Boyce-Codd normal form  
“Represent Every Fact Only ONCE”

A relation is in Boyce-Codd normal form (BCNF) if for every FD A → B either

* + B is contained in A (the FD is trivial), or
  + A contains a candidate key of the relation,

In other words: every determinant in a non-trivial dependency is a (super) key.

* The same as 3NF except in 3NF we only worry about non-key Bs
* If there is only one candidate key then 3NF and BCNF are the same
* A relation R with FDs F is said to be in **Boyce-Codd normal form (BCNF)** if for all X→A in F+ then
  + Either A∈X (‘trivial dependency’), or
  + X is a superkey for R
* Intuition: A relation R is in BCNF if the left side of every non-trivial FD contains a key

Consider R=R(city, street&no, zipcode) with FDs:

* + city,street&no → zipcode
  + zipcode → city
  + This is **not** in BCNF, because zipcode is not a superkey for R
  + We potentially duplicate information relating zipcodes and cities
* BankerSchema(brname,cname,bname)
* With FDs
  + bname → brname
  + brname,cname → bname
* Not in BCNF (*Why?*)
* We might decompose to
  + BBSchema(bname,brname)
  + CBrSchema(cname,bname)
* This is in BCNF ☺
* BUT this is **not** dependency-preserving
* A relation R with FDs F is said to be in **Boyce-Codd normal form (BCNF)** if for all X→A in F+ then
  + Either A∈X (‘trivial dependency’), or
* X is a superkey for R

# BCNF Decomposition

The definition of BCNF can be used to directly test if a relationship is in BCNF. If a relation is not in BCNF it can be decomposed to create relations that are in BCNF.

 decomposing relations to BCNF.

Determine BCNF:  
For relation R to be in BCNF, all the functional dependencies (FDs) that hold in R need to satisfy property that the determinants X are all superkeys of R. i.e. if X->Y holds in R, then X must be a superkey of R to be in BCNF.

In your case, it can be shown that the only candidate key (minimal superkey) is ACE. Thus both FDs: A->B and C->D are violating BCNF as both A and C are not superkeys or R.

Decompose R into BCNF form:  
If R is not in BCNF, we decompose R into a set of relations S that are in BCNF.  
This can be accomplished with a very simple algorithm:

Initialize S = {R}

While S has a relation R' that is not in BCNF do:

Pick a FD: X->Y that holds in R' and violates BCNF

Add the relation XY to S

Update R' = R'-Y

Return S

Decomposition into BCNF

Given: relation R with FD’s F § Look among the given FD’s for a BCNF violation X → Y

If any FD following from F violates BCNF, then there will surely be an FD in F itself that violates BCNF §

Compute X +

Not all attributes, or else X is a superkey 5 Decompose R Using X → Y

Replace R by relations with schemas: 1. R1 = X + 2. R2 = R – (X + – X )

Project given FD’s F onto the two new relations

## Characterizing Schedules based on Recoverability:

**Transaction schedule or history**:

* + When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history).

A **schedule** (or **history**) S of n transactions T1, T2, …, Tn:

* + It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.
  + Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

Schedules classified on recoverability:

**Recoverable schedule**:

* + One where no transaction needs to be rolled back.
  + A schedule S is recoverable if no transaction T in S commits until all transactions T’ that have written an item that T reads have committed.

**Cascadeless schedule**:

* + One where every transaction reads only the items that are written by committed transactions.

**Schedules requiring cascaded rollback**:

* + A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.

**Strict Schedules**:

A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

Characterizing Schedules based on Serializability:

# Serial schedule:

A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.

Otherwise, the schedule is called nonserial schedule.

# Serializable schedule:

A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

Characterizing Schedules based on Serializability:

Result equivalent:

* + Two schedules are called result equivalent if they produce the same final state of the database.

Conflict equivalent:

* + Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

Conflict serializable:

* + A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S’.

Being serializable is not the same as being serial

Being serializable implies that the schedule is a correct schedule.

* + It will leave the database in a consistent state.
  + The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

Serializability is hard to check.

* + Interleaving of operations occurs in an operating system through some scheduler
  + Difficult to determine beforehand how the operations in a schedule will be interleaved.

Practical approach:

Come up with methods (protocols) to ensure serializability.

It’s not possible to determine when a schedule begins and when it ends.

* + Hence, we reduce the problem of checking the whole schedule to checking only a **committed** **project** of the schedule (i.e. operations from only the committed transactions.)

Current approach used in most DBMSs:

* + Use of locks with two phase locking

The premise behind view equivalence:

* + As long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction must produce the same results.
  + “**The view**”: the read operations are said to see *the same view* in both schedules.

**Relationship between** **view and conflict equivalence**:

* + The two are same under **constrained write assumption** which assumes that if T writes X, it is constrained by the value of X it read; i.e., new X = f(old X)
  + Conflict serializability is **stricter** than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serializable.
  + Any conflict serializable schedule is also view serializable, but not vice versa.
  + Consider the following schedule of three transactions

T1: r1(X), w1(X); T2: w2(X); and T3: w3(X):

* + Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;
  + In Sa, the operations w2(X) and w3(X) are blind writes, since T1 and T3 do not read the value of X.
  + Sa is view serializable, since it is view equivalent to the serial schedule T1, T2, T3.
  + However, Sa is not conflict serializable, since it is not conflict equivalent to any serial schedule.

**Testing for conflict serializability:**

**Algorithm**

* + Looks at only read\_Item (X) and write\_Item (X) operations
  + Constructs a precedence graph (serialization graph) - a graph with directed edges
  + An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
  + The schedule is serializable if and only if the precedence graph has no cycles.

## Transaction Support in SQL

A **single** SQL statement is always considered to be **atomic**.

* + Either the statement completes execution without error or it fails and leaves the database unchanged.

With SQL, there is no explicit Begin Transaction statement.

* + Transaction initiation is done implicitly when particular SQL statements are encountered.

Every transaction must have an explicit end statement, which is either a COMMIT or ROLLBACK.

Characteristics specified by a SET TRANSACTION statement in SQL2:

**Access mode**:

READ ONLY or READ WRITE.

* + - The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.

**Diagnostic size** n, specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area.

Characteristics specified by a SET TRANSACTION statement in SQL2 (contd.):

**Isolation level** <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.

* + With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability.
  + However, if any transaction executes at a lower level, then serializability may be violated.

Potential problem with lower isolation levels:

**Dirty Read**:

* + Reading a value that was written by a transaction which failed.

**Nonrepeatable Read**:

* + Allowing another transaction to write a new value between multiple reads of one transaction.
  + A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.

Consider that T1 reads the employee salary for Smith. Next, T2 updates the salary for Smith. If T1 reads Smith's salary again, then it will see a different value for Smith's salary.

Phantoms:

New rows being read using the same read with a condition.

* + - * A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.
      * Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1.
      * If T1 is repeated, then T1 will see a row that previously did not exist, called a phantom.