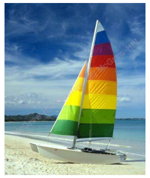
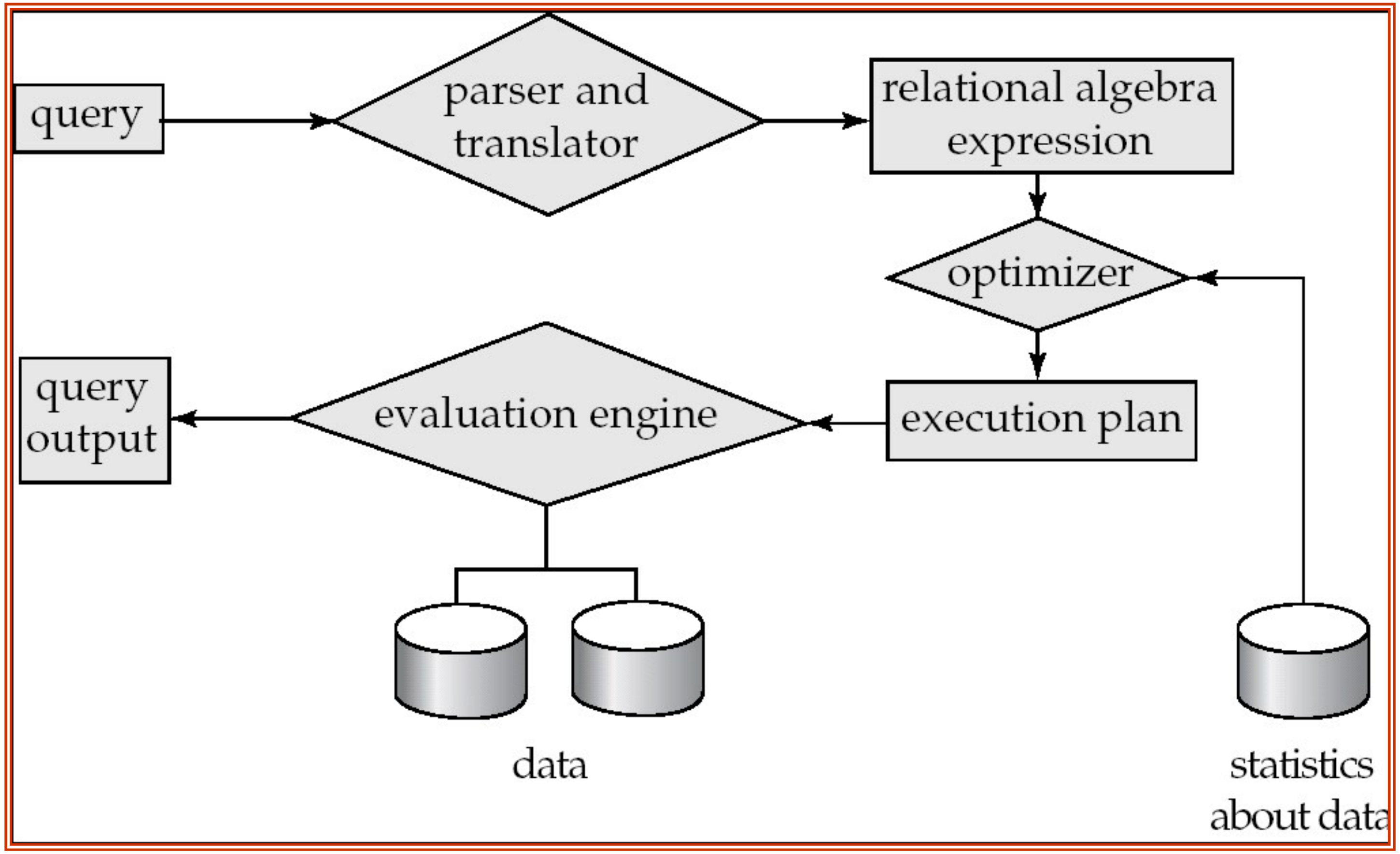


Introduction to Query Processing and Transactions

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Basic Steps in Query Processing : Optimization (example)

select *balance*

from *account*

where *balance<2500*

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{balance<2500}(\Pi_{balance}(account))$ is equivalent to
    $\Pi_{\text{balance}}(\sigma_{\text{balance}<2500}(account))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
  - E.g., can use an index on *balance* to find accounts with balance < 2500,
  - or can perform complete relation scan and discard accounts with balance $\geq$ 2500

Basic Steps: Optimization (Cont.)

