COMP90050
Advanced Database Systems

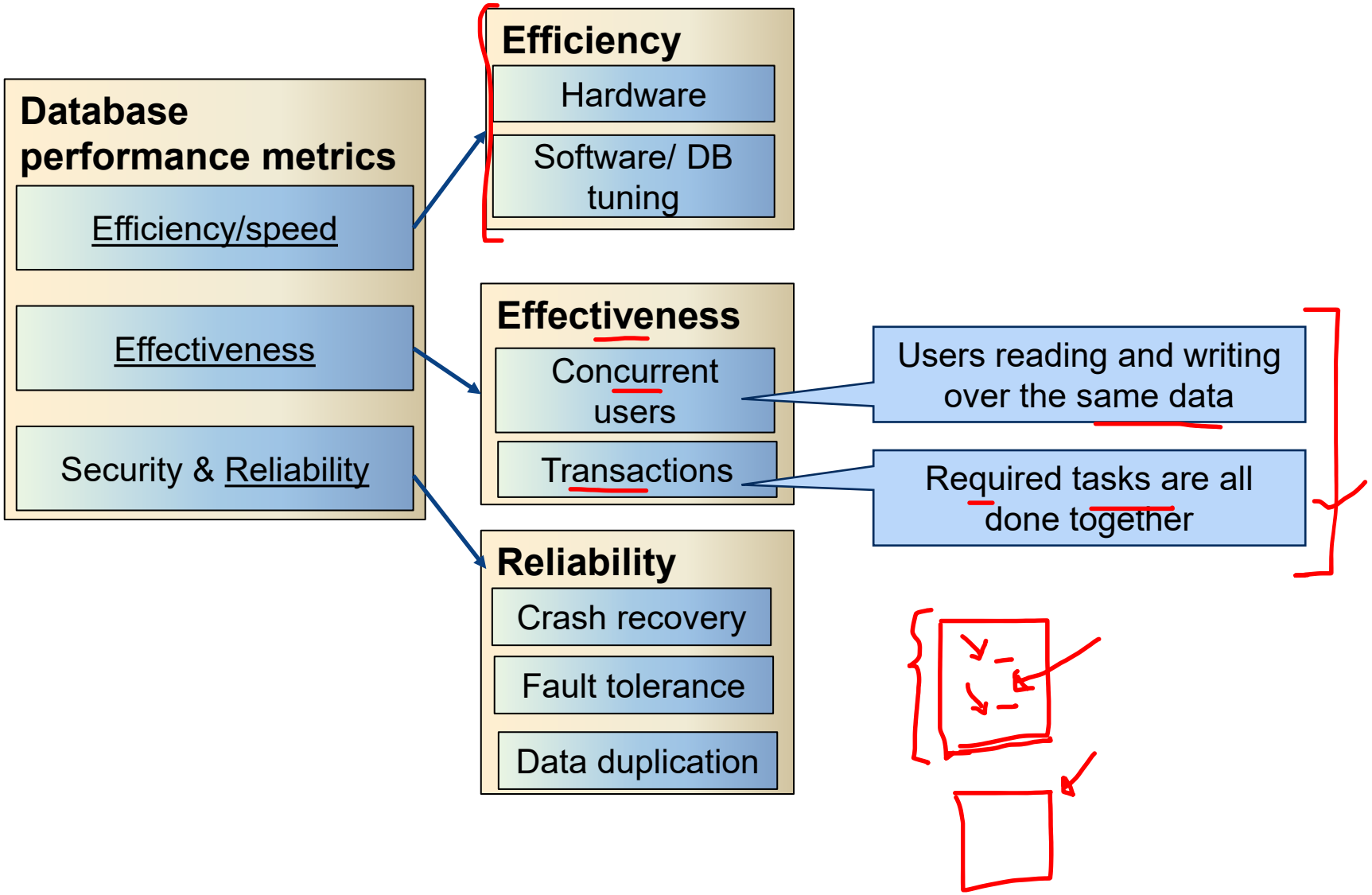
THE UNIVERSITY OF MELBOURNE

Winter Semester, 2023

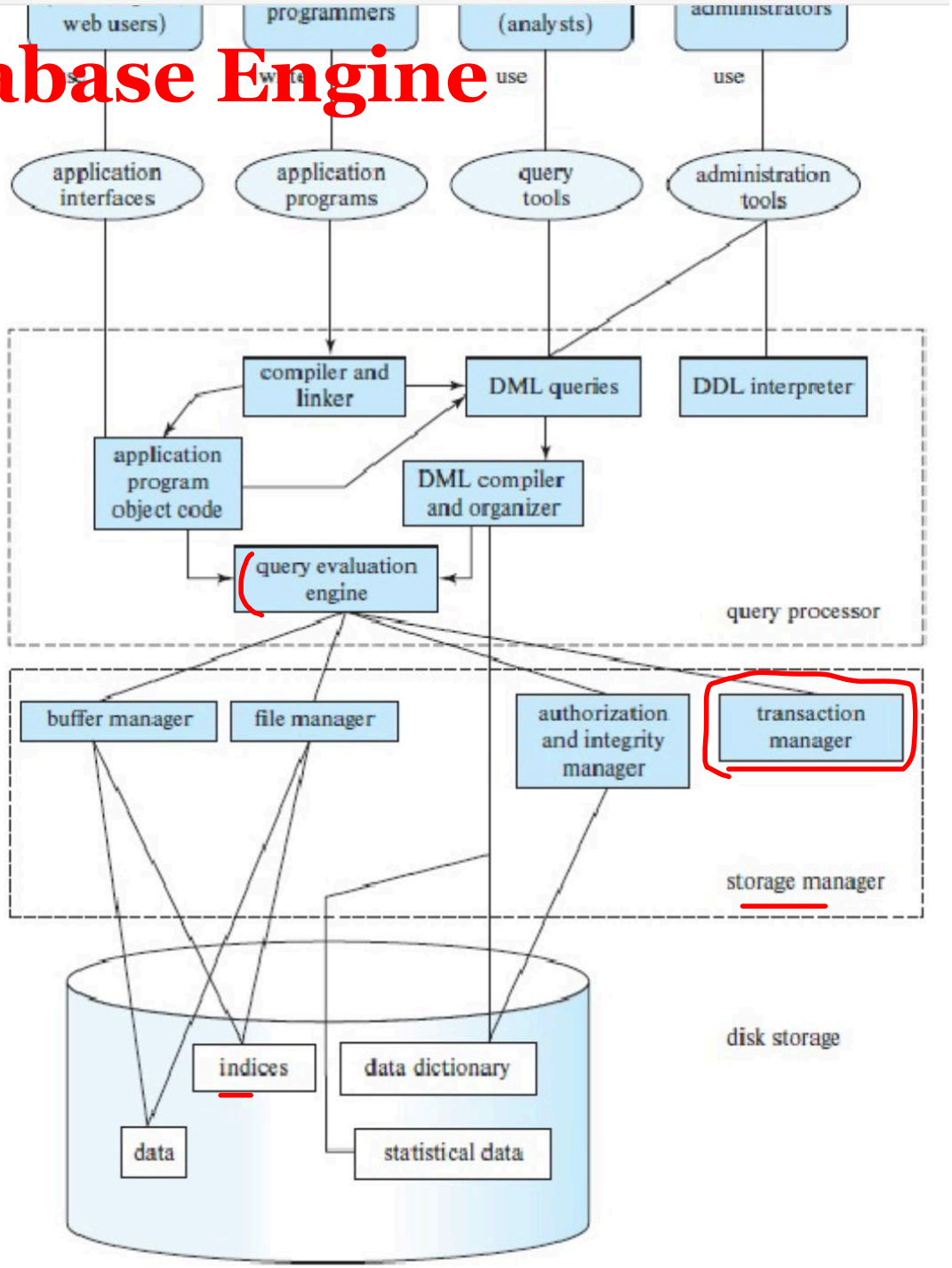
Lecturer: Farhana Choudhury (PhD)

Week 3 part 1-2

Core Concepts of Database management system



Database Engine



Topics

- Transaction concepts
- Different types of transactions
- Concurrent use of data and its issues
- Isolation concepts - ensuring that concurrent transactions leaves the DB in the same state as if they were executed separately
- How to achieve isolation while keeping good efficiency

Database Transactions

Transaction - A unit of work in a database

- A transaction can have any number and type of operations in it
- Either happens as a whole or not
- Transactions ideally have four properties, commonly known as ACID properties

Transaction models

ACID (Atomicity, Consistency, Isolation, Durability) properties:

Atomicity - All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.

