VT Sistem Gerçekleme Ders Notları- #7

Remote: Kullanıcıdan gelen JDBC isteklerini karşılar.
Planner: SQL ifadesi için işleme planı oluşturur ve karşılık gelen ilşkisel cebir ifadesini oluşturur.
Parse: SQL ifadesindeki tablo, nitelik ve ifadeleri ayrıştırır.
Query: Algebra ile ifade edilen sorguları gerçekler.
Metadata: Tablolara ait katalog bilgilerini organize eder.
Record: disk sayfalarına yazma/okumayı kayıt seviyesinde gerçekler.
Transaction&Recovery: Eşzamanlılık için gerekli olan disk sayfa erişimi kısıtlamalarını organize eder ve veri kurtarma için kayıt_defteri (*log*) dosyalarına bilgi girer.
Buffer: En sık/son erişilen disk sayfalarını ana hafıza tampon bölgede tutmak için gerekli işlemleri yapar.
Log: Kayıt_defterine bilgi yazılmasını ve taranması işlemlerini düzenler.
File: Dosya blokları ile ana hafıza sayfaları arasında bilgi transferini organize eder.

Hareket (*transaction*) yönetimi (HY)

Temel Kavramlar

Konu başlıkları

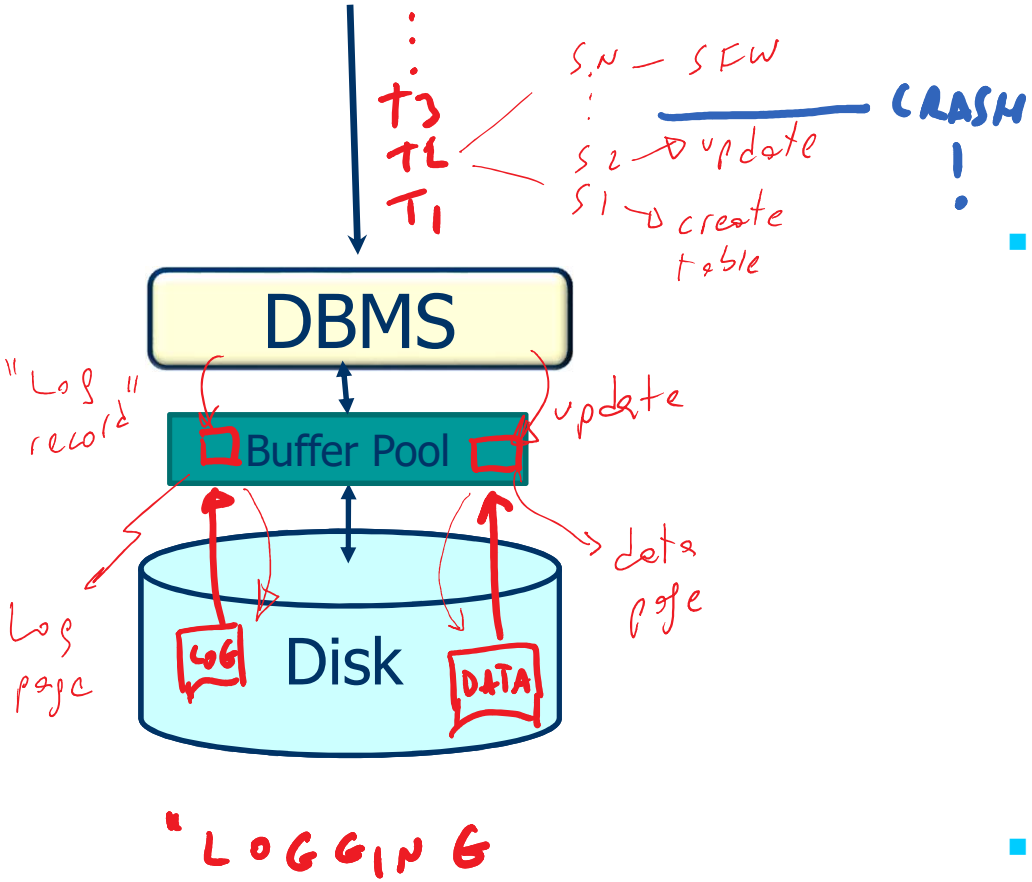
- Hareketin tanımlanması
  - ACID
  - Sistem hatası, Kurtarma ne demek?
  - Eşzamanlılık ne demek? (Tarihçe, Plan tanımları)
    - Nitelik seviyesinde, Kayıt seviyesinde, tablo seviyesinde eşzamanlı erişim örnekleri
- HY'nin önemi nedir? Olmazsa olur mu?
- Varsayılan(*default*) hareket yönetimi: (Açıktan HY olmaması: *Autocommit(true)*)
- Eşzamanlılık Kontrolü: Yalıtım seviyeleri
  - Yalıtım Hataları *(Kirli Okuma, yanlış toplama, yazma kaybolması, hayalet kayıtlar)*
  - Doğru yalıtım seviyesi tespiti

Hareket (*transaction, tx*)

- **Hareket**, birden çok veri tabanı işleminin, **tek bir işlem gibi** çalıştığı bir programdır. İstemcinin VTYS'den alacağı her servis, bir "**hareket"** kapsamında gerçekleştirilir.
- 2 esas :
  - **Veri bütünlüğü :**Hareket, VT'nı tutarlı bir durumdan başka bir tutarlı bir duruma getirir.
  - **Performans:** hareketlerin eşzamanlı çalışması gerekir. Hareketler "veri bütünlüğü" ne zarar vermeden diğer hareketlerle eşzamanlı olarak çalışır .
- **VTYS hareket yönetimi (HY),** çok sayıda istemciden gelen hareketleri, eşzamanlı çalıştırarak, VT bütünlüğünü (*integrity*) sağlayacak şekilde sonlanmasını sağlamakla yükümlüdür. Bunun için de hareketin, ACID özelliklerini sağlaması gerekir.
  - Atomicity (atomik, bölünemezlik..): "ya hep ya hiç!"
  - Consistency (tutarlılık): "vt'nını tutarlı bırak!"
  - Isolation (yalıtım): "sanki bir tek kendisi var!"
  - Durability (süreklilik): "onaydan sonra artık sürekli var!"

