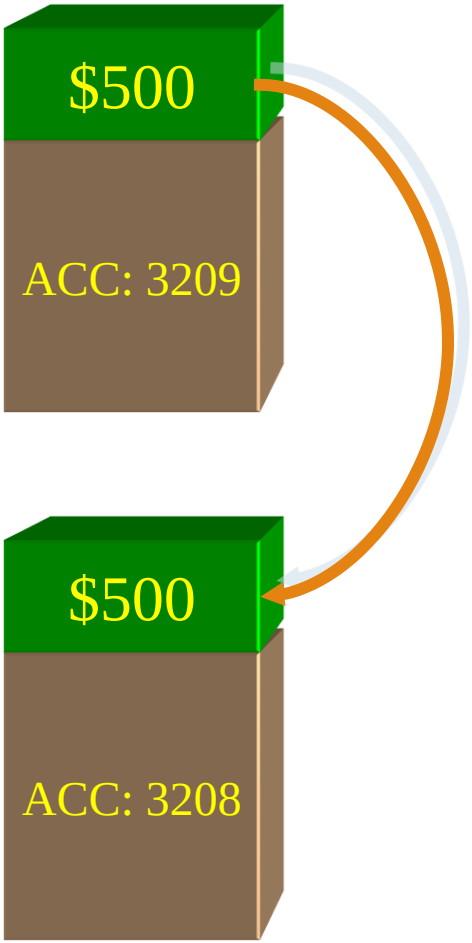
Atomicity

Example: Transfer $500 from the account number 3209 to account number 3208 by two steps

- (1) Subtract $500 from account number 3209
- (2) Add $500 to account number 3208

What happen if there is a failure after step (1) but before step (2)?

Atomicity



A Banking Transaction

```
UPDATE savings_accounts
     SET balance = balance - 500
     WHERE account = 3209;
```

Decrement Savings Account

```
UPDATE checking_accounts
     SET balance = balance + 500
     WHERE account = 3208;
```

Increment Checking Account

```
INSERT INTO journal VALUES
     (journal_seq.NEXTVAL, '1B'
     3209, 3208, 500);
```

Record in Transaction Journal

```
COMMIT WORK;
```

End Transaction

Transaction Ends

Transactions

- Transaction is a collection of one or more operations on the database that must be executed atomically
- That is, either all operations are performed or none are
- In SQL, each statement is a transaction by itself
- SQL allows to group several statements into a single transaction

Transactions

Transaction begins by SQL command START TRANSACTION

Two ways to end a transaction

- The SQL statement COMMIT causes the transaction to end successfully
- The SQL statement ROLLBACK causes the transaction to abort, or terminate unsuccessfully

ACID properties of Transaction

- Atomicity
- Consistency
- Isolation
- Durability

ACID properties of Transaction

Atomicity: a transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

- At the end of the transaction, either all statements of the transaction is successful or all statements of the transaction fail.
- If a partial transaction is written to the disk then the *Atomic* property is violated

Consistency: a transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

Fpt University

ACID properties of Transaction

Isolation: a transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is the execution of a transaction should not be interfered with by any other transactions executing concurrently.

Durability : the changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure..

Fpt University

Read-Only Transactions

- A transaction can read or write some data into the database
- When a transaction only reads data and does not write data, the transaction may execute in parallel with other transactions
- Many read-only transactions access the same data to run in parallel, while they would not be allowed to run in parallel with a transaction that wrote the same data

Transactions

SQL statement set read-only to the next transaction

- SET TRANSACTION READ ONLY;

SQL statement set read/write to the next transaction

- SET TRANSACTION READ WRITE;

Fpt University

Dirty Reads

- **Dirty data**: data written by a transaction that has not yet committed
- **Dirty read**: read of dirty data written by another transaction
- Problem: the transaction that wrote it may eventually abort, and the dirty data will be removed
- Sometimes the dirty read matters, sometimes it doesn’t

Dirty Reads

Transaction 1	Transaction 2	Logical value	Uncommitted value	What transaction 2 show
START T1		3		
UPDATE A=5	START T2	3	5	
. . .	SELECT @v=A	3	5	5
ROLLBACK	UPDATE B=@v	3		5

Dirty Reads

We can specify that dirty reads are acceptable for a given transaction

- SET TRANSACTION READ WRITE
  ISOLATION LEVEL READ UNCOMMITTED;

