Different Transactional Models

1. Flat transaction
   (a) Begin Transaction. Work more work. . . .End
      i. Keywords commit, abort. Cover the syntax/semantics of a transaction.
      ii. No structure, single commit or abort, can have program flow, however, there cannot be transactions inside transactions, etc.
   (b) Makes all the work inside the transaction invisible from the application perspective
      i. Databases may be temporarily inconsistent during a transaction, i.e. actions are not truly done all at the same time. But the inconsistencies must only be seen by the executing transaction (they are isolated) and only exist during the transaction. (OV Figure 10.1)
   (c) Simplest transaction, what people mean when they say transaction
   (d) Generally the only transaction supported explicitly at the application programming level
2. Transactions are both external and internal
   (a) Used by DB internally to resolve issues of:
      i. Deadlock
      ii. Failure
      iii. The DB reserves the right to abort any transaction in the middle of their execution
3. Limitations of flat transactions
   (a) Do not model real life work flow and dependencies well.
   (b) Do not integrate well into mixed models with other actors (other computers outside of transactional domain or even humans)
4. Example 1 where flat transactions fail – Trip Planning
   (a) In a flat model
      i. Begin Work
      ii. Make a plane reservation
      iii. Make a hotel reservation
      iv. Make a rental car reservation
      v. Commit Work
   (b) Two problems
      i. All operations are executed serially, even though they are independent
      ii. A failure in rental car reservation requires the whole transaction to be rolled back to the start.
5. Another example – bulk updates
   (a) Suppose you are a credit card company in financial trouble and need to raise your rates to survive. You have a database of all your clients.. Why not use the following transaction?

```
Begin Work
Update CustAccount
        set interest = interest * 1.05
Commit Work
```

(b) While conceptually simple, this transaction affects every customer in your database in the same transaction, $i.e$ no where clause. This transaction could take hours to complete and therefore the system is exposed to losing all of its work for even the smallest failure.

(c) However, we want ACID guarantees

(d) Placing each update in its own transaction is no solution, because then there is no way to tell what has been updated and what has not on a restart.

(e) The DB can take advantage of the log to look and see where to restart the transaction, but this is a "dirty" solution because the log is not part of the data model

i. Sadly, making the log part of the data model is exactly how this problem is solved.

6. There are solutions to these problems but they come at a cost

(a) Complex transactions can be addressed by nesting or savepoints

(b) Bulk updates can be addressed with chained transactions or sagas

(c) However, all extensions to flat transactions are incompatible with the fundamental transaction model and most violate transactions semantics is some way

7. There are differing opinions of whether extending transaction models is a good thing

(a) "The discussion of the limitations of ACID transactions and the description of suggested remedies must not create the impression that ACID transactions are a temporary solution that must be dispensed with.... To the contrary, flat transactions and the techniques to make them work account for more than 90% of this book " – Gray and Reuter

(b) "Flat transaction model relatively simple and short activities very well. However, they are less appropriate for modeling longer and more elaborate activities." – OV

(c) Both are correct.

i. Gray and Reuter take a programmatic point-of-view. Most things can and should be done with flat transactions. They solve most of the world's problems and enforce ACIDity.

ii. OV want to use extended transactional models to help computers run complex tasks that do not fit into a flat model. Also a worthy goal. But, more focused on interfacing non-computers systems. Not the point-of-view of a database systems person.

8. Views of transactions

(a) Finite state machine – figure 4.5 GR

i. While FST model single transactions well they fail to capture relationships between multiple transactions

(b) Rather, we use a less formal, but more expressive model of atomic state control blocks. This can capture both

i. Structural dependencies. A transaction B, invoked by transaction A, might think that it can commit, but it cannot really commit until A decides upon all or nothing. If A aborts, B (which wanted to commit) must also abort. This is a system ordering problem

ii. Dynamic dependencies, a transaction B might think that it can commit, but depend upon a value read from transaction A, which is as of yet uncommitted. If A aborts, B must also abort. Only after A commits can B commit.

A. dynamic dependencies can occur between transactions that are not structurally related through any shared data item.

(c) Figure GR 4.6, graphical representation

i. Actions (abort, begin, commit)

ii. Transaction identifier – globally unique
iii. Outcomes, aborted or committed
iv. To this we add a system transaction, which exists forever and cannot commit. It can abort, this is a crash.
v. Interactions between these atomic states are represented as triggers
vi. Develop 4.7 showing trigger and life of a transaction

9. Save points idea is to let the application a save a certain amount of work that is completed.
   (a) implemented by the SAVE WORK command which returns a monotonically increasing handle
   (b) when transactions start they immediately create save point 1
      i. there is a subtle difference between rolling back to save point 1 and aborting a transaction, in that an aborted transaction becomes detached from its process and the identifier goes away. Aborted trans. cannot be restarted, whereas save point 1 is an active position in the original transaction.
   (c) Develop figure 4.9
      i. Monotonic numbers
      ii. The ACID properties are broken for the caller
         A. Lots of control by the application, must be careful
         B. Concurrent transactions cannot see the break in the ACID
      iii. Basically, the application is given a view into the log, and is allow to rollback to a given log entry, but savepoints do not hit the real DB until commit time.

10. Chained transactions
    (a) Turn a big transaction into a series of small, dependent transactions
    (b) Figure 4.10
    (c) The subsequent transactions see a changed version of the database
       i. Particularly if there are concurrent accesses from other transactions.
    (d) Only guarantee is that commitment of one starts the next one.
    (e) Restart handling GR 4.11

11. Comparing to savepoint model and chained transactions
    (a) Both allow substructure to long running jobs
    (b) In chained, rollback is restricted to the active transaction only
    (c) Savepoints do not preserve work across a system crash whereas chained transactions do.