- Bunlar VTYS'de 2 farklı modül ile sağlanır: **Veri Kurtarma ve Eşzamanlılık**
- **Kurtarma modulü:**
  - Atomik ve Sürekliliği, COMMIT ve ROLLBACK ile sağlar.
  - Log kayıtlarının oluşturulması ve diske yazılması
  - Hareketin gerektiğinde Geriye dönüşünü (*rollback*) sağlamak
  - Sistemin çökmesinde, veri tabanını kurtarma (*recovery*)
- **Eşzamanlılık modulü:**
  - Tutarlılık ve Yalıtım, LOCK ve benzeri yöntemler ile sağlanır.
  - Birden çok hareketin, aynı anda paralel olarak çalışmasının (sanki seri olarak çalışıp sonlanmış gibi) doğruluğunu sağlamak.

Kurtarma Yönetimi Ne demek? Ne ihtiyaç var?



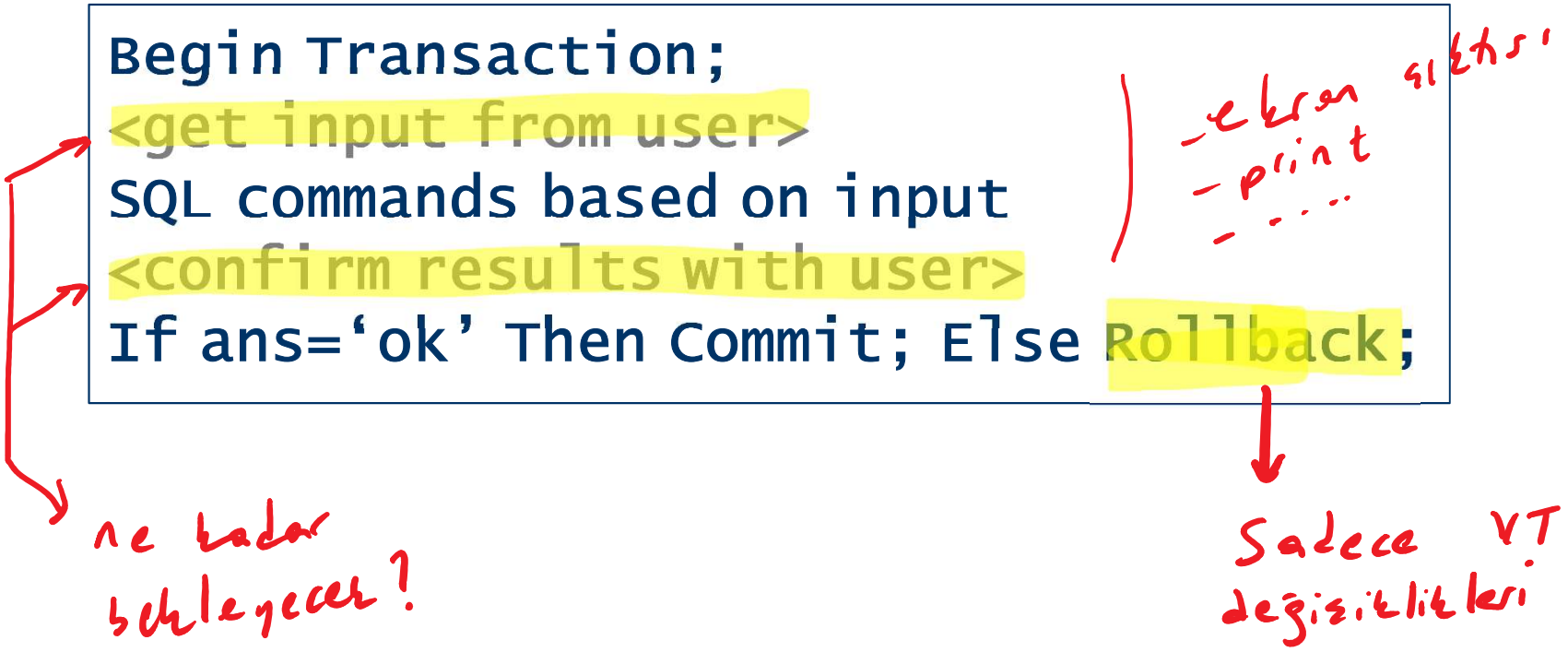
- SİSTEM KURTARMA, bir hareketin herhangi bir **tipik hatada** «geri sarılamaması **(A ihlali)** veya bütün yaptığı değişikliklerin diske kaydedilememesi **(D ihlali)**» durumlarına mani olacak önlemlerin alınmasını sağlar. *(WAL, Partially commit gibi)*

- Tipik hatalar: *(log veya shadow yöntemleri ile kurtarılabilenler)*
  - Sistem kilitlenmesi *(örn. Ana hafiza hatalari..)*
  - Hareket veya sistem hatası *(örn. hareket içi mantıksal hatalar, 0 ile bölme ..)*
  - Yerel hatalar veya hareketin tespit ettiği özel durumlar *(hareketin ulaşmak istedigi bilginin bulunamaması veya yetersizliği)*
  - Eşzamanlılık kontrol mekanizasının zorlaması *(deadlock, serilenebilirlik ihlali gibi..)*

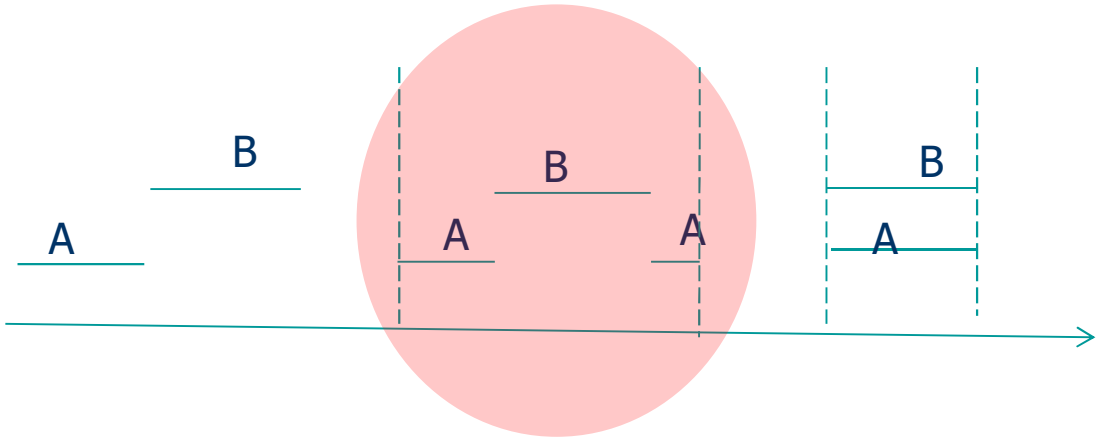
- Seyrek hatalar: *(ancak VT arşiv/backup ile kurtarılabilenler)*
  - Disk hataları *(disk kolu hataları)*
  - Fiziksel veya dışsal felaketler.. (yangın, sel ..ve diğerleri)

Hareket Geri sarılması (*rollback=abort*)

- Hareketin yaptığı (kısmi) değişiklikleri geriye alması. (*hareketin ilk başındaki tutarlı duruma geri dönmesi*)
- Sistem veya uygulama (istemci) tarafından çalıştırılabilir.