Other Isolation Levels

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITED;

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

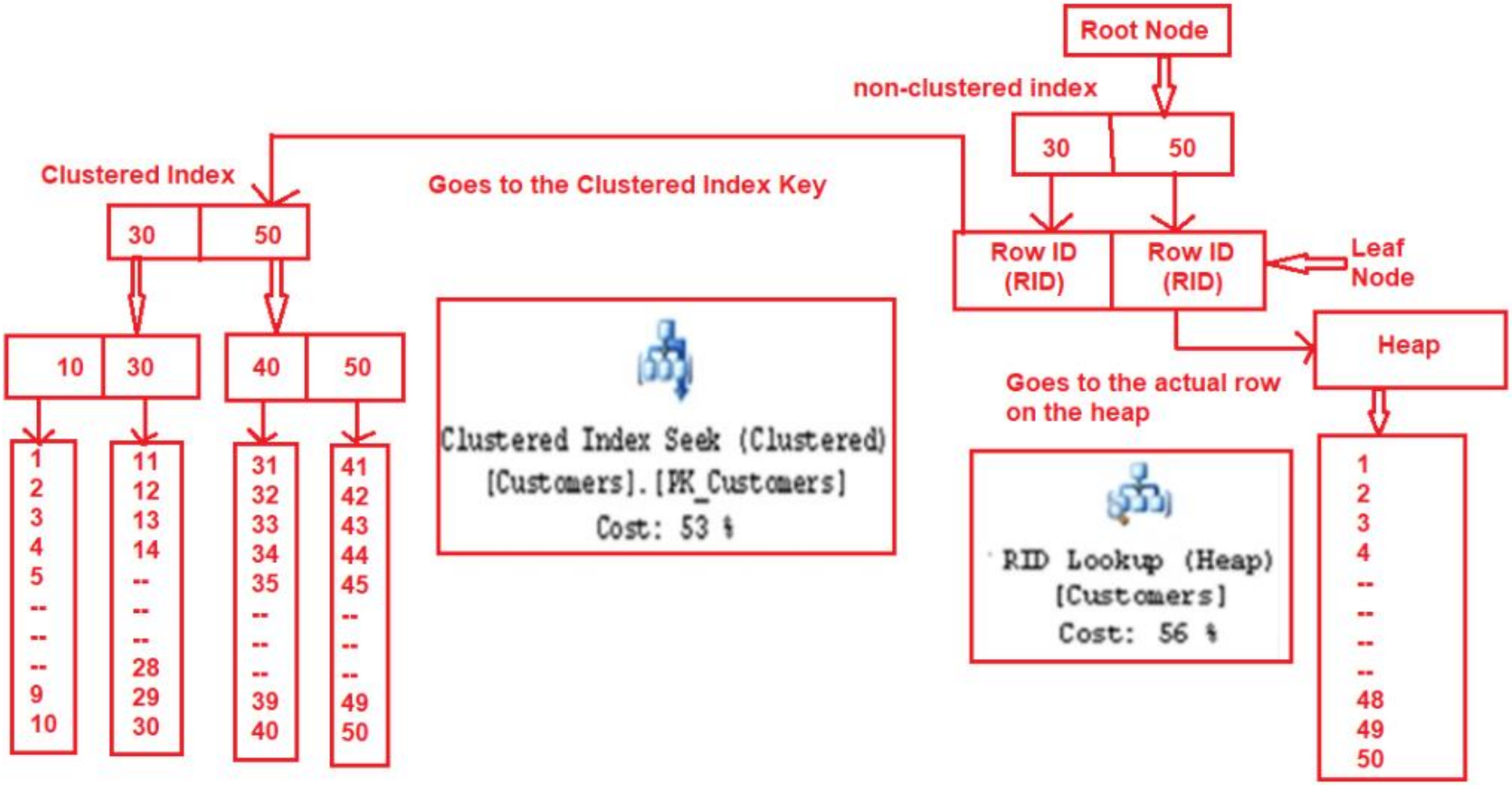
Index overview

- An index on attribute A is a data structure that makes it efficient to find those tuples that have a fixed value for attribute A
- Can dramatically speed up certain operations:
  - Find all R tuples where R.A = v
  - Find all R and S tuples where R.A = S.B
  - Find all R tuples where R.A > v (sometimes, depending on index type)
  - Example

    SELECT * FROM Student WHERE name = 'Mary'
    - Without index: Scan all Student tuples
    - With index: Go "directly" to tuples with name='Mary'

    Indexes are built on single attributes or combinations of attributes.