Example – A transaction that (i) subtracts $100 if balance >100 (ii) deposits $100 to another account

(both actions with either happen together or none will happen)

Transaction models

ACID (Atomicity, Consistency, Isolation, Durability) properties:

Consistency - Data is in a 'consistent' state when a transaction starts and when it ends – in other words, any data written to the database must be valid according to all defined rules (e.g., no duplicate student ID, no negative fund transfer, etc.)

- What is 'consistent' - depends on the application and context constraints
- It is not easily computable in general
- Only restricted type of consistency can be guaranteed, e.g. serializable transactions which will be discussed later.

Transaction models ...

ACID (Atomicity, Consistency, Isolation, Durability) properties:

Isolation- transaction are executed as if it is the only one in the system.

- For example, in an application that transfers funds from one account to another, the isolation ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

Durability- the system should tolerate system failures and any committed updates should not be lost.

Transaction models (cont.)

Types of Actions

- **Unprotected actions** - no ACID property
- **Protected actions -** these actions are not externalised before they are completely done. These actions are controlled and can be rolled back if required. These have ACID property.
- **Real actions -** these are real physical actions once performed cannot be undone. In many situations, atomicity is not possible with real actions (e.g., firing two rockets as a single atomic action)

Let’s look at a simple example that will be used to understand different types of transactions …

Embedded SQL example in C

(Open Database Connectivity)

```
int main()
{       exec sql INCLUDE SQLCA; /*SQL Communication Area*/
        exec sql BEGIN DECLARE SECTION;
          /* The following variables are used for communicating
                between SQL and C */
          int OrderID; /* Employee ID (from user) */
          int CustID; /* Retrieved customer ID */
          char SalesPerson[10] /* Retrieved salesperson name */
          char Status[6] /* Retrieved order status */
        exec sql  END DECLARE SECTION;
/* Set up error processing */
        exec sql WHENEVER SQLERROR GOTO query_error;
        exec sql WHENEVER NOT FOUND GOTO bad_number;
```

```
 /* Prompt the user for order number */
        printf ("Enter order number: ");
        scanf_s("%d", &OrderID);
/* Execute the SQL query */
        exec sql SELECT CustID, SalesPerson, Status
             FROM Orders
          WHERE OrderID = :OrderID // ":" indicates to refer to  C variable
          INTO :CustID, :SalesPerson, :Status;
/* Display the results */
        printf ("Customer number: %d\n", CustID);
        printf ("Salesperson: %s\n", SalesPerson);
        printf ("Status: %s\n", Status);
       exit();
 query_error:
         printf ("SQL error: %ld\n", sqlca->sqlcode); exit();
 bad_number:
         printf ("Invalid order number.\n"); exit(); }
```

Some concepts…

Host Variables

Declared in a section enclosed by the BEGIN DECLARE SECTION and END DECLARE SECTION. While accessing these variables, they are prefixed by a colon ":". The colon is essential to distinguish between host variables and database objects (for example tables and columns).

Data Types

The data types supported by a DBMS and a host language can be quite different. Host variables play a dual role:

- Host variables are program variables, declared and manipulated by host language statements, and
- they are used in embedded SQL to retrieve database data.

If there is no host language type corresponding to a DBMS data type, DBMS automatically converts the data. So, the host variable types must be chosen carefully.

Error Handling

The DBMS reports run-time errors to the applications program through an SQL Communications Area (SQLCA) by INCLUDE SQLCA. The WHENEVER...GOTO statement tells the pre-processor to generate error-handling code to process errors returned by the DBMS.

Singleton SELECT

The statement used to return the data is a singleton SELECT statement; that is, *it returns only a single row of data.* Therefore, the code example does not declare or use *cursors*.