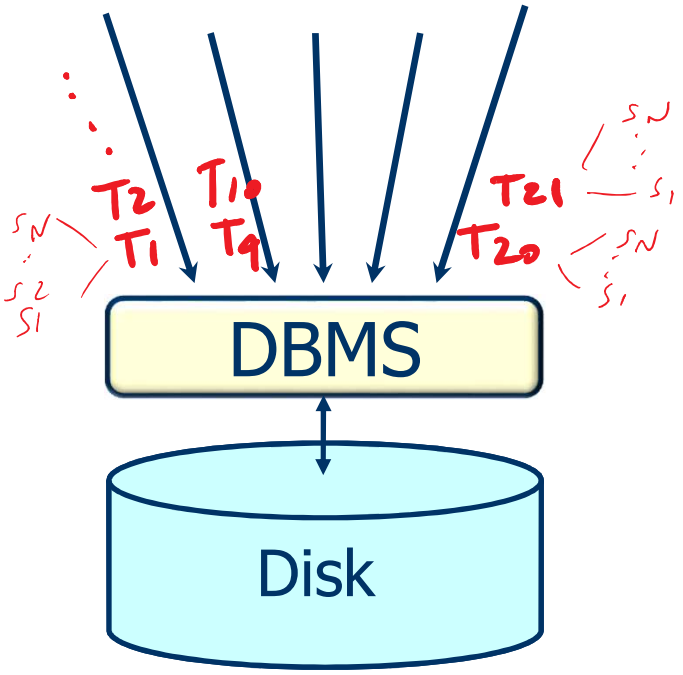
Eşzamanlılık Ne Demek? Ne ihtiyaç var?



Daha çok eşzamanlılık (daha yüksek performans)

- Yüksek kaynak (cpu,disk) kullanım verimliliği (better resource utilization)
- Sistem tx hizmet kapasitesi (high system througput)
- Ortalama sistem cevap süresinin azalması (better ave. response time)

Yalıtım (*isolation*)



"SERIALIZABILITY".

seri bir plan:

$\langle T1, T9, T_{20}, T_{10}, T_2, T21 \rangle$

Bazı Tanımlar

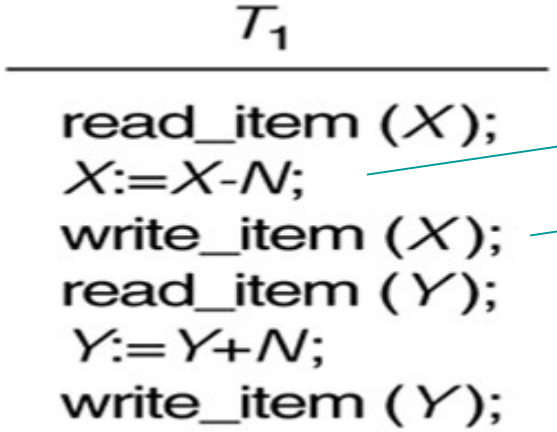
Veri tabanı eylemi (db action):bir bloğun diskten okunması-yazılması

R(X): X bloğunun diskten tampona yazılması ve okunması

W(X): X bloğunun diskten tampona yazılması, değişiklik yapılması (lokal+tampon) ve sonra hemen veya bir zaman sonra diske geri yazılması

Tarihçe (*History*): bir tx'nın vt dosyaları üzerindeki erişim dizisi. Örneğin:
tx: R(student.tbl,0); R(student.tbl,0); W(student.tbl,0);

Plan (*Schedule*): vt üzerinde işlem yapan **bütün tx'ların (*öngörülemez halde*)** içiçe girmiş tarihçeleri.



X lokalde: daha kimse görmüyor.

Tampona yaz : başkaları «görebilir». «Hemen» veya «bir zaman sonra» diske yazılacak.

Hangisi daha önce diske yazılıyor: Yeni X, yeni Y ?

Örnek Plan (*schedule*)

(a)

T_1
read_item (X);
X:=X-N;
write_item (X);
read_item (Y);
Y:=Y+N;
write_item (Y);

(b)

T_2
read_item (X);
X:=X+M;
write_item (X);

- T1: $R_1(X)$,$W_1(X)$, $R_1(Y)$, $W_1(Y)$
- T2: $R_2(X)$,$W_2(X)$
- Farklı plan sayısı toplam (4+2)! / (4! * 2!) = 6*5*4*3*2*1/ 4*3*2*1*2*1 = 15.
- Farklı SERİ planların sayısı : 2!=2
- PLAN 1: $r_1(X)$; $w_1(X)$; $r_1(Y)$; $r_2(X)$; $w_1(Y)$; $w_2(X)$;
- PLAN 2: $r_2(X)$; $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;
- ……
- PLAN 15: ...

Farklı tanesellikte eşzamanlılık örnekleri

COLLEGE (cName, state, enrollment)
STUDENT (sID, sName, GPA, HSsize)
APPLY (sID, cName, major, decision)

enrollment: üniversite'nin kapasitesi
HSsize: high school size,mezun olduğu lise kapasitesi
öğrenci kabul edilince (decision) ='A', red alınca dec = 'R' değerini alıyor.

Nitelik seviyesinde eşzamanlı erişim

```
Update College Set enrollment = enrollment + 1000
Where cName = 'Stanford'
```

```
Update College Set enrollment = enrollment + 1500
Where cName = 'Stanford'
```

Kayıt seviyesinde eşzamanlı erişim

```
Update Apply Set major = 'CS' Where sID = 123
```

```
Update Apply Set decision = 'Y' Where sID = 123
```

Farklı tanesellikte eşzamanlılık örnekleri

COLLEGE (cName, state, enrollment) — *enrollment: üniversite'nin kapasitesi*
STUDENT (sID, sName, GPA, HSsize) — *HSsize: high school size,mezun olduğu lise kapasitesi*
APPLY (sID, cName, major, decision) — *öğrenci kabul edilince dec(ision) ='A', red alınca dec = 'R' değerini alıyor.*

Tablo seviyesinde eşzamanlı erişim

```
Update Apply Set decision = 'Y'
Where sID In (Select sID From Student Where GPA > 3.9)
```

```
Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500
```

