

# Extended Operators in SQL and Relational Algebra

Dr. Bing Zhou

# Bags

- A *bag* (or *multi-set*) is like a set, but an element may appear more than once.
- Example:  $\{1,2,1,3\}$  is a bag.
- Example:  $\{1,2,3\}$  is also a bag that happens to be a set.

# Why Bags?

- So far, we have said that relational algebra and SQL operate on relations that are sets of tuples.
- Real RDBMSs treat relations as bags of tuples.
  - SQL, the most important query language for relational databases, is actually a bag language.
- Performance is one of the main reasons; duplicate elimination is expensive since it requires sorting.
  - Some operations, like projection, are much more efficient on bags than sets.
  - The union of two relations as bags is more efficient than the union of sets
- If we use bag semantics, we may have to redefine the meaning of each relational algebra operator.

# Operations on Bags

- **Selection** applies to each tuple, but as a bag operator, we do not eliminate duplicates.
- **Projection** also applies to each tuple, but as a bag operator, we do not eliminate duplicates.
- **Products** and **joins** are done on each pair of tuples, so duplicates in bags have no effect on how we operate, but when constructing the answer, we do not eliminate duplicates

# Bag Semantics: Projection and Selection

- ▶ Projection ( $\pi()$ ): process each tuple independently; a tuple may appear in the resulting relation multiple times.
- ▶ Selection ( $\sigma()$ ): process each tuple independently; a tuple may appear in the resulting relation multiple times.

$R$		
A	B	C
1	2	3
1	2	4
2	3	4
2	3	4

$\pi_{A,B}(R)$	
A	B
1	2
1	2
2	3
2	3

$\sigma_{C \geq 3}(R)$		
A	B	C
1	2	3
1	2	4
2	3	4
2	3	4

# Bag Union

- An element appears in the union of two bags the sum of the number of times it appears in each bag.
- $R \cup S$ : if tuple  $t$  appears  $k$  times in  $R$  and  $l$  times in  $S$ ,  $t$  appears in  $R \cup S$   $k + l$  times.

$R$	
A	B
1	2
1	2
2	3
2	3

$S$	
A	B
1	2
1	2
1	2
2	3
2	4

$R \cup S$	
A	B
1	2
1	2
1	2
1	2
1	2
2	3
2	3
2	3
2	3
2	4

# Bag Intersection

- An element appears in the intersection of two bags the minimum of the number of times it appears in either.
- $R \cap S$ : if tuple  $t$  appears  $k$  times in  $R$  and  $l$  times in  $S$ ,  $t$  appears  $\min \{k, l\}$  times in  $R \cap S$

$R$	
A	B
1	2
1	2
2	3
2	3

$S$	
A	B
1	2
1	2
1	2
2	3
2	4

$R \cap S$	
A	B
1	2
1	2
2	3

# Bag Difference

- An element appears in the difference  $R - S$  of bags as many times as it appears in  $R$ , minus the number of times it appears in  $S$ .
  - But never less than 0 times.
- $R - S$ : if tuple  $t$  appears  $k$  times in  $R$  and  $l$  times in  $S$ ,  $t$  appears in  $R - S$   $\max\{0, k - l\}$  times.

$R$	
A	B
1	2
1	2
2	3
2	3

$S$	
A	B
1	2
1	2
1	2
2	3
2	4

$R - S$	
A	B
2	3



# Bag Semantics: Products and Joins

- **Product ( $\times$ ):** If a tuple  $r$  appears  $k$  times in a relation  $R$  and tuple  $s$  appears  $l$  times in a relation  $S$ , then the tuple  $rs$  appears  $kl$  times in  $R \times S$ .
- **Theta-join and Natural join ( $\bowtie$ ):** Since both can be expressed as applying a selection followed by a projection to a product, use the semantics of selection, projection, and the product.

