Cristian Gonzales (cralgonz) (Student ID: 1465923)

- **[Important] data types for attributes (relations):** CHAR(n), VARCHAR(n), BOOLEAN, INT/INTEGER, DATE 'YYYY-MM-DD', TIME 'HH:MM:SS.sss', DECIMAL(n,d) (→ n = total digits, d = digits after decimal point)
- **String comparison**: One string is less than the second string if the first string is a proper prefix of the second (length does not matter)
- **Modifying Relation Schemas**
  - ALTER TABLE <R> ADD <name> CHAR(n);
  - ALTER TABLE <R> DROP <name>; DROP TABLE <R>;
  - ALTER TABLE <R> ALTER COLUMN <attribute> [SET | DROP] NOT NULL;
- **Comparisons:** s LIKE 'Star%' (%: ≥ 0 characters; _: one character)
- **NULL values in WHERE clauses**
  - In arithmetic, w/ any value the answer will always be NULL
    - 0 * x = NULL   x - x = NULL
  - In comparisons, the result is UNKNOWN
  - Syntax: x IS NULL;   x IS NOT NULL;
- **Truth-Value UNKNOWN**
  - TRUE = 1, FALSE = 0, UNKNOWN = ½
  - AND of 2 truth values is the minimum of those values
  - OR of 2 truth values is the maximum of those values
  - Negation is simply the opposite (UNKNOWN remains the same)
- **Ordering:** ORDER BY <list of attributes> [ASC | DESC]
- **UNION, INTERSECT, EXCEPT ⇔ ∪, ∩, –**
  - A – B ⇔ Elements of A that aren't in B
  - A INTERSECT ALL B: The number of times tuple *t* appears is the minimum number of times it appears in A & B
  - A EXCEPT ALL B: Tuple *t* appears as many times as the difference of the number of times it appears in A minus the number of times it appears in B
- **JOIN Expressions**
  - CROSS JOIN: Simply cross the two schemas together and that becomes the join.
  - JOIN: Simply join the relations (all attributes will remain constant).
  - NATURAL JOIN: *Seamlessly* submerge two relations based on common attributes.
  - NATURAL FULL OUTER JOIN: Same as natural join, except augment result of a join by dangling tuples, padded with NULL values for both relations
  - NATURAL [LEFT | RIGHT] OUTER JOIN: Same as natural join, except augment result of a join by dangling tuples, padded with NULL values for the [left | right] relations
- **Miscellaneous SQL Expressions**
  - WHERE [NOT] EXISTS, WHERE <attribute> [IN | NOT IN] <expression>, ANY(<expression>), ALL(<expression>)
- **Aggregation: SUM, AVG, MIN, MAX, COUNT**
  - COUNT(*) → counts all tuples in relation of FROM & WHERE clause of query
  - COUNT(DISTINCT x) → counts number of distinct values in column x
  - GROUP BY follows WHERE clause in aggregated statements
  - NULL is ignored in any aggregation
  - Any aggregation over an empty bag of values is NULL (except for COUNT, which is 0) (i.e., SUM(R) → NULL, COUNT(R) → 0)
    - GROUP BY does not ignore NULLs.
  - GROUP BY … HAVING <condition of group>;
    - Any attribute of relations in FROM clause may be aggregated in HAVING, but only attributes in GROUP BY may appear unaggregated in HAVING
    - May also include ANY, or EVERY (EVERY is bound to HAVING clauses *only*) in HAVING clause
  - Both of these are the same:

```
SELECT studioName             SELECT DISTINCT studioName
FROM Movies                   FROM Movies;
GROUP BY studioName;
```

- **DB Modifications**
  - INSERT INTO R(A_1, …, A_n) VALUES (V_1, …, V_n);
  - DELETE FROM R WHERE <condition>;
    - Delete all tuples from a table, but not the table itself: DELETE FROM R;
  - UPDATE R SET <new value assignment(s)> WHERE <condition>;
    - (Example of concatenation)

```
UPDATE MovieExec
SET name = 'Pres. ' || name
WHERE […]
```