Çok-operasyonlu eşzamanlı erişim

```
Insert Into Archive
  Select * From Apply Where decision = 'N';
Delete From Apply Where decision = 'N';
```

```
Select Count(*) From Apply;
Select Count(*) From Archive;
```

HY'nin önemi nedir? Olmazsa olur mu?

```
public void reserveSeat(Connection conn, int custId,
                        int flightId) throws SQLException {

   Statement stmt = conn.createStatement();
   String s;

// Step 1: Get availability and price
   s = "select NumAvailable, Price from SEATS "
     + "where FlightId = " + flightId;
   ResultSet rs = stmt.executeQuery(s);
   if (!rs.next()) {
      System.out.println("Flight doesn't exist");
      return;
   }
   int numAvailable = rs.getInt("NumAvailable");          R(A)
   int price = rs.getInt("Price");
   rs.close();                                  R(B)

   if (numAvailable == 0) {
      System.out.println("Flight is full");
      return;
   }

// Step 2: Update availability
   int numseats = numAvailable - 1;                       W(A)
   s = "update SEATS set NumAvailable = " + numseats
    + " where FlightId = " + flightId;
   stmt.executeUpdate(s);

// Step 3: Get and update customer balance
   s = "select BalanceDue from CUST "
     + "where CustID = " + custId;                R(C)
   rs = stmt.executeQuery(s);
   int newBalance = rs.getInt("BalanceDue") + price;
   rs.close();

   s = "update CUST set BalanceDue = " + newBalance
    + " where CustId = " + custId;
   stmt.executeUpdate(s);                                 W(C)
}
```

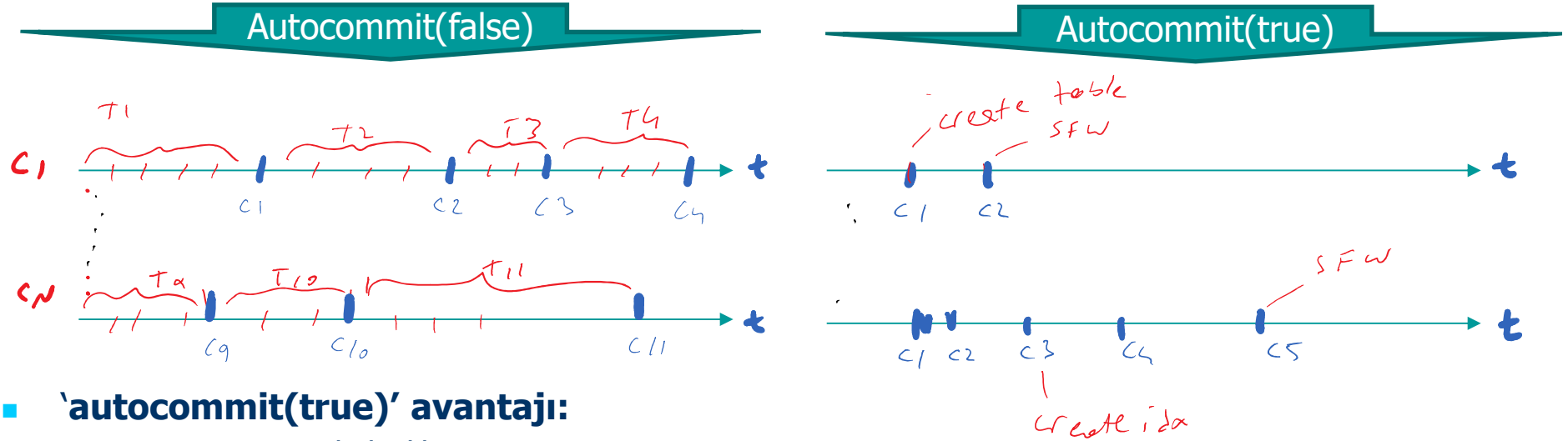
Figure 14-1
JDBC code to reserve a seat on a flight

- Bir uçak rezervasyon sistemine ait veri tabanı şeması: *(KOLTUKLAR tablasunda, her uçuş için boş koltuk sayısı ve koltuk ücreti tutuluyor. MÜŞTERİ tablosunda ise, her müşterinin ödemesi gereken miktar bilgisi saklanmaktadır.)*
  - **KOLTUKLAR(UçuşNo, BoşKoltukSayısı, Ücret)**
  - **MÜŞTERİ(MüşteriNo, Borc)**
- Aşağıdaki 3 farklı senaryoyu inceleyin:

1. **2 farklı müşteri** bu programı çalıştırıyorlar. **1. müşteri** 1. adım sonunda iş kesme (*interrupt*) ile beklemeye başladı. **2. müşteri** programı çalıştırıp sonlandırdı. **1. müşteri** kaldığı yerden devam edip programı sonlandırdı. (**izolasyon ihlal problemi**)
2. **Sadece 1 müşteri** programı çalıştırıyor. Programın 2. adım sonlandıktan sonra, veri tabanı yönetim sistemi çöküyor.(**atomiklik ihlali)**
3. **Sadece 1 müşteri** programı çalıştırıyor. Programın 3. adımı sonlandıktan sonra, veri tabanı yönetim sistemi çöküyor. **(süreklilik ihlali)**

Varsayılan(*default*) hareket yönetimi :*autocommit (true)*

- **autocommit(true) :** hareket kontrolünün kullanıcıdan saklanmasıdır. «Açıktan» HY olmaması halidir. **Her SQL komutu bir hareket gibi işlenir.**
  - *executeupdate(…)* yenileme gerçekleşince sonlanır.
  - *executequery(…)* ise, kullanıcı sorgu sonuç setini kapatınca, *rs.close() veya yeni bir sorgu çalışmaya başlayınca* sonlanır.



- **'autocommit(true)' avantajı:**
  - Programcı için kolaylık.
  - Verimli kaynak kullanımı. *Hareket, sonlanıncaya kadar sahip olduğu kaynakları (kilitler, tablolar..) tutar; bu sistemin başka hareketlere olan hizmetini yavaşlatır. Onun için, Kısa hareketler tercih edilir. Autocommit(true) da 1 sorgu → 1 hareket.*
- **'autocommit (true)' nun** yetersiz kalması :
  1) **Birden çok işlemin (*statement*)** aktif olması.
  2) Veri tabanında **birden çok değişikliğin bütünlük içinde** (atomik) olması
  3) *Bulk loading* denilen, bilginin dış ortamdan çok sayıda insert işlemi ile vt'ye aktarılması.