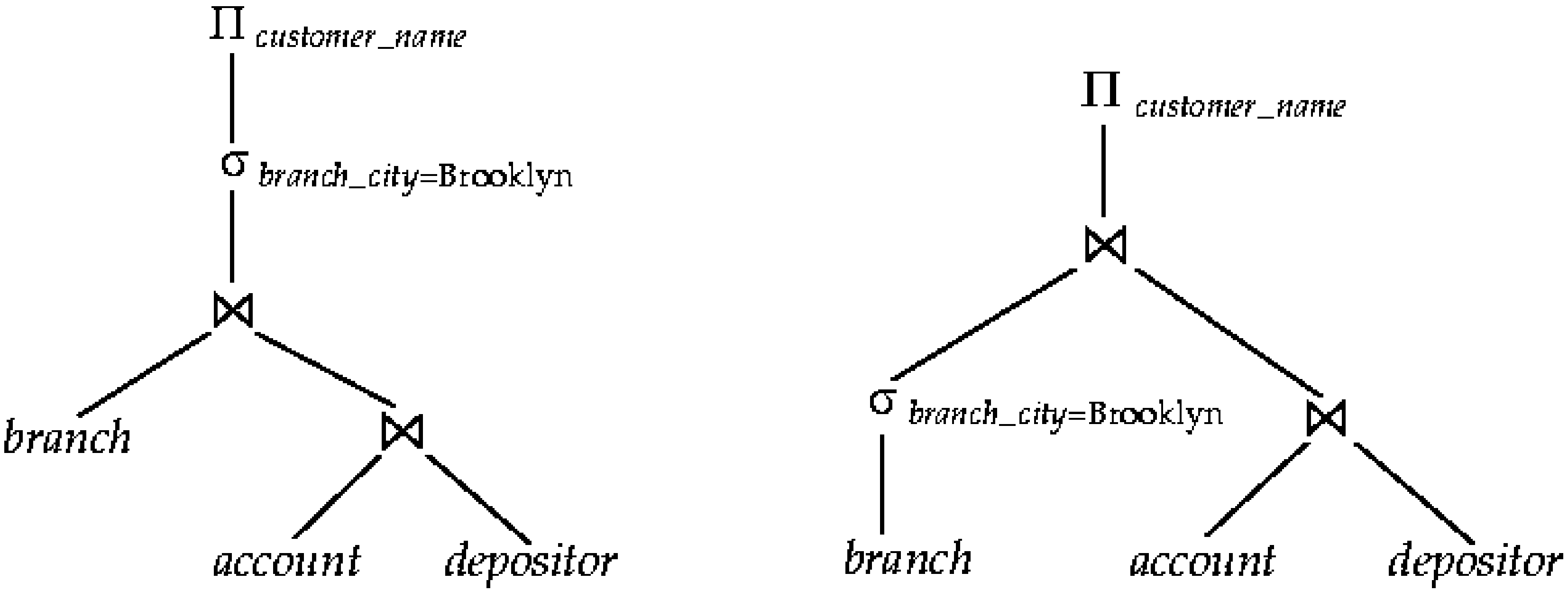
- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block
      - data is read back after being written to ensure that the write was successful