# Example: Product

R = (

A	B
1	2
1	2

)

S = (

B	C
2	3
4	5
4	5

)

R x S =

A	R.B	S.B	C
1	2	2	3
1	2	4	5
1	2	4	5
1	2	2	3
1	2	4	5
1	2	4	5

# Example: Natural Join

R = (

A	B
1	2
1	2

)

S = (

B	C
2	3
4	5
4	5

)

R  $\bowtie$  S =

A	B	C
1	2	3
1	2	3

# Example: Theta Join

$R = ($

A	B
1	2
1	2

$)$

$S = ($

B	C
2	3
4	5
4	5

$)$

$R \bowtie_{R.B < S.B} S =$

A	R.B	S.B	C
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

# Extended Operators

- Powerful operators based on basic relational operators and bag semantics.
- **Sorting**: convert a relation into a list of tuples.
- **Duplicate elimination**: turn a bag into a set by eliminating duplicate tuples.
- **Grouping**: partition the tuples of a relation into groups, based on their values among specified attributes.
- **Aggregation**: used by the grouping operator and to manipulate/combine attributes.
- **Extended projections**: projection on **steroids**.
- **Outerjoin**: extension of joins that make sure every tuple is in the output.

# Sorting

RA  $\tau_{A_1, A_2, \dots}(R)$ .

SQL SELECT ... FROM ... WHERE ... ORDER BY  $A_1, A_2, \dots$

- ▶ The result is a list of tuples in  $R$  but with the tuples sorted by their values in attributes  $A_1, A_2, \dots$
- ▶ In SQL, use DESC after an attribute to specify sorting in descending order; ASC is the default.
- ▶ If you use the result in another query, sorted order is lost.

# Example: Sorting

$R = ($

A	B
1	2
3	4
5	2

$)$

$$\text{TAU}_B(R) = [(5,2), (1,2), (3,4)]$$

# Duplicate Elimination

**RA**  $\delta(R)$  is the relation containing exactly one copy of each tuple in  $R$ .

**SQL** SELECT DISTINCT ...

- ▶ Duplicate elimination is *expensive*, since tuples must be sorted or partitioned.



# Example: Duplicate Elimination

$R = ($

A	B
1	2
3	4
1	2

$)$

$\delta(R) =$

A	B
1	2
3	4

# Extended Projection

- Using the same  $\pi_L$  operator, we allow the list  $L$  to contain arbitrary expressions involving attributes, for example:
  - Arithmetic on attributes, e.g.,  $A+B$ .
  - Duplicate occurrences of the same attribute.

# Example: Extended Projection

$R = ($

A	B
1	2
3	4

$)$

$\pi_{A+B, A, A} (R) =$

A+B	A1	A2
3	1	1
7	3	3

# Aggregation Operators

- Operators that summarize or aggregate the values in a single attribute of a relation.
- Operators are the same in relational algebra and SQL.
- All operators treat a relation as a bag of tuples.
- SUM: computes the sum of a column with numerical values.
- AVG: computes the average of a column with numerical values.
- MIN and MAX:
  - for a column with numerical values, computes the smallest or largest value, respectively.
  - for a column with string or character values, computes the lexicographically smallest or largest values, respectively.
- COUNT: computes the number of tuples in a column.
- In SQL, can use COUNT (\*) to count the number of tuples in a relation.

# Example: Aggregation

R = (

A	B
1	3
3	4
3	2

)

SUM(A) = 7

COUNT(A) = 3

MAX(B) = 4

AVG(B) = 3

# Grouping Operator

- How do we answer the query “Count the number of classes and the total enrollment of the classes each department teaches”?
- Can we answer the query using the operators discussed so far?
- We need to group the tuples of Teach by DeptName and then aggregate within each group.
- Use the grouping operator.

# Applying $\gamma_L(R)$

- ▶ How do we answer the query “Count the number of classes and total enrollment of the classes each department teaches”?
  1. Group Courses by DeptName.
  2. For each group, create a new attribute that stores the number of classes taught by the department.
  3. For each group, create a new attribute that stores the total enrollment of the classes taught by the department.
- ▶  $\gamma_L(\text{Courses})$ , where  $L$  is a list containing three elements:
  1. DeptName: the *grouping attribute*,
  2. COUNT(Number)  $\rightarrow$  NumCourses: an *aggregated attribute* computing the count of the Number attribute in each group and naming the new attribute NumCourses, and
  3. SUM(Enrollment)  $\rightarrow$  TotalEnrollment: an aggregated attribute computing the total of the Enrollment attribute and naming the new attribute TotalEnrollment.