Varsayılan(*default*) *autocommit (true)* yetersizliği

```
DataSource ds = ...
Connection conn = ds.getConnection();
Statement stmt1 = conn.createStatement();
Statement stmt2 = conn.createStatement();
String qry = "select * from COURSE";
ResultSet rs = stmt1.executeQuery(qry);
while (rs.next()) {
   String title = rs.getString("Title");
   boolean goodCourse = getUserDecision(title);
   if (!goodCourse) {
      int id = rs.getInt("CId");
      stmt2.executeUpdate("delete from COURSE "
                        + "where CId =" + id);
   }
}
rs.close();
```

1)

```
DataSource ds = ...
Connection conn = ds.getConnection();
Statement stmt = conn.createStatement();
String cmd1 = "update SECTION set Prof= 'brando' "
            + "where SectId = 43";
String cmd2 = "update SECTION set Prof= 'einstein' "
  ..        + "where SectId = 53";

stmt.executeUpdate(cmd1);
// suppose that the server crashes at this point
stmt.executeUpdate(cmd2);
```

2)

- Yukarıdaki 1) ve 2) durumlarına karşılık gelen örnek senaryoların doğru çalışması için kullanıcının, **Connection** sınıfında hareketin **autocommit** modunu **false** yapması ve bu yüzden program içerisinde **commit** ve **rollback** yapması gerekir...JDBC Connection sınıfının bununla ilgili kısmı:

```
public void setAutoCommit(boolean ac) throws SQLException;
public void commit()                  throws SQLException;
public void rollback()                throws SQLException;
```

Figure 8-11
The *Connection* API for handling transactions explicitly

Varsayılan(*default*) *autocommit (true)* yetersizliği

Önceki sunuda 1) numarali programın doğru hali

```
DataSource ds = ...
Connection conn = ds.getConnection();
Statement stmt1 = conn.createStatement();
Statement stmt2 = conn.createStatement();
String qry = "select * from COURSE";
ResultSet rs = stmt1.executeQuery(qry);
while (rs.next()) {
   String title = rs.getString("Title");
   boolean goodCourse = getUserDecision(title);
   if (!goodCourse) {
      int id = rs.getInt("CId");
      stmt2.executeUpdate("delete from COURSE "
                          + "where CId =" + id);
   }
}
rs.close();
```

1)

```
Connection conn = null;
try {
   DataSource ds = ...
   Connection conn = ds.getConnection();
   conn.setAutoCommit(false);

   Statement stmt = conn.createStatement();
   String qry = "select * from COURSE";
   ResultSet rs = stmt.executeQuery(qry);
   while (rs.next()) {
      String title = rs.getString("Title");
      boolean goodCourse = getUserDecision(title);
      if (!goodCourse) {
         int id = rs.getInt("CId");
         String cmd = "delete from COURSE"
                      + "where CId =" + id;
         stmt.executeUpdate(cmd);
      }
   }
   rs.close();
   conn.commit();
}
catch (SQLException e) {
   try {
      e.printStackTrace();
      conn.rollback();
   }
   catch (SQLException e2) {
      System.out.println("Could not roll back");
   }
}
finally {
   try {
      if (conn != null)
          conn.close();
   }
   catch (Exception e) {
```

Eşzamanlılık Kontrolünde Yalıtım seviyeleri

Eşzamanlı çalışma : hızlı fakat veri bütünlüğü riski yüksek

seri olarak çalışmaya denk : her zaman doğru fakat yavaş ...

Kontrollü eşzamanlı çalışma

Yalıtım Seviyeleri

Read Uncommitted
-- dirty read/write
-- nonrepeatable read, incorrect summary
-- lost update,phantom

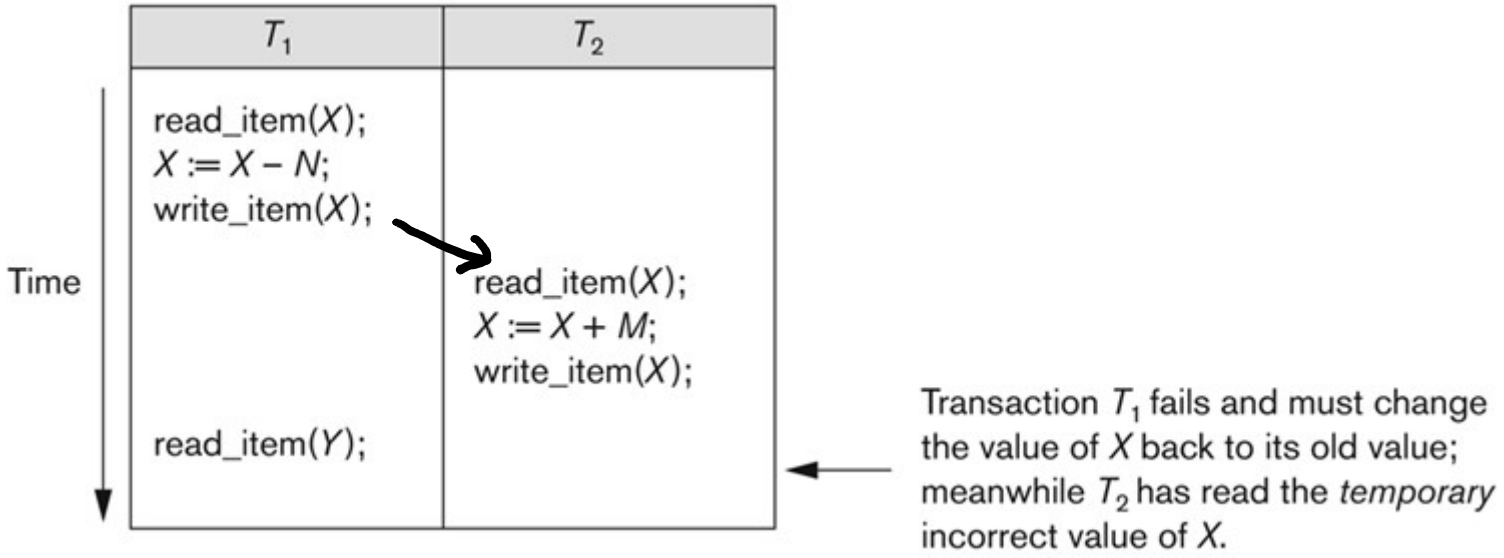
Read Committed
-- nonrepeatable read,incorrect summary
-- lost update ,phantom