Reference:

http://msdn.microsoft.com/enus/library/ms714570(VS.85).aspx

Flat Transaction

```
exec sql CREATE Table accounts (
   AccId                    NUMERIC(9),
   BranchId        NUMERIC(9), FOREIGN KEY REFERENCES branches,
   AccBalance    NUMERIC(10),
   PRIMARY KEY(AccId));
**********************************
```

Everything inside BEGIN WORK and COMMIT WORK is at the same level; that is, the transaction will either survive together with everything else (commit), or it will be rolled back with everything else (abort)

Flat Transaction ...

```
exec sql BEGIN DECLARE SECTION;
        long AccId, BranchId, TellerId, delta, AccBalance;
exec sql END DELCLARATION;


/* Debit/Credit Transaction*/
DCApplication()
{read input msg;
exec sql BEGIN WORK;
AccBalance = DodebitCredit(BranchId, TellerId, AccId, delta);
send output msg;
exec sql COMMIT WORK;
}
```

```
/* Withdraw money -- bank debits;  Deposit money – bank credits */
Long DoDebitCredit(long BranchId,
            long TellerId, long AccId, long AccBalance, long delta){
   exec sql UPDATE accounts
   SET AccBalance =AccBalance + :delta
   WHERE AccId = :AccId;

   exec sql SELECT AccBalance INTO  :AccBalance
   FROM accounts WHERE AccId = :AccId;

   exec sql UPDATE tellers
   SET TellerBalance = TellerBalance + :delta
   WHERE TellerId = :TellerId;

   exec sql UPDATE branches
   SET BranchBalance = BranchBalance + :delta
   WHERE BranchId = :BranchId;


   Exec sql INSERT INTO history(TellerId, BranchId, AccId, delta, time)
   VALUES( :TellerId, :BranchId, :AccId, :delta, CURRENT);
return(AccBalance);
}
```

Let's include a check on the account balance and refusing any debit that overdraws the account.

```
…
DCApplication(){
        read input msg;
        exec sql BEGIN WORK;
        AccBalance = DodebitCredit(BranchId, TellerId, AccId, delta);
        if (AccBalance < 0 && delta < 0){
                exec sql ROLLBACK WORK;
        }
        else{
                send output msg;
                exec sql COMMIT WORK;
        }
}
```

Limitations of Flat Transactions

Flat transactions do not model many real applications.

E.g. airline booking

BEGIN WORK

S1: book flight from Melbourne to Singapore

S2: book flight from Singapore to London

S3: book flight from London to Dublin

END WORK

Problem: from Dublin if we cannot reach our final destination instead we wish to fly to Paris from Singapore and then reach our final destination.