# Example: Grouping/Aggregation

$R = ($

A	B	C
1	2	3
4	5	6
1	2	5

$)$

$\gamma_{A,B,AVG(C)}(R) = ??$

First, group  $R$  by  $A$  and  $B$  :

A	B	C
1	2	3
1	2	5
4	5	6

Then, average  $C$  within groups:

A	B	AVG(C)
1	2	4
4	5	6



# Outerjoin

- Suppose we join  $R \bowtie_C S$ .
- A tuple of  $R$  that has no tuple of  $S$  with which it joins is said to be *dangling*.
  - Similarly for a tuple of  $S$ .
- Outerjoin preserves dangling tuples by padding them with a special NULL symbol in the result.

# Example: Outerjoin

R = (

A	B
1	2
4	5

)

S = (

B	C
2	3
6	7

)

(1,2) joins with (2,3), but the other two tuples are dangling.

R OUTERJOIN S =

A	B	C
1	2	3
4	5	NULL
NULL	6	7

# Exercise

- $R(A,B): \{(0,1),(2,3),(0,1),(2,4),(3,4)\}$
- $S(B,C): \{(0,1),(2,4),(2,5),(3,4),(0,2),(3,4)\}$

Computer:

- 1)  $\pi_{B+1,C-1}(S)$
- 2)  $\tau_{b,a}(R)$
- 3)  $\delta(R)$
- 4)  $\gamma_{a, \text{sum}(b)}(R)$
- 5)  $R \text{ outjoin } S$

# Transactions, Views, Indexes

# Why Transactions?

- **Concurrent database access**
  - Execute *sequence of SQL statements* so they appear to be running in isolation
- **Resilience to system failures**
  - Guarantee all-or-nothing execution, regardless of failures

# Why Transactions?

- Database systems are normally being accessed by many users or processes at the same time.
  - Both queries and modifications.
- Unlike operating systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions.

# Example: Bad Interaction

- Joint account holders each take \$100 from different ATM's at about the same time.
  - The DBMS better make sure one account deduction doesn't get lost.
- **Compare:** An OS allows two people to edit a document at the same time. If both write, one's changes get lost.