Repeatable Read
--lost update, phantom

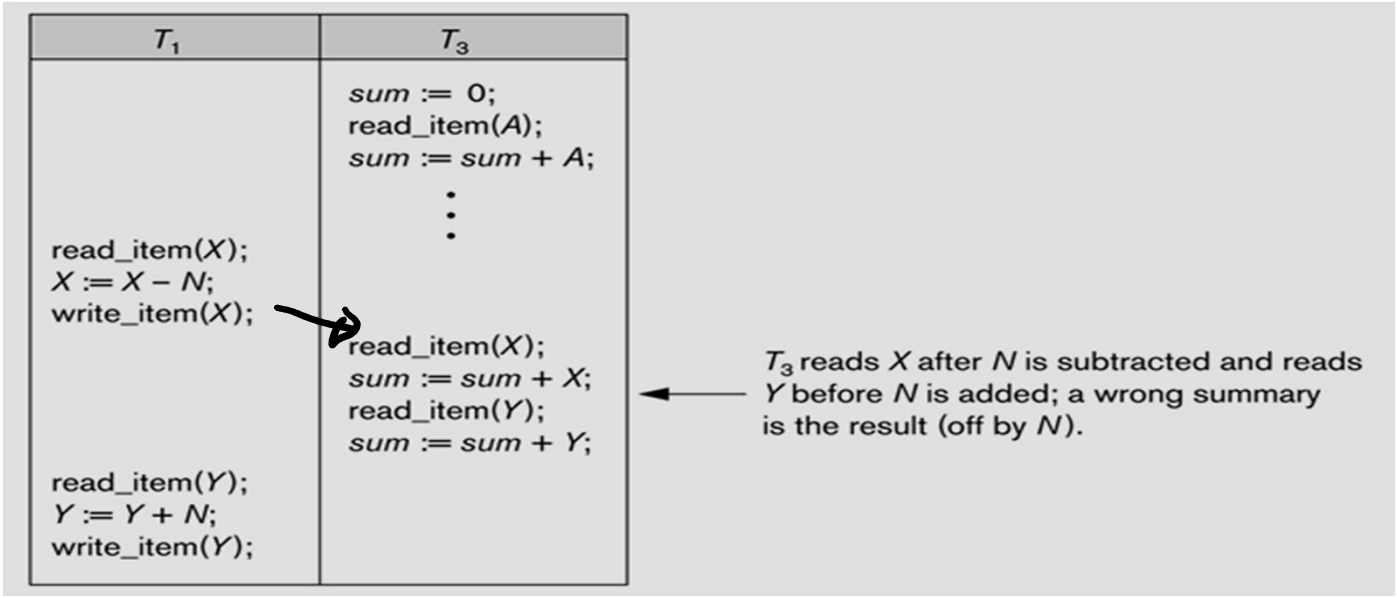
Serilenebilir

Hareket, vt'nı tutarlı bir durumdan başka bir tutarlı duruma taşıyor. Bu geçişte, vt tutarsız durumlardan geçebilir. Bu geçişteki tutarsız durumların yalıtılması; yani diğer hareketlere yansıtılmaması gerekiyor..Yansılıtılmaması miktarı **yalıtım seviyesinin** set edilmesi ile oluyor.

Yalıtım Hataları: Kirli okuma –(*dirty read , WR çelişkisi)*



Yalıtım Hataları: Yanlış toplam- *incorrect summary, WR çelişkisi*



Yandaki H1 hareketi, her öğretmenin kaç ders verdiğini hesaplayan H2 hareketi ile beraber çalışsın..

```
Connection conn = ds.getConnection();
 conn.setAutoCommit(false);
 Statement stmt = conn.createStatement();
 String cmd1 = "update SECTION set Prof= 'brando' "
             + "where SectId = 43";
 String cmd2 = "update SECTION set Prof= 'einstein' "
   ..        + "where SectId = 53";

 stmt.executeUpdate(cmd1);

 stmt.executeUpdate(cmd2);
```

Yalıtım Hataları: Yazma kaybolması–*lost update, WW çeliskisi*

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
String qry = "select MealPlanBalance "
           + "from STUDENT where SId = 1";
ResultSet rs = stmt.executeQuery(qry);

rs.next();
int balance = rs.getInt("MealPlanBalance");
rs.close();

int newbalance = balance - 10;
if (newbalance < 0)
    throw new NoFoodAllowedException();

String cmd = "update STUDENT "
           + "set MealPlanBalance = " + newbalance
           + " where SId = 1";
stmt.executeUpdate(cmd);

conn.commit();
```

(a) Transaction T_1 decrements the meal plan balance

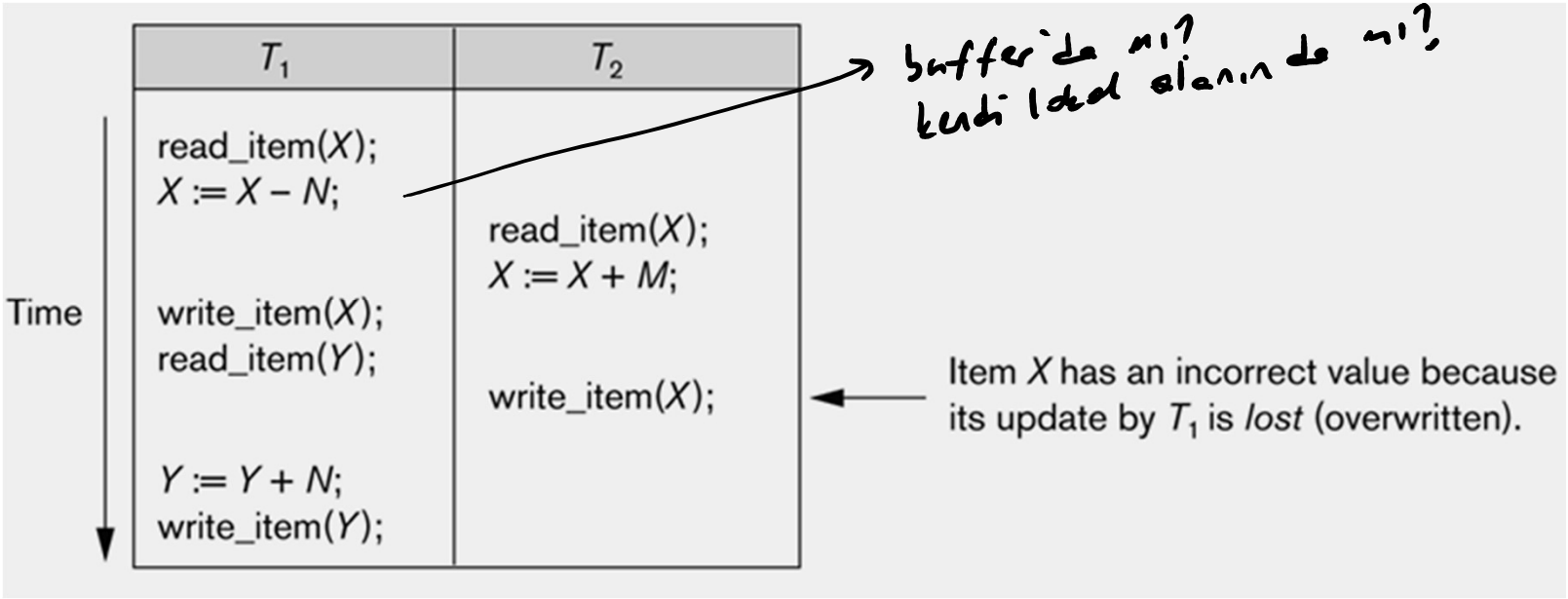
```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
String cmd = "update STUDENT "
           + "set MealPlanBalance = MealPlanBalance + 1000"
           + " where SId = 1";
stmt.executeUpdate(cmd);