12. Nested transactions
    (a) Generalization of savepoints
       i. Savepoints are sequentially related
       ii. Nested transactions are hierarchically related
       iii. Allow for parallel actions
    (b) What is nesting
       i. Tree of transactions
       ii. Flat transactions at the leaves
       iii. Subtransactions (children) can only *finally* commit if owner (parent) commits
    (c) Figure 4.13

(d) Powerful concept in scoping rollbacks

(e) HW Problem: Nested transactions can be emulated using savepoints. How? Describe a interpreter that operates on the graphical representation of a nested transaction can implement it with savepoints.

13. Distributed transactions – typically a flat transaction that runs in a distributed environment.

    (a) However, the distributed nature gives it structure making it similar to a nested transaction

    (b) Suppose a flat transaction T that operates on X and Y, for Y that must be accessed at another site.

        i. T creates flat transactions T1 for accessing X and T2 for accessing Y

        ii. Invocation is similar to a nested transaction

        iii. But the dependencies are different, GR figure 4.16

    (c) We can see the circular dependencies in the commit. T1 and T2 cannot commit unless T will commit and T cannot commit until T1 and T2 commit.

    (d) Whole section of class (distributed commit protocols) to solve this problem

14. Multi-level transactions – a generalized and more liberal version of nested transactions

    (a) Allows for an early commit of a sub-transaction, also called a pre-commit

        i. Violation of ACID properties might result from pre-commit data being visible while the owner transaction is running, but these changes can be *isolated*.

    (b) Committed sub-transactions can be undone through a compensating transaction. So the final state of the DB will be the same, but not through the rollback mechanism.

    (c) *What is the downside of this approach?* A compensating transaction must be retained for every sub-transaction until the owner transaction commits. A little complex to implement and certainly less efficient.

    (d) Figure 4.17

    (e) *Has anyone noticed the real problem?* In figure 4.17, the compensating transaction is not allowed to abort. So what if the system encounters deadlock, or unavailability, or device failure that would prevent CN from committing, uh oh.

        i. This is not necessarily a totally new problem. In general, it is assumed that aborts always work. If any abort fails, the DB is in an inconsistent state. However, abort failure is does not have some of the locking problems that compensating transactions have. This is a subject for a later date.

    (f) *What is the real advantage?* A corollary of the real problem. Sub-transactions can release resources that nested transactions are holding and increase overall system performance.

        i. Consider a B-Tree. When inserting a tuple as part of a sub-transaction, a transaction splits a page and balances the tree from the top level. This would require the tree root to be locked and the transaction should hold the whole tree until commit time. So on abort, the tree can be returned to its original form. Until commit, the tree is unavailable. However, a nested tran would release locks at pre-commit.

        ii. This example is not perfect, because many systems choose to have data structure integrity treated separately from transactions, but it makes the point.

15. Open-Nested Transactions – the "anarchic" version of multi-level transactions, a open-nested transaction has an owning transaction that fires off many top-level actions that can commit/abort independently

16. Long-Lived Transactions – transactions that run over many tuples and take a long time

    (a) Problem – outer level ACID properties, require that all work is undone in a system crash, but this is too high a price to pay on restart.

    (b) Savepoints and nesting do not help, because they enforce outer-level ACIDity

    (c) Transaction chaining allows completed results to be kept and only rolls back the active transaction.

i. *Does this meet the needs of long lived computations, like updating a million accounts?* No, because the chain does not keep any information on where a crash occurred, it does not let the application know where to pick up at restart. The handle's returned to the program invoking the chain are lost on system crash.

(d) All the necessary data to restart the computation are in the log, but we do not want system applications to have to deal with the log abstraction

(e) The real problem, the system crash has taken away the applications execution context. The actions the application takes are a function of external values, in this case values in the DB.

(f) Solution batch contexts. Application operates out two tables. The table it is updating and a content table it uses to keep track of its own results.

i. If the application starts and the batch context is empty, then it does the whole task.
ii. Otherwise, the application starts where the batch context tells it to.
iii. E.g. update accounts by account number, in groups of 100000.
iv. Write the batch context +=100000 after each small transaction
v. Outer level atomicity and isolation are lost
vi. But perhaps outer-level atomicity is not what the application needs. *Huh?*

17. Sagas – combination of chained transactions and compensating transactions

(a) Figure 4.22

(b) Employ chaining technique to save work in the middle of long transactions

(c) Use compensating transactions (a la multi-level trans) to be able to move to a DB consistent point

(d) Like multi-level trans, destroy atomicity of a trans

18. Workflows – are all these types of transactions not enough?

(a) Transactional models are very focused on ACID semantics. That is, they are very system oriented.

(b) However, there are many examples of computer driven processes where the work is not conducted by a system, rather by a human. Human driven workflows

i. Computer driven manufacturing and processing
ii. Humans are eventually responsible for the consistency of many actions, regardless of the transaction model driving the process.

(c) The real problem is to combine DB state and human action

i. Corrigo – property management with a database and humans
ii. DB manages
A. workflow
B. inventory, restocking
C. job scheduling
D. billing
E. statistics and reporting
iii. Humans manage
A. repairs
B. inputs to system, calls and problem entry and definition
C. fault detection

(d) What is a workflow: an *activity* (as opposed to a transaction) with open nesting semantics

i. Permits partial results to be visible outside of activity boundaries
ii. Has atomic and isolated sub-actions with compensating actions
iii. Has non-atomic (unprotected and real) actions with contingency task
A. contingency is different that compensating. Compensating guarantees return to the original state.