Type of indexes



Indexes implementation on SQL

```
CREATE CLUSTERED INDEX index_name ON
dbo.Tablename(ColumnName1,ColumnName2...)
```

```
CREATE NONCLUSTERED INDEX index_name
ON dbo.Tablename(ColumnName1,
ColumnName2...)
```

```
DROP INDEX index_name
```

//demo required

Index design guidelines

- Choosing which indexes to create is a difficult and very important design issue. The decision depends on size of tables, data distributions, and most importantly query/update load.
  - Table Size: enough large. Why?
  - Column Types:
    - Small size (INT, BIGINT). Should create clustered index on Unique and NOT NULL field.
    - Identity field (automatically increment)
    - Static field
  - Number of indexes
  - Storage Location of Indexes
  - Index Types
  - Query design

Fpt University

Relations vs. Views

Relations

- Actual exist in database in some physical organization
- Defined with a CREATE TABLE statement
- Exist indefinitely and not to change unless explicit request

Virtual views

- Do not exist physically
- Defined by an expression like a query
- Can be queried and can even be modified

Views

- A view just a relation, but we store a definition rather than a set of tuples

- When do we use view?
- Syntax:

```
CREATE VIEW ViewName
AS
SELECT * / RequiredColumnNames
FROM TableName
```

Types of views in SQL Server:

- Simple view or Updatable views: single table, can INSERT, UPDATE, DELETE through view.
- Complex view or non-updatable views: multi tables

Virtual Views

Example 1:

- Create a view to list all employees who are in Department number 1

```
IF OBJECT_ID('Employee_Dep1','V') IS NOT NULL
drop view Employee_Dep1
GO

CREATE VIEW Employee_Dep1 AS
SELECT * FROM tblemployee WHERE depnum=1;
GO
```

Virtual Views

The simplest form of view definition is

CREATE VIEW <view-name> AS <view-definition>;

- The <view-definition> is a SQL query

Virtual Views

Example 1:

- Create a view to list all employees who are in Department number 1

```
IF OBJECT_ID('Employee_Dep1','V') IS NOT NULL
drop view Employee_Dep1
GO

CREATE VIEW Employee_Dep1 AS
SELECT * FROM tblemployee WHERE depnum=1;
GO
```

Querying Views

Example 2

- Find all employees who work in department 1

```
SELECT * FROM tblEmployee te WHERE te.depNum=1
GO

SELECT * FROM Employee_Dep1
GO
```

Querying Views

Find all dependents of the employees who work in department 1

```
SELECT *
FROM Employee_Dep1 ed1, tblDependent d
WHERE ed1.empSSN=d.empSSN


SELECT *
FROM  (SELECT *
       FROM tblEmployee
       WHERE depNum=1
      ) ed1, tblDependent d
WHERE ed1.empSSN=d.empSSN
```

Renaming Attributes

Example 3:

- Create view for all employees of Department number 1, including: SSN, Fullname, Age, Salary, Sex

```
IF OBJECT_ID('Employee_Dep1','V') IS NOT NULL
drop view Employee_Dep1
GO

CREATE VIEW Employee_Dep1 AS
SELECT te.empSSN AS 'Mã số nhân viên',te.empName AS 'Họ và tên',
        YEAR(GETDATE())-YEAR(te.empBirthdate) AS 'Tuổi',
        te.empSalary AS 'Lương',
        CASE WHEN te.empSex ='F' THEN N'Nữ' ELSE N'Nam' END AS 'Giới tính'
FROM tblEmployee te  WHERE te.depNum=1;
GO
```

Fpt University

Modifying Views

With *updatable views*, the modification is translated into an equivalent modification on a base table

The modification can be done to the base table

View Removal

As we know, Employee_Dep1 is associated to tplEmployee relation

DROP VIEW Employee_Dep1;

- Delete the definition of the view
- Does not effect on tplEmployee relation

DROP TABLE tplEmployee;

- Delete tplEmployee relation
- Make the view Employee_Dep1 unusable

Updatable Views

We can modify views that are defined by selecting some attributes from one relation R, and