conn.commit();
```

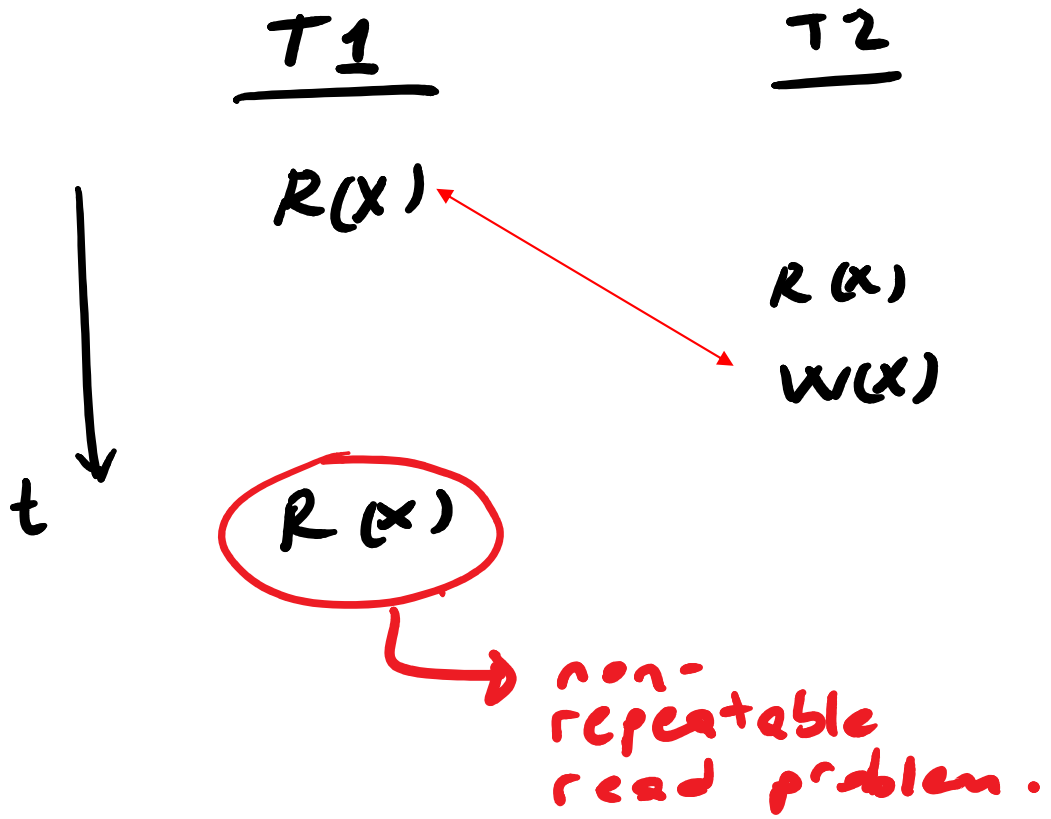
(b) Transaction T_2 increments the meal plan balance

Soldaki T1 hareketi, **balance** değerini okumasıyla, değiştirmesi arasında **balance'ın** değişmediğini varsayıyor…Oysa bu bu ancak *repetable read* yalıtımında mumkun…

Yalıtım Hataları: Yazma kaybolması–*lost update, WW çeliskisi*



Yalıtım Hataları: tekrar okuyamama hatası- nonrepeatable read, RW conflict



Bir hareketin okuduğu bir değeri, kendisi değiştirmediği halde, sonradan farklı okunması problemidir.

Yalıtım Hataları: Hayalet kayıtlar-*phantom problem*

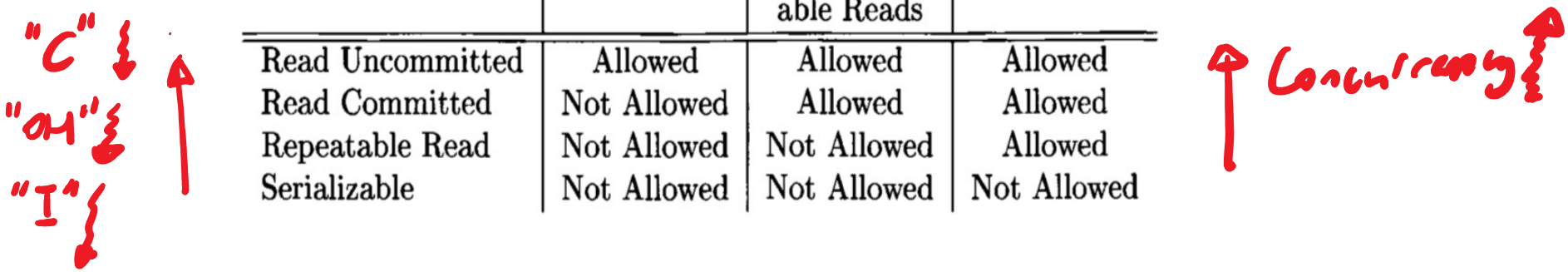
```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
String qry = "select count(SId) as HowMany "
     + "from STUDENT "
     + "where GradYear >= extract(year, current_date)";
ResultSet rs = stmt.executeQuery(qry);
rs.next();
int count = rs.getInt("HowMany");
rs.close();