# Introduction to Transactions

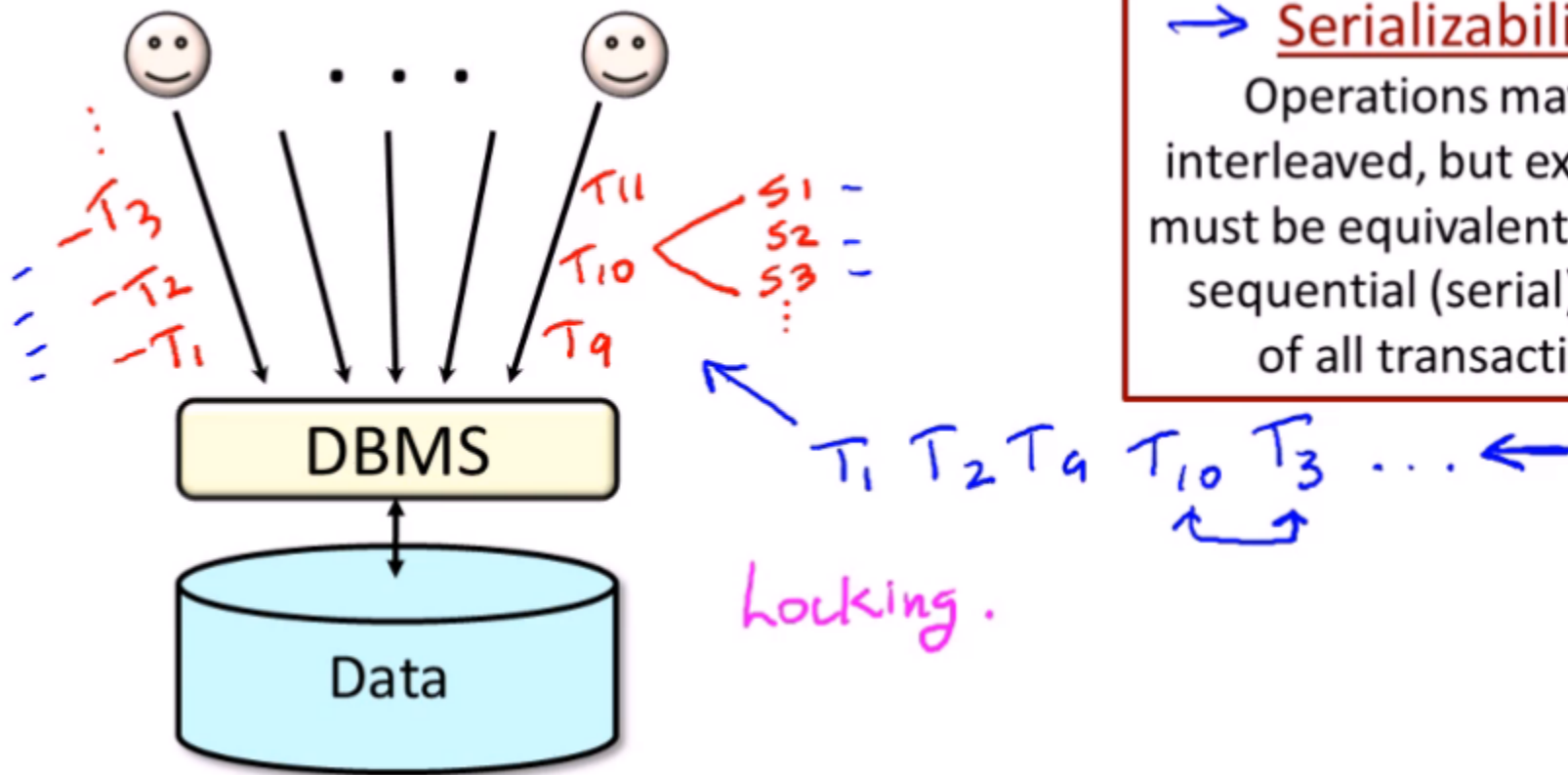
- *Transaction* = a sequence of one or more SQL statements treated as one unit.
- Normally with some strong properties regarding concurrency.
- Formed in SQL from single statements or explicit programmer control.
- Depending on the implementation, a transaction may start:
  - Implicitly, with the execution of a SELECT, UPDATE, ... statement, or
  - Explicitly, with a **BEGIN TRANSACTION** statement
- Transaction finishes with a **COMMIT** or **ROLLBACK** statement



# ACID Properties

- *ACID Properties* are:
  - *Atomic* : Whole transaction or none is done.
  - *Consistent* : Database constraints preserved.
  - *Isolated* : It appears to the user as if only one process executes at a time.
  - *Durable* : Effects of a process survive a crash.