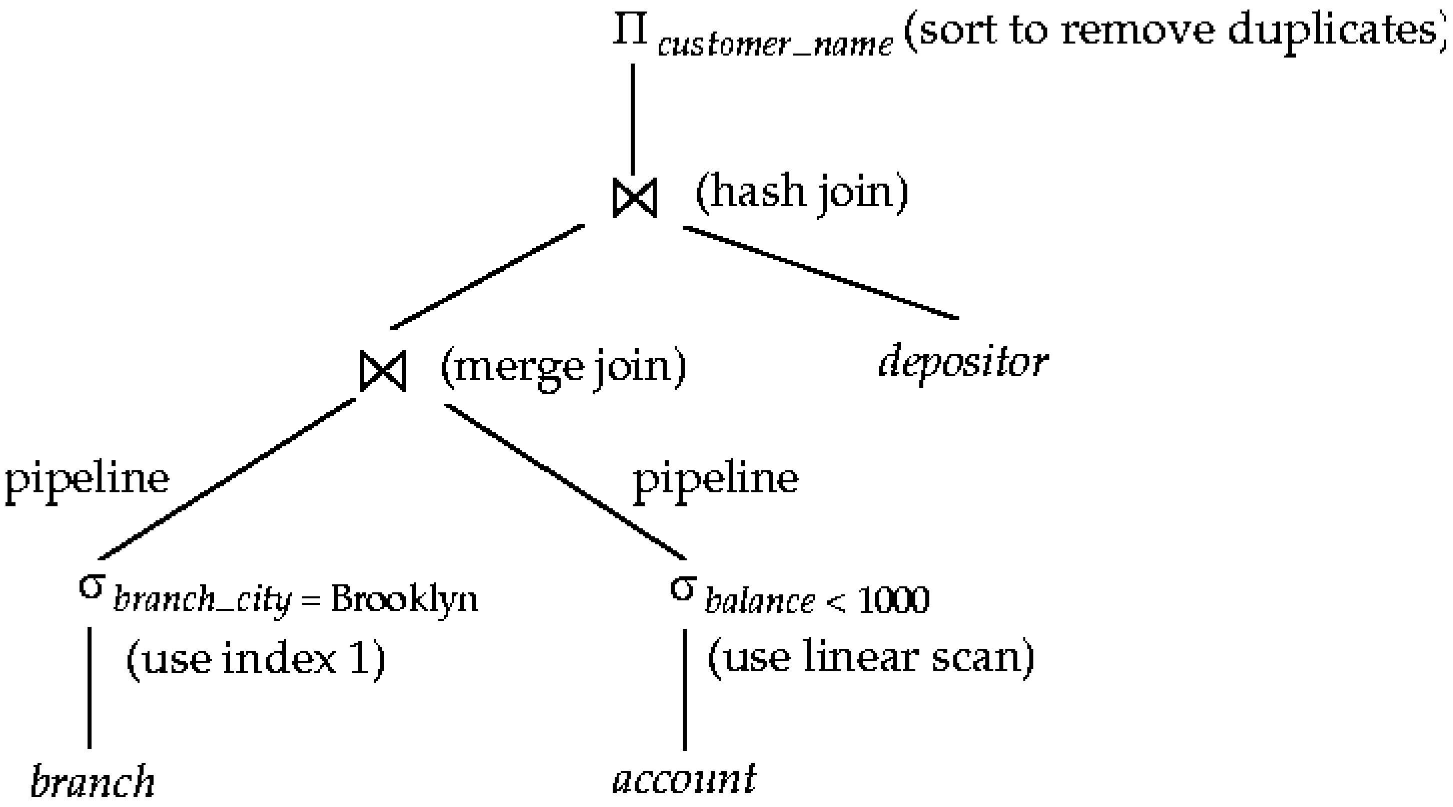
Alternative Query Expressions (ex.)

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation



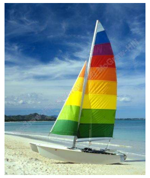
Query Evaluation Plan (example)

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.

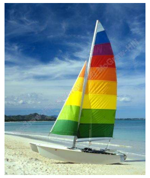


Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimate of plan cost based on:
  - Statistical information about relations. e.g. -
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics



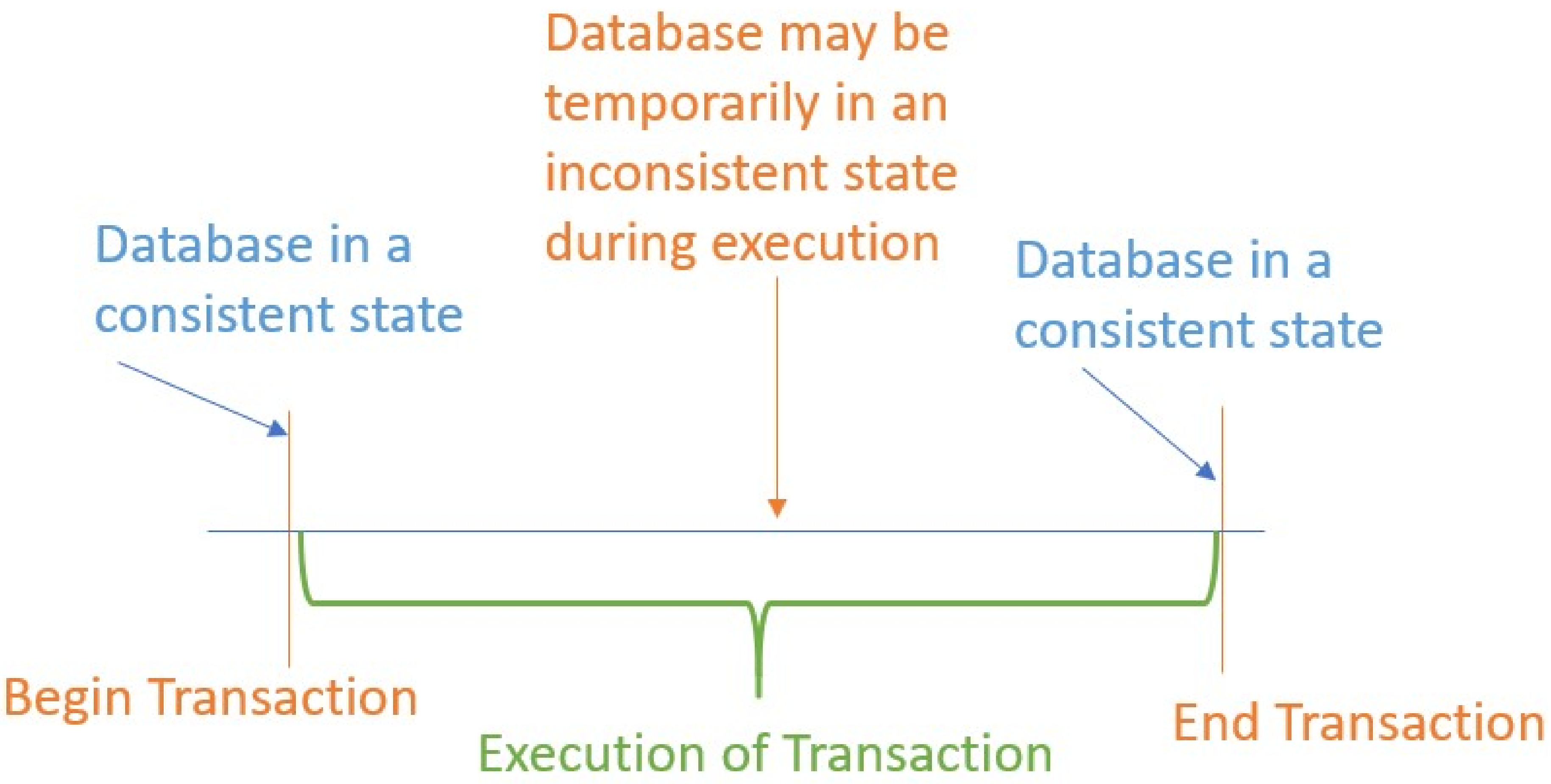
Transactions



Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully (is committed), the database must be consistent.
- After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- Multiple transactions can execute in parallel.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Transaction Execution



Database in a consistent state

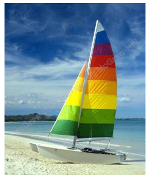
Database may be temporarily in an inconsistent state during execution

Database in a consistent state

Begin Transaction

Execution of Transaction

End Transaction



ACID Properties of Transactions

A **transaction** is a unit of program execution that accesses and possibly updates various data items.To preserve the integrity of data the database system must ensure:

- **Atomicity**. Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency**. Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation**. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

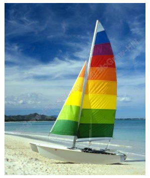
Example of Fund Transfer (Transaction)

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*

$$\mathrm{SUM}(A + B)_{\mathrm{BeforeTrans}}$$

$$= \mathrm{SUM}\ (A + B)_{\mathrm{AfterTrans}}\ ?$$

- **Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.



Example of Fund Transfer (Cont.)