- **Foreign Keys**
  - Referenced keys must be declared PRIMARY KEY or UNIQUE
  - In CREATE TABLE: FOREIGN KEY (<attribute>) REFERENCES <table>(<attribute>) [ON DELETE | ON UPDATE] [SET NULL | CASCADE]   – or –
    <attribute> <type> REFERENCES <table>(<attribute>) [ON DELETE | ON UPDATE] [SET NULL | CASCADE]
  - Enforcing Foreign Key Constraints: If there is a foreign-key constraint from referring relation R to referenced relation S, then violations may occur two ways:
    - An insert/update to R that introduces values that are not found in S, or
    - A deletion/update to S causes some tuples of R to reference a value that no longer exists
  - Referential Integrity
    - The Default Policy: Reject violating modifications
    - The Cascade Policy: Changes to referenced attributes are mimicked at foreign key
    - The Set-Null Policy: When modification to referenced relation affects foreign-key value, set the foreign key value to NULL
  - Circular constraints
    - Group two insertions into single transaction, or
    - Tell DBMS to not check constraints until transaction is about to commit: SET CONSTRAINT <foreign key name> [DEFERRABLE | NOT DEFERRABLE] [INITIALLY [IMMEDIATE | DEFFERED]];   *(this is followed after CREATE TABLE declaration, can also be declared after REFERENCES declaration)*
- **Constraints and Triggers**
  - Table declaration: NOT NULL (this bars the Set-Null policy for referential integrity)
  - CHECK (<condition>) can be declared after table declaration or WHERE clause in a query (must appear in FROM in latter case)
    - Only checked on an insert or update, but not a delete
    - Must evaluate to TRUE or UNKNOWN
  - SQL supports CHECK with subquery, but Postgres does *not*.
  - Attribute based CHECK example

```
CREATE TABLE Sells (
    bar CHAR(20),
    beer CHAR(20) CHECK (beer IN (SELECT name FROM Beers)),
    price REAL CONSTRAINT price_is_cheap CHECK (price <=5.00)
);
ALTER TABLE Sells DROP CONSTRAINT price_is_cheap;
ALTER TABLE Sells ADD CONSTRAINT price_is_cheap CHECK (price <= 5.00);
```

  - Tuple based CHECK example

```
CREATE TABLE Sells (
    bar CHAR(20),
    price REAL,
    CHECK (bar = 'Joe''s Bar' OR price <= 5.00)
);
```

  - In a trigger,
    - FOR EACH ROW indicates a row-level trigger, and if it is omitted it is a statement-level trigger
    - [AFTER | BEFORE] [INSERT | DELETE | UPDATE [ON]] OF […]
    - INSERT statements imply a new tuple, or a new table (set of inserted tuples). Syntax: REFERENCING [NEW | OLD] [TUPLE | TABLE] AS <name>
    - Surround by BEGIN […] END for multiple actions
  - Trigger example

```
CREATE TRIGGER PriceTrig
    AFTER UPDATE OF price ON Sells
    REFERENCING
        OLD ROW AS old
        NEW ROW AS new
    FOR EACH ROW
    WHEN (new.price > old.price + 1)
    INSERT INTO RipoffBars
        VALUES (new.bar);
```

- **Transactions**
  - ACID transactions
    - Atomicity: Both transactions [dependent on each other] are done, or neither is done.
    - Consistency: Any transaction will bring the database from one valid state to another.
    - Isolation: Concurrent execution of transactions results in a system state that would be obtained if transactions were executed one after the other.
    - Durability: Once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.
  - Serializability: Illusion that two transactions happen one after the other, even if they happen near the same time as one another (ensures atomicity)
  - Dirty data: data that is written by a transaction but has not yet been committed by the transaction.
  - Dirty read: read of dirty data written by another transaction.
    - Allowing: More parallelism, serious problems
    - Not allowing: Less parallelism, more overhead, cleaner semantics
  - Phantom tuples: tuples that result from insertions into DB while transaction is executing
  - START TRANSACTION, COMMIT (ensures durability), ROLLBACK
  - SET TRANSACTION [READ ONLY | READ WRITE] ISOLATION LEVEL <isolation level> *(stated before transaction begins)*

Isolation Level	Dirty Reads	Nonrepeatable reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read committed	Not allowed	Allowed	Allowed
Repeatable reads	Not allowed	Not allowed	Allowed
Serializable	Not allowed	Not allowed	Not allowed

    - Snapshot Isolation and Read committed are most frequently used iso levels *(and also the default depending on DB implementation)*
    - Read committed may still read different values committed by transactions (nonrepeatable reads allowed)
    - Repeatable read keeps value consistent even if it was changed by different transaction
- **Views and Indexes**
  - CREATE VIEW <view name> AS SELECT […] FROM […] WHERE […];
  - Renaming view:

```
CREATE VIEW <view name>(<new attr. name>, <new attr. name>) AS
        SELECT <old attribute name>, <old attribute name> FROM […];
```

  - Views are queried in the same fashion as regular relations
  - Updating views
    - List in SELECT clause must include enough attributes that for every tuple inserted into the view, we can fill other attributes with null values
    - FROM can only consist of one occurrence R and no other relation
    - WHERE must not involve R in subquery
    - Example

```
CREATE VIEW ParamountMovies AS
    SELECT title, year
    FROM Movies
    WHERE studioName = 'Paramount';
[…]
UPDATE ParamountMovies
SET year = 1979
WHERE title = 'Star Trek the Movie';
```

- Instead-Of Triggers on Views (example)

```
CREATE TRIGGER ParamountInsert
INSTEAD OF INSERT ON ParamountMovies
REFERENCING NEW ROW AS NewRow
FOR EACH ROW
INSERT INTO Movies(title, year, studioName)
VALUES(NewRow.title, NewRow.year, 'Paramount');
```