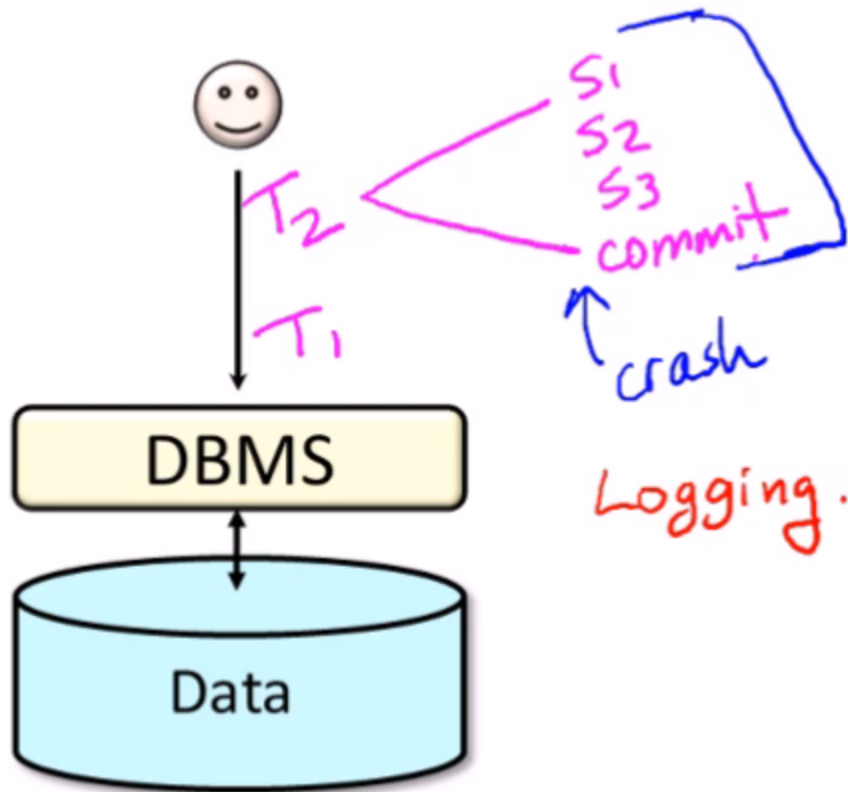
## (ACID Properties) **Isolation**



### → Serializability

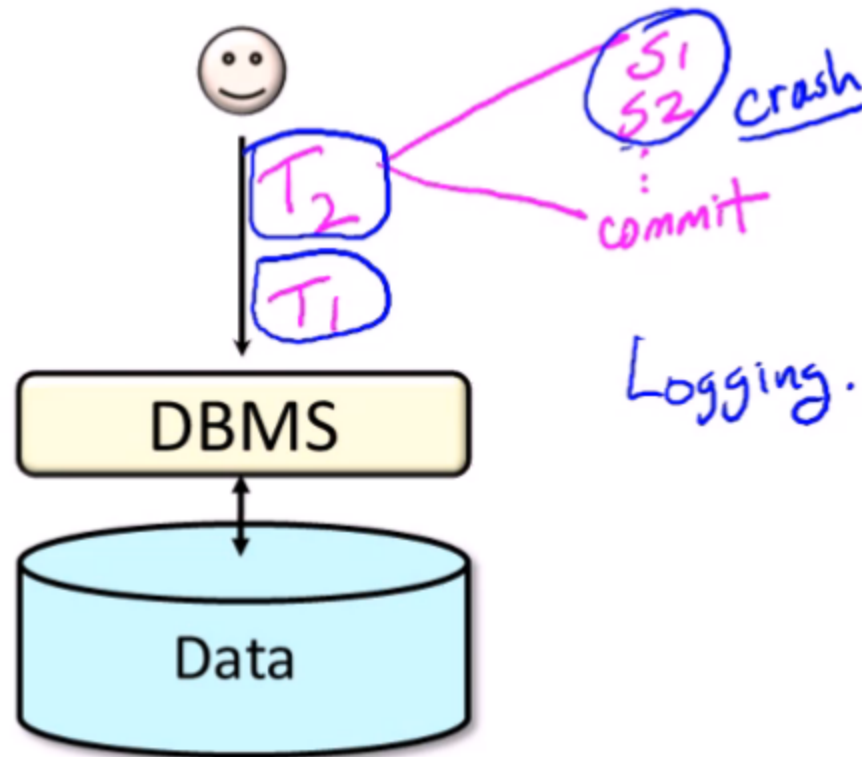
Operations may be interleaved, but execution must be equivalent to *some* sequential (serial) order of all transactions