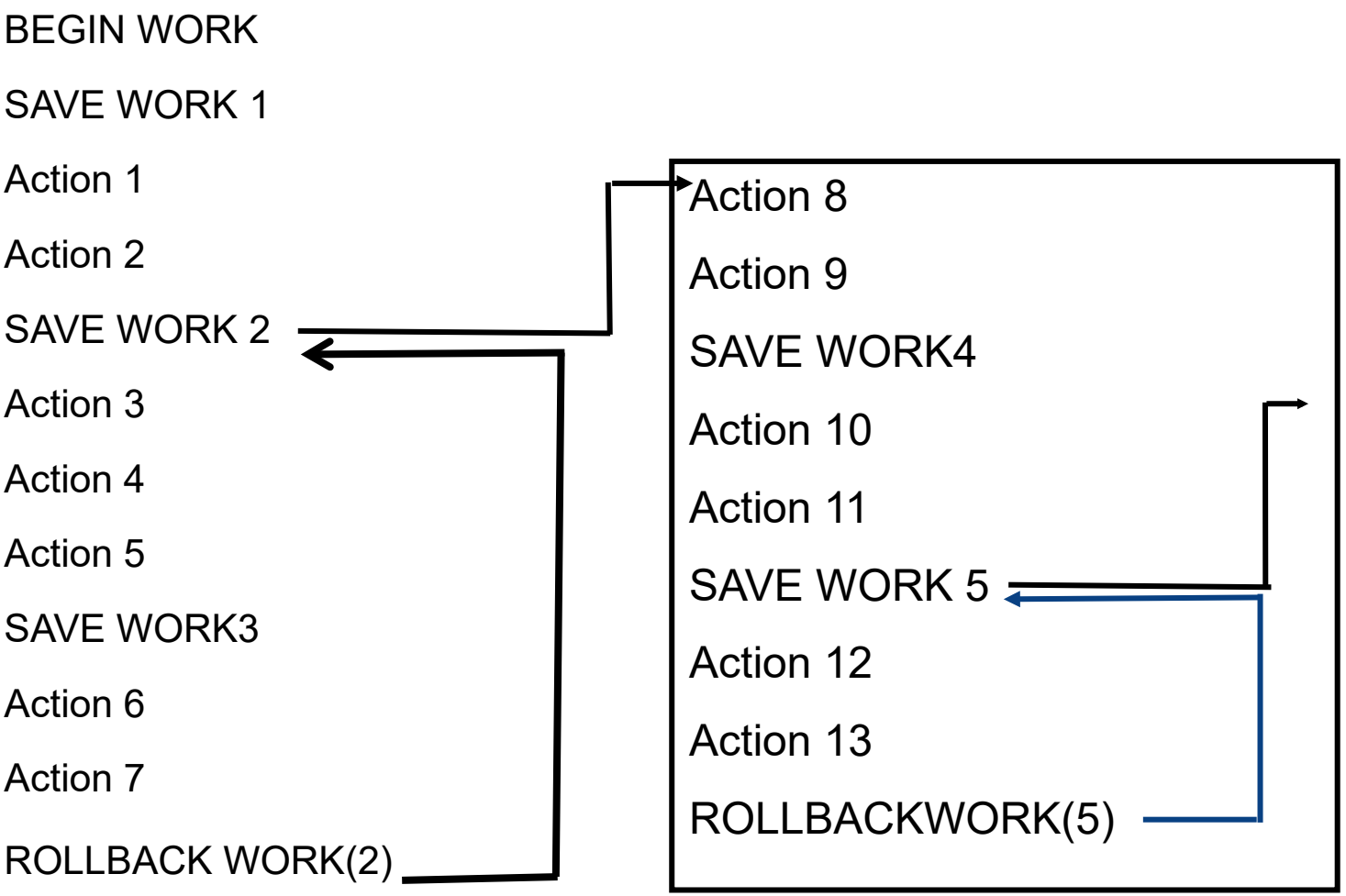
If we roll back we need to re do the booking from Melbourne to Singapore *which is a waste.*

Limitations of Flat Transactions ...

```
IncreaseSalary()
{       real percentRaise;
        receive(percentRaise);
        exec SQL BEGIN WORK;
                exec SQL UPDATE employee
                set salary = salary*(1+ :percentRaise)
                send(done);
        exec sql COMMIT WORK;
        return
}
```

