Fixing the problem via transactions

- If we group Siegfried's statements into one transaction...
- ... then he cannot be returned inconsistent results
- That is, there are two possibilities:
  - Either max and min are computed before the change in price...
  - ... or they are computed after the change in price...
  - ... or in the middle of the changed prices.
  - Regardless, the computation will be done with the same prices

Another possible problem: reversing

- Suppose Brunnhilde executes the (del) and (ins) operations
  - ... and does these as separate actions (i.e., not as a transaction).
- However, she has a soft spot for Siegfried, and loves him despite the fact he likes Blue and Coors Light
  - She thinks better of the table modifications and undoes her table modifications.
- If Siegfried executes the (max)(min) statements after the (ins) but before the undoing...
  - ... He will see a value of 4.50 for both max and min...
  - ... despite the fact that the only prices will end up being {4.00, 4.25}.

Solution: use rollback

- If Brunnhilde executes (del)(ins) as a transaction...
  - ... then its effects cannot be seen until the transaction executes a commit.
- If the transaction executes a **rollback** instead...
  - ... then its effects can never be seen (because it is as if the transaction's operations never happened).

Isolation levels

- SQL defines four **isolation levels**
  - A level reflects a choice regarding what interactions are allowed amongst transactions executing at about the same time.
  - Here we mean, amongst other things, **what changes are visible in the midst of a transaction**.
- Full ACID equals the **serializable** isolation level
- Note:
  - Each DBMS has its own subtle twists on transaction implementations.
  - When concerned about subtle details, you must check the DBMS documentation.

Specifying isolation level

- One of four possibilities:
  - serializable
  - repeatable read
  - read committed (default in Postgres)
  - read uncommitted

```
begin transaction isolation level serializable;
< transaction operations >
commit;
```

```
-- also
begin transaction isolation level repeatable read;
begin transaction isolation level repeatable committed;
begin transaction isolation level repeatable uncommitted;
```

Example

- If Siegfried = (max)(min) is a transaction ...
- ... and Brunnhilde = (del)(ins) is a transaction ...
- ... and if Siegfried runs with isolation-level **serializable**
  - then he will see the database either before or after Brunnhilde's transaction ...
  - ... **but not in the middle of her transaction!**

Important subtlety

- An isolation level determines for some transaction **how that transaction** sees the database
  - It does not constrain others to see the database in the same way!
  - **That is, we cannot force other processes / users into a specific isolation level.**
- Relating this back to the previous example:
  - If Brunnhilde runs as serializable...
  - ... but Siegfried doesn't ..
  - ... then it is possible that Siegfried might see an empty Sells relation (i.e., no prices for the Valhalla Inn).
  - That is, the relation would look this way if Siegfried ran his operations when Brunnhilde's was in the middle of the transaction.

Isolation level: read committed

- If Siegfried runs with this isolation level...
- ... then he can only see committed data, but not necessarily the same data each time.
- Example:
  - Using **read committed**, the interleaving of (max)(del)(ins)(min) is allowed...
  - ... as long as Brunnhilde commits!
  - This means Siegfried would see the **max** price as less than the **min** price (i.e., inconsistent).
  - However, we may decide this possibility is a price we are willing to pay to allow increased concurrency.

Isolation level: repeatable read

- Similar to read committed with the addition...
- ... that if tuples are read again...
- ... then values seen the first time will be seen the second time.
- Note though:
  - The second and subsequent reads may see more tuples as well...
  - ... but the original tuple results will not be modified.

Isolation level: repeatable read

- Example: Assuming Siegfried uses **repeatable read**...
- ... and the order of execution is (max)(del)(ins)(min):
- ... then:
  - (max) obtains prices {4.00, 4.25} for processing
  - (min) obtains prices {4.00, 4.25, 4.50} for processing (i.e., first two values were in the original read; second value was added after the original read)

Isolation level: read uncommitted

- A transaction running under **read uncommitted** can see data in the database...
- ... even if it was written by an uncommitted transaction...
- ... and even if that transaction will never commit!
- Example: If Siegfried runs under read uncommitted, he could see the price of 4.50 even if Brunnhilde later aborts her transaction.
- Note: Postgres treats read uncommitted as read committed (i.e., example above is impossible in Postgres).

Isolation level: summary

- Different isolation levels permit different phenomena
  - **dirty read**: a transaction reads data written by a concurrent uncommitted transaction
  - **non-repeatable read**: a transaction re-reads data it previously read and finds that data has been modified by another transaction (that committed since the initial read)
  - **phantom read**: a transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to a recently-committed transaction.

isolation level	dirty read	non-repeatable read	phantom read
read uncommitted	possible	possible	possible
read committed	not possible	possible	possible
repeatable read	not possible	not possible	possible
serializable	not possible	not possible	not possible

Views

- An SQL **view** is a relation normally defined in terms of stored tables
  - Stored tables sometimes also called stored relations or base tables
- Views can also be defined in terms of other views...
  - ... along with a combination of views and stored tables
- Two kinds of views

1. **Virtual view**
   - Not stored in the database
   - Simply a query for constructing a relation instance
2. **Materialized view**
   - View is constructed and stored as if it were a base table
   - (Note that there are some gotchas here.)

View definition

```
create [materialized] view
       <name> as <query>
```

- Example: CanQuaff(patron, beer) is a view that "contains" patron/beer pairs such that a patron frequents at least one pub that serves the beer
- **quaff** (*kwäf*): (verb) to drink heartily. ORIGIN early 16th century; probably imitative of the sound of drinking.

```
create view CanQuaff as
    select patron, beer
    from Frequents, Sells
    where Frequents.pub = Sells.pub;
```

Accessing a view

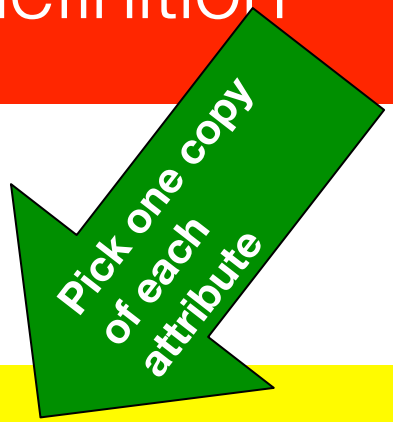
- Views are queried as if it were a base table with a set of attributes
- Note: There exists a limited ability to modify a view (usually the materialized kind) if it is clear what base tables need to be modified

```
select beer from CanQuaff where patron='Cliff';
```

Views and triggers

- In general terms, we cannot modify a virtual view directly
  - After all, it does not really exist as a set of stored tuples independent of the base tables
- However, an instead of trigger associated with a virtual view can help us here
  - The trigger may allow us to interpret view modifications in a way that is sensible
  - The view modification will be broken down (in the trigger's action procedure) into base-table modifications
- Example: a view named **HappyCombo(patron, beer, pub)**
  - This triple indicates that not only does the pub serve the beer...
  - ... but also that the patron frequents the pub and likes the beer.

HappyCombo view definition




```
create view HappyCombo as
    select Likes.patron, Likes.beer, Sells.pub
    from   Likes, Sells, Frequents
    where  Likes.patron = Frequents.patron and
           Likes.beer   = Sells.beer       and
           Sells.pub    = Frequents.pub;
```