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

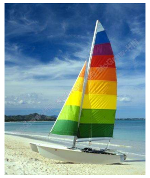
1. **read**(*A*)
2. Display (A)

1. **read**(*A*)
2. Display (A)

- **Isolation requirement** — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).
  - Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
  - However, executing multiple transactions concurrently has significant benefits, as we will see later.

Example of Fund Transfer (Cont.)

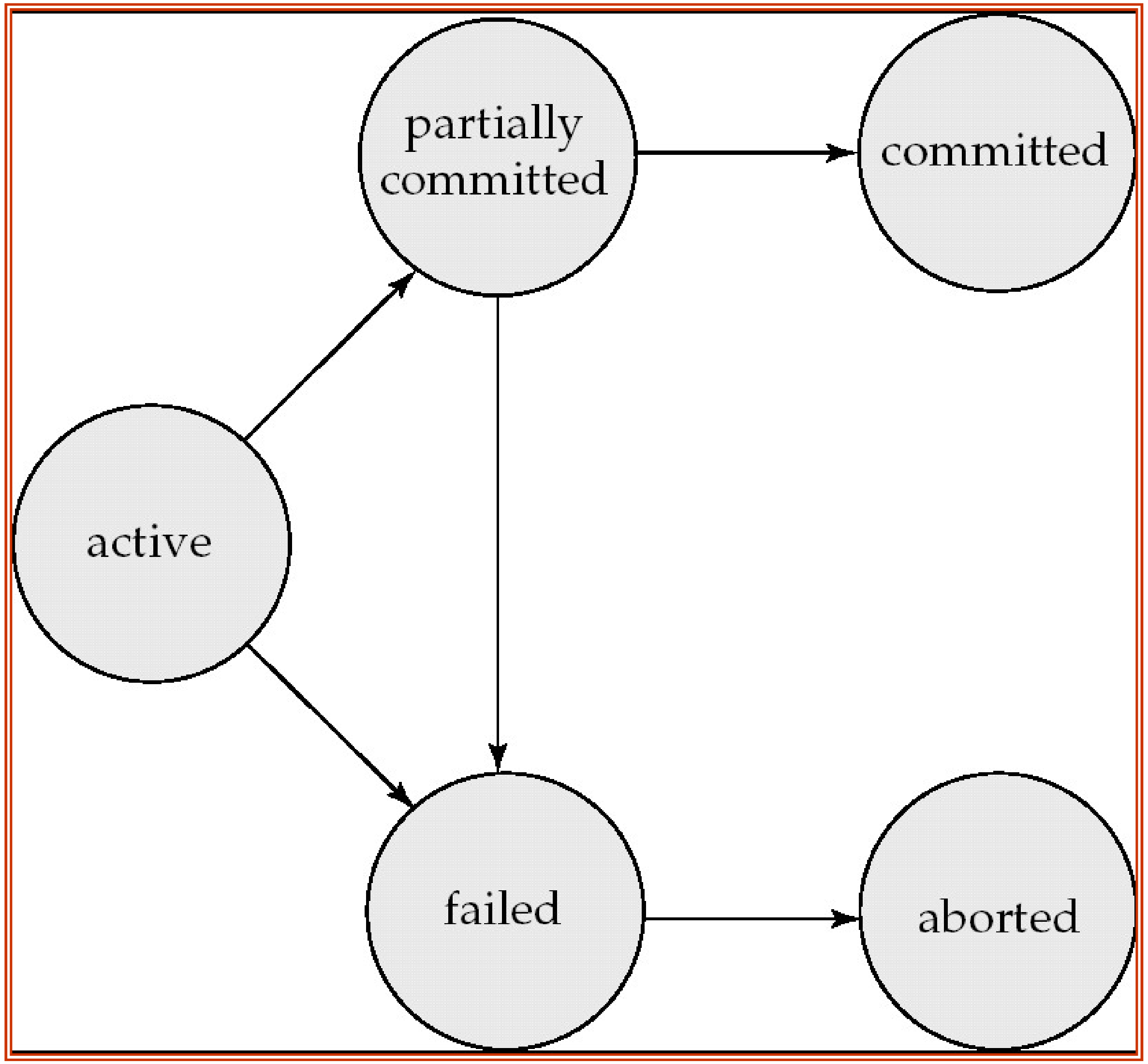
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.



Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction; can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)


partially committed
committed
active
failed
aborted