## (ACID Properties) **Durability**



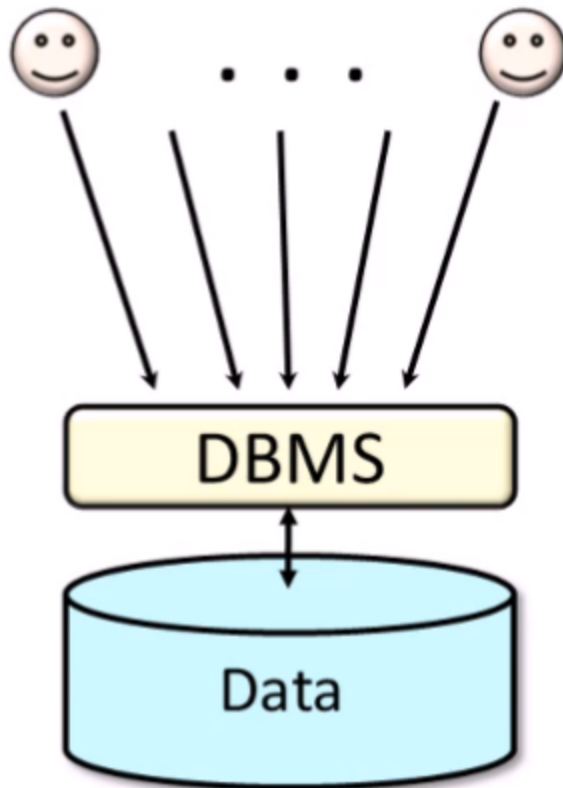
If system crashes  
after transaction commits,  
all effects of transaction  
remain in database

## (ACID Properties) **Atomicity**



Each transaction is  
“all-or-nothing,”  
never left half done

## (ACID Properties) **Consistency**



Each client, each transaction:

- Can assume all constraints hold when transaction begins
- Must guarantee all constraints hold when transaction ends

Serializability  $\Rightarrow$  constraints always hold

$T_1$   $T_2$   $T_3$

Handwritten pink arrows point from the labels  $T_1$ ,  $T_2$ , and  $T_3$  towards the DBMS box in the diagram.

# COMMIT

- The SQL statement **COMMIT** causes a transaction to complete.
  - Its database modifications are now permanent in the database.

# ROLLBACK

- The SQL statement **ROLLBACK** also causes the transaction to end, but by *aborting*.
  - No effects on the database.
- Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it.

# Example: Interacting Processes

- Assume a **Sells(bar,beer,price)** relation, and suppose that Joe's Bar sells only Export for \$2.50 and Sleeman for \$3.00.
- Sally is querying **Sells** for the highest and lowest price Joe charges.
- Joe decides to stop selling Export and Sleeman, and to sell only Heineken at \$3.50.



# Sally's Program

- Sally executes the following two SQL statements called (min) and (max) to help us remember what they do.

(max) SELECT MAX(price) FROM Sells  
WHERE bar = 'Joe''s Bar';

(min) SELECT MIN(price) FROM Sells  
WHERE bar = 'Joe''s Bar';

# Joe's Program

- At about the same time, Joe executes the following steps: (del) and (ins).

(del)      DELETE FROM Sells  
             WHERE bar = 'Joe's Bar';

(ins)      INSERT INTO Sells  
             VALUES('Joe's Bar', 'Heineken', 3.50);

# Interleaving of Statements

- Although **(max)** must come before **(min)**, and **(del)** must come before **(ins)**, there are no other constraints on the order of these statements, unless we group Sally's and/or Joe's statements into transactions.

# Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min).

Joe's Prices:      {2.50,3.00}      {2.50,3.00}      {3.50}      {3.50}

Statement:                      (max)                      (del)                      (ins)                      (min)

Result:                              3.00    3.50

- Sally sees  $MAX < MIN$ !

# Fixing the Problem by Using Transactions

- If we group Sally's statements (max)(min) into one transaction, then she cannot see this inconsistency.
- She sees Joe's prices at some fixed time.
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.

# Another Problem: Rollback

- Suppose Joe executes `(del)(ins)`, not as a transaction, but after executing these statements, thinks better of it and issues a ROLLBACK statement.
- If Sally executes her statements after `(ins)` but before the rollback, she sees a value, 3.50, that never existed in the database.

# Solution

- If Joe executes `(del)(ins)` as a transaction, its effect cannot be seen by others until the transaction executes COMMIT.
  - If the transaction executes ROLLBACK instead, then its effects can *never* be seen.

# Isolation Levels

- SQL defines four *isolation levels*  
= choices about what interactions are allowed by transactions that execute at about the same time.
- Each DBMS implements transactions in its own way.