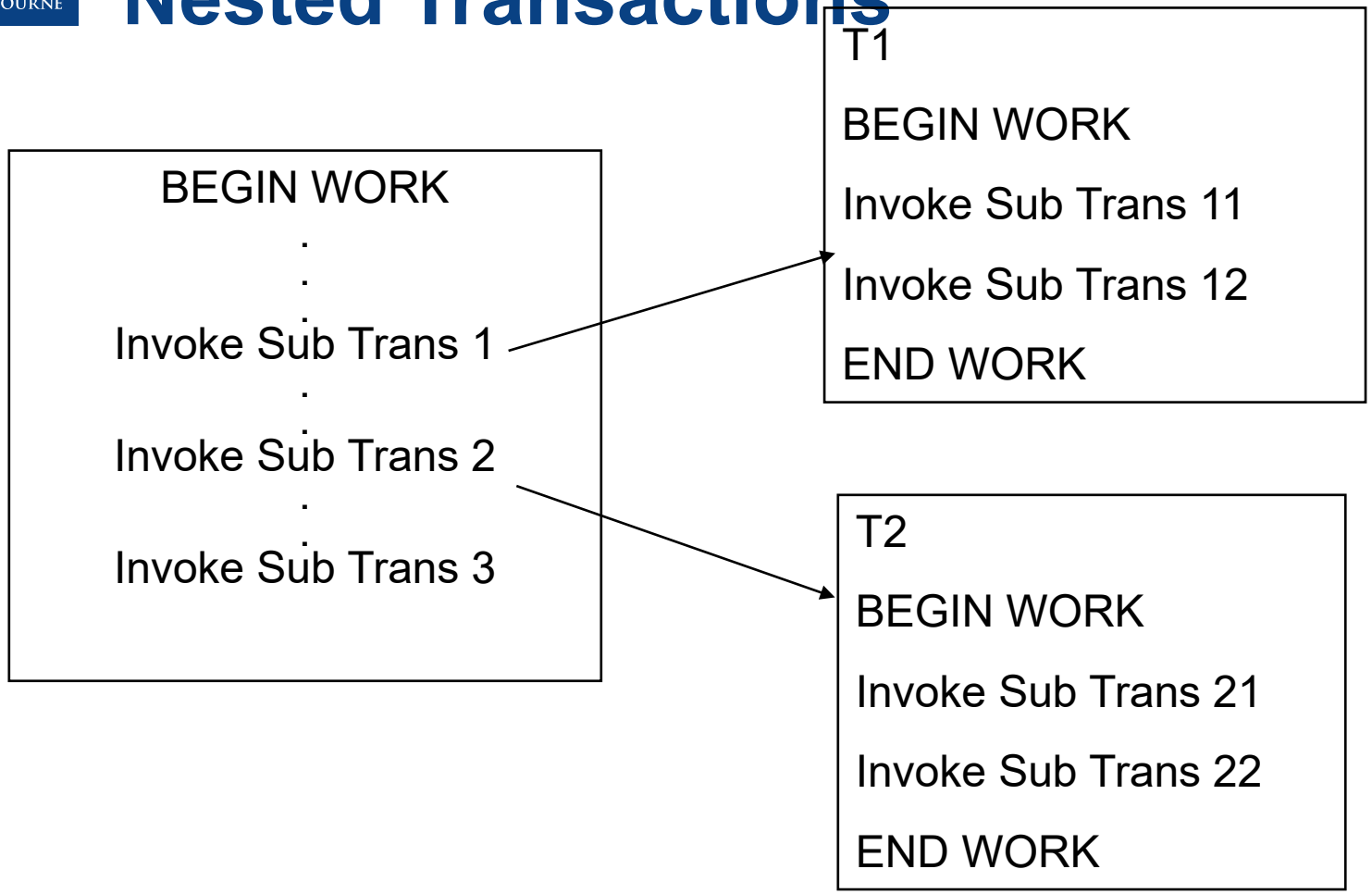
This can be a very long running transaction. ***Any failure of transaction requires lot of unnecessary computation.***

Flat transaction with save points



Reading - Chained transactions

Nested Transactions



Nested Transaction Rules

Commit rule

- A subtransaction can either commit or abort, however, **commit cannot take place unless the parent itself commits.**

- Subtransactions have A, C, and I properties but not D property unless all its ancestors commit.

- Commit of a sub transaction makes its results available only to its parents.

Roll back Rules

If a subtransaction rolls back, all its children are forced to roll back.

Visibility Rules

Changes made by a subtransaction are visible to the parent only when the subtransaction commits. All objects of parent are visible to its children. Implication of this is that the **parent should not modify objects while children are accessing them.** This is not a problem as parent does not run in parallel with its children.

Reading – Open-nested transactions

Transaction Processing Monitor

The main function of a TP monitor is to *integrate* other system components and manage resources.

- TP monitors manage the transfer of data between clients and servers
- breaks down applications or code into transactions and ensures that all databases are updated properly
- It also takes appropriate actions if any error occurs

TP monitor services

Heterogeneity: If the application needs access to different DB systems, local ACID properties of individual DB systems is not sufficient. Local TP monitor needs to interact with other TP monitors to ensure the overall ACID property. A form of 2 phase commit protocol must be employed for this purpose (will be discussed later).

Control communication: If the application communicates with other remote processes, the local TP monitor should maintain the communication status among the processes to be able to recover from a crash.

TP Services ...

Terminal management: Since many terminals run client software, the TP monitor should provide appropriate ACID property between the client and the server processes.

Presentation service: this is similar to terminal management in the sense it has to deal with different presentation (user interface) software -- e.g. X-windows

Context management: E.g. maintaining the sessions etc.

Start/Restart: There is no difference between start and restart in TP based system.