int rebate = 100000 / count;
String cmd = "update STUDENT "
     + "set MealPlanBalance = MealPlanBalance + "
                            + rebate
     + " where GradYear >= extract(year, current_date)";
stmt.executeUpdate(cmd);
conn.commit();
```

- Yukarıdaki H1 hareketi, yeni öğrenciler ekleyen başka bir H2 hareketi ile beraber çalışsın…

Yalıtım seviyeleri ve hataları

Isolation Level	Dirty Reads	Nonrepeat-able Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed



- **read committed** …Yani, bir hareket, başka bir hareket tarafından **değiştirilmiş ve commit edilmemiş** bir bilgiye erişemez.. Bu, **dirty read/write** problemlerini çözer..
- Bir hareket, okuduğu bir değerin başkaları tarafından değiştirilmesine maruz kalıyorsa bu bir **non-repeatable read** problemidir. Bunu çözen yalıtım seviyesi **repeatable read** dir.
- H1 hareketi tabloda seçtiği kayıtlar üzerinde bir işlem gerçekleştiriyorsa; ve bu arada diğer bir hareket H1'in seçim kriterine uygun yeni kayıtlar ekliyosa bu bir **phantom** problemidir. Bunu çözen yalıtım ancak **serilenebilir çalışmadır**.
- **Yazma çekişmeleri:** Bu çekişmeler farklı formlarda olabilir: Kirli yazma, Lost update, Skewed write..bunların yalıtımının Serilenebilir yalıtım olmadan önleyen bazı teknikler var. Fakat her durum için çok sağlam olmayabilir. (Yukarıdaki tabloda bunlar gözükmüyor, eksik/eski bir tablo)
- **Serilenebilir plan:** Bir grup hareketin «eşzamanlı» çalışmasının neticesi, bu hareketlerin herhangi bir SERI PLAN ile çalışıp sonlanmasına DENK geliyorsa; bu eşzamanlı çalışma «serilenebilir»dir. **Ancak bütün hareketler «Serializable» yalıtımda ise GLOBAL serilenebilirlik sağlanır.** Bu yalıtımı sağlayan kilit protokolü ise STRICT 2PL'dir. (*bir sonraki ders konusu*)

Doğru Yalıtım seviyesinin tespiti

- Hareketin içerdiği veri tabanı eylemlerine göre ve/veya veri bütünlüğü noktasındaki beklentilere göre izolasyon seviyesi belirlenmelidir.
  - Mesela, hareket başında SET TX READ ONLY veya SET TX READ WRITE ile hareketin muhteviyatının belirlenmesi eşzamanlı çalışabilme noktasında sisteme önceden verilen bir ipucudur.
- Bir hareketin tespit edilen izolasyon seviyesi sadece o hareket için bağlayıcı olur. Mesela, T hareketi izolasyon seviyesi *serilenebilir* ise; bu **sadece T için**, diğer bütün hareketlerin sanki T hareketinden önce veya sonra çalışması demektir. Diğer başka hareketler için bağlayıcı bir durum yok; mesela *read uncommitted* seviyede başka bir hareket var ise bu T'nin *commit* olmayan kirli verisini okuyabilir.

- Mesela, sadece kayıt ekleme (*insert into values (.....)*) ve/veya bazı kayıtları silme*(delete from STUDENT where **SId=1***) operasyonları içeren bir hareket için izolasyon seviyesi ***read Committed*** yeterlidir.

- Mesela, 2 hareketimiz var. T1, bir okuldaki bütün öğrencilerin ortalama notlarını hesaplıyor. Diğeri yeni gelen öğrencileri ekliyor. T1 için ***Read committed*** hatta ***read uncommitted*** yeterli. (*ortalamanın %100 doğru olması şart değil)*

Doğru Yalıtım seviyesinin tespiti

STUDENT (sID, sName, GPA, HSsize) *HSsize: high school size,mezun olduğu lise kapasitesi*

Her kutu bir harekete karşılık geliyor. Her Kutunun sonunda COMMIT var.
Her örneğin ilk hareketi (T), serilenebilir. Diğer hareketin (T1 veya T2), T ile beraber çalışırken bulunduğu «düşük» yalıtım seviyesinde başına gelenler tahlil ediliyor.

T
```
Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500
```

T1
```
Set Transaction Isolation Level Read Uncommitted;
Select Avg(GPA) From Student;
```

T2
```
Set Transaction Isolation Level Read Committed;
Select Avg(GPA) From Student;
Select Max(GPA) From Student;
```

Read Uncommitted?

T
```
Update Student Set GPA = (1.1) * GPA;
Update Student Set HSsize = 1500 Where sID = 123;
```

T1
```
Set Transaction Isolation Level Repeatable Read;
Select Avg(GPA) From Student;
Select Avg(HSsize) From Student;
```

Read Committed?

Delete ?

T
```
Insert Into Student [ 100 new tuples ]
```

T1
```
Set Transaction Isolation Level Repeatable Read;
Select Avg(GPA) From Student;
Select Max(GPA) From Student;
```

Doğru Yalıtım seviyesinin tespiti

1. örnekte:

T ve T1 serilenebilir değil. Cunku T1 (araya girip) kirli okuma yapabilir. T,T1 veya T1,T seri planına denk gelemez.

T ve T2 serilenebilir değil. T, T2'nin operasyonları arasına girebilir. Bu durumda kirli okuma olmuyor. Ancak «nonrepeatable read» hatası var. Bu durum herhangi bir seri plana denk gelemiyor. Burda «read uncommitted» olsa: yalitimi daha gevşettik ve T, T2 boyunca her yerde araya girebilir. T2 de T'den kirli veri okuyabilir. Gene serilenebilir değil. Ort. ve Max yanlış hesaplama olacak.

2. örnekte:

T ve T1 serilenebililir. T1, T'nin operasyonlarının arasına giremez. T, gene T1'in işlemlerinin arasına giremez. Çünkü «Repeatable read» (*başkası okuduğum değeri değiştiremez, bu GPA kilidinin bırakılmaması ile sağlanır)* Gerçi bir daha GPA okunmuyor. T1 «read committed» olsa: tekrar okuyamama hatası olmayacak. Ancak T, T1'in arasına girebilir, serilenebilir de olabilir veya olmaz.

3. örnekte:

T ve T1 serilenebilir değil. T1,T'nin arasına girebilir. Bunun T'ye bir zararı yok. Diğer taraftan; T, T1'in operasyonları arasına girebilir: yeni, hayalet kayıtlar. Mevcut kayıtları değiştirmiyoruz. Yani «repeatable read»'e aykiri bir durum olmuyor. Ancak T, DELETE yapıyor olsaydı:::,T araya giremeyecekti! Bu durumda; T1, «repeatable read» serilenebilir'lik için yeterli olurdu.