# Choosing the Isolation Level

- Within a transaction, we can say:  
    SET TRANSACTION ISOLATION LEVEL  $X$   
    where  $X$  =
  1. SERIALIZABLE
  2. REPEATABLE READ
  3. READ COMMITTED
  4. READ UNCOMMITTED

# Serializable Transactions

- If Sally = (max)(min) and Joe = (del)(ins) are each transactions, and Sally runs with isolation level SERIALIZABLE, then she will see the database either before or after Joe runs, but not in the middle.

# Isolation Level Is Personal Choice

- Your choice, e.g., run serializable, affects only how *you* see the database, not how others see it.
- **Example:** If Joe runs serializable, but Sally doesn't, then Sally might see no prices for Joe's Bar.
  - i.e., it looks to Sally as if she ran in the middle of Joe's transaction.

# Read-Committed Transactions

- If Sally runs with isolation level READ COMMITTED, then she can see only committed data, but not necessarily the same data each time.
- **Example:** Under READ COMMITTED, the interleaving (max)(del)(ins)(min) is allowed, as long as Joe commits.
  - Sally sees  $MAX < MIN$ .

# Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time.
  - But the second and subsequent reads may see *more* tuples as well.

# Example: Repeatable Read

- Suppose Sally runs under REPEATABLE READ, and the order of execution is (max)(del)(ins)(min).
  - (max) sees prices 2.50 and 3.00.
  - (min) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by (max).

# Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never).
- **Example:** If Sally runs under READ UNCOMMITTED, she could see a price 3.50 even if Joe later aborts.

# Views

- A *view* is a relation defined in terms of stored tables (called *base tables* ) and other views.
- Two kinds:
  - 1. Virtual* = not stored in the database; just a query for constructing the relation.
  - 2. Materialized* = actually constructed and stored.
- Just like a table, a view can be queried.
- Unlike a table, a view cannot be updated unless it satisfies certain conditions.



# Declaring Views

- Declare by:

```
CREATE [MATERIALIZED] VIEW <name> AS <query>;
```

- Default is virtual.

# Example: View Definition

- Suppose we want to perform a set of queries on those students who have taken courses both in the computer science and the mathematics departments.
- Let us create a view to store the PIDs of these students and the CS-Math course pairs they took.

**CREATE VIEW** CSMath AS

```
SELECT T1.StudentPID, T1.Number, T2.Number
FROM Take AS T1, Take AS T2
WHERE (T1.StudentPID = T2.StudentPID)
      AND (T1.DeptName = 'CS')
      AND (T2.DeptName = 'Math');
```

# Example: View Definition

- **CanDrink(drinker, beer)** is a view “containing” the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
  SELECT drinker, beer
  FROM Frequents, Sells
  WHERE Frequents.bar = Sells.bar;
```

# Example: Accessing a View

- Query a view as if it were a base table.
  - Also: a limited ability to modify views if it makes sense as a modification of one underlying base table.
- **Example query:**  
SELECT beer FROM CanDrink  
WHERE drinker = 'Sally';

# Indexes

- *Index* = data structure used to speed access to tuples of a relation, given values of one or more attributes.
- Could be a hash table, but in a DBMS it is always a balanced search tree with giant nodes called a *B-tree*.

# Declaring Indexes

- No standard!

- Typical syntax:

```
CREATE INDEX BeerInd ON Beers(manf);
```

```
CREATE INDEX SellInd ON Sells(bar, beer);
```

# Using Indexes

- Given a value  $v$ , the index takes us to only those tuples that have  $v$  in the attribute(s) of the index.
- **Example:** use BeerInd and SellInd to find the prices of beers manufactured by Pete's and sold by Joe. (next slide)

# Using Indexes --- (2)

```
SELECT price FROM Beers, Sells
WHERE manf = 'Pete's' AND
      Beers.name = Sells.beer AND
      bar = 'Joe's Bar';
```

1. Use BeerInd to get all the beers made by Pete's.
2. Then use SellInd to get prices of those beers, with bar = 'Joe's Bar'



# Database Tuning

- A major problem in making a database run fast is deciding which indexes to create.
- **Pro:** An index speeds up queries that can use it.
- **Con:** An index slows down all modifications on its relation because the index must be modified too.