- Sub query started by SELECT, not SELECT DISTINCT
- The WHERE clause must not involve R in a sub query
- The FROM clause can only consist of one occurrence of R and no other relation
- The list in the SELECT clause must include enough attributes that for every tuple inserted into the view, we can fill the other attributes out with NULL values or the proper default

Updatable Views

Example 4:

- Create view from table Employee
- Do changes on Employee and review created view
- Do changes on created view and review Employee

Update on table effects on view

```
IF OBJECT_ID('Employee_Dep1v2','V') IS NOT NULL
drop view Employee_Dep1v2
GO

CREATE VIEW Employee_Dep1v2 AS
SELECT te.empSSN,te.empName,te.empSalary,te.empSex
FROM tblemployee te WHERE te.depnum=1;
GO


INSERT INTO tblEmployee (empSSN, empName, empSalary, empSex, depNum)
    VALUES (100000,N'Lê Văn Tám',100000,'M',1)
GO
SELECT * FROM tblEmployee te WHERE te.depNum=1
GO
SELECT * FROM Employee_Dep1v2
GO
```

Update on view effects on table with unexpected result

```
IF OBJECT_ID('Employee_Dep1v2','V') IS NOT NULL
drop view Employee_Dep1v2
GO

CREATE VIEW Employee_Dep1v2 AS
SELECT te.empSSN,te.empName,te.empSalary,te.empSex
FROM tblemployee te WHERE te.depnum=1;
GO

INSERT INT Employee_Dep1v2 VALUES (100001,N'Lê Văn Bảy',100000,'M')
GO
SELECT * FROM tblEmployee te WHERE te.depNum=1
GO
SELECT * FROM Employee_Dep1v2
GO
```

Update on view raises error on table

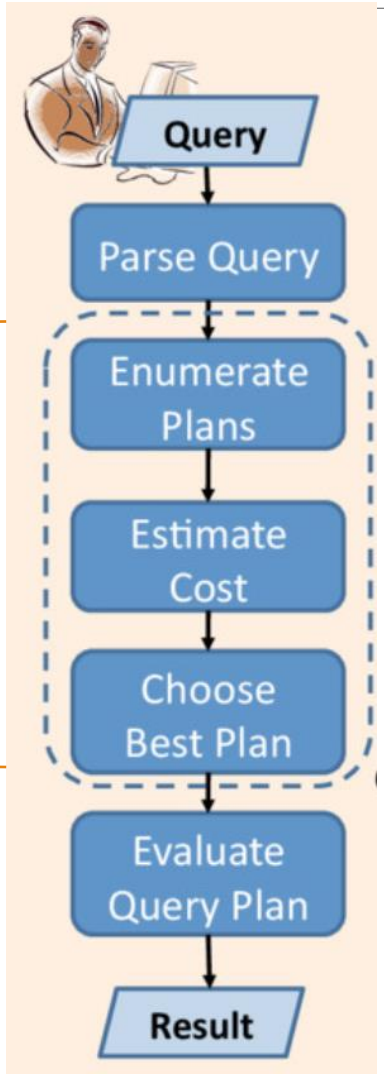
```
IF OBJECT_ID('Employee_Dep1v3','V') IS NOT NULL
drop view Employee_Dep1v3
GO

CREATE VIEW Employee_Dep1v3 AS
SELECT te.empSSN AS 'Mã số nhân viên',te.empName AS 'Họ và tên',
        YEAR(GETDATE())-YEAR(te.empBirthdate) AS 'Tuổi',
        te.empSalary AS 'Lương',
        CASE WHEN te.empSex ='F' THEN N'Nữ' ELSE N'Nam' END AS 'Giới tính'
FROM tblEmployee te  WHERE te.depNum=1;
GO



INSERT INTO Employee_Dep1v3 VALUES (100002,N'Lê Văn Chín',30,90000,N'Nam')
GO
```

Query optimization



- In practice:

1. Define the requirements: Who? What? Where? When? Why?
2. SELECT fields instead of using SELECT *
3. Avoid SELECT DISTINCT
4. Indexing
5. Create joins with INNER JOIN (not WHERE)
6. To check the existence of records, use EXISTS() rather than COUNT()
7. Ignore linked subqueries
8. Use of temp table
9. Don’t run queries in a loop
10. Limit your working data set size