- Creating an index:

```
CREATE INDEX <index name> ON <relation>(<attribute>, <attribute>, […]);
DROP INDEX <index name>;
```

- Choice of order matters in multi-attribute indexes
- Motivation for indexes
  - Frequent queries to multi-attribute sets can be done in a more efficient manner.
  - Data that is stored on multiple pages on a disk may require a long lookup time. Indexes would decrease the lookup time linearly.
- Disadvantages of indexes
  - Huge number of indexes, space for indexes, cache impact of searching indexes, update time for indexes when table is modified
- Keys are indexed (usually automatically) to help maintain uniqueness and check foreign key references to primary keys

- Relational Algebra
  - Strictly set based (*no duplicates*)
  - (σ, π, x, $\cup$, –) cannot be proven using any of the other operators, but may prove all other operators
  - Selection: $\sigma_{condition}(R)$ Projection: $\pi_{\text{<attribute list>}}(R)$
  - Set Union: $R \cup S$ Set Difference: $R - S$
    - R & S must be union compatible (same number/type of columns)
    - $R \cup S = S \cup R$ (commutativity); $(R \cup S) \cup T = R \cup (S \cup T)$ (associativity)
    - $R \times (S \cup T) = (R \times S) \cup (R \times T)$
  - How set difference is derived: $R \cap S = R - (R - S) = S - (S - R)$
  - Cardinality of R: $|R|$ (the number of elements in a set)
  - Theta Join: $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
    - To calculate, take R x S, and then select from product only those that satisfy θ
  - Natural Join: $R \bowtie S = \pi_{(attr(R) \cup attr(S))}(\sigma_{R.A1 = S.A1 \text{ AND } […] \text{ AND } R.Ak = S.Ak}(R \times S))$
  - Semi-Join: $R \ltimes S = \pi_{attr(R)}(R \bowtie S)$
    - To calculate, take ($R \bowtie S$) and project the output the projection of just the attributes of R
  - The division of $R \div S$ is the relation consisting of all tuples $(a_1, …, a_{r-s})$ such that for every tuple $(b_1, …, b_s)$ in S, the tuple $(a_1, …, a_{r-s}, b_1, …, b_s)$ is in R
  - For $R(a_1, …, a_m, b_1, …, b_n)$ and $S(b_1, …, b_n)$, $R \div S = \pi_{a1,…,am}(R) - \pi_{a1,…,am}((\pi_{a1,…,am}(R) \times S) - R)$
    - Example: Find the sids of all students who are enrolled in all courses taught by "Ullman": $\pi_{sid,cid}(\text{Enrollment}) \div \pi_{cid}(\sigma_{instructor\text{-}name='Ullman'}(\text{Course}))$
  - Renaming: $\rho_{s(A1,…,An)}(R)$
  - Outerjoins ($R \bowtie S$ with a dot on top of the operator) starts with $R \bowtie S$ and then adds any dangling tuples from R or S, padded with the null symbol.
    - For a theta outer join, first calculate a theta join, and then any original tuple that failed to join with any tuple of the other relation on the theta join, are included and padded with nulls
- **Three-Tier Architecture:** Web-Server tier (client) → Application tier (business logic) → DB
- **The SQL Environment**: Schemas (Group of tables) → Catalogs (Group of Schemas) → Clusters (Group of Catalogs)
- **Application Programming**
  - plpgsql programming: (Note: Procedures do not have a return type)
  - Equivalence → := , Comparison → =
  - Triggers may invoke stored procedures

```
CREATE FUNCTION functionName(IN var INTEGER) RETURNS INTEGER AS $body$
DECLARE
    theBeer CHAR(20); thePrice REAL;
    DECLARE c CURSOR FOR
        SELECT beer, price FROM Sells WHERE bar = 'Joe''s Bar' FOR UPDATE;
BEGIN
    OPEN c;
    LOOP;
        FETCH c INTO theBeer, thePrice;
        EXIT WHEN NOT FOUND;
        IF thePrice < 3.00 THEN
            UPDATE Sells SET price = thePrice + 1.00
            WHERE CURRENT OF c;
        END IF;
    END LOOP;
    CLOSE c;
END;
$body$
LANGUAGE plpgsql;
```

- Embedded SQL

```
EXEC SQL BEGIN DECLARE SECTION;
char theBeer[21]; float thePrice; char query[MAX_LENGTH];
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE c CURSOR FOR
        SELECT beer, price FROM Sells WHERE bar = 'Joe''s Bar';
EXEC SQL OPEN CURSOR c;
while(1){
        EXEC SQL PREPARE q FROM :query;
        EXEC SQL EXECUTE q;
        EXEC SQL FETCH c INTO :theBeer, :thePrice;
        if (NOT_FOUND) break;
}
EXEC SQL CLOSE CURSOR c;
```

- JBDC: `Statement execStat = conn.createStatement(); ResultSet worths = execStat.executeQuery(""); while(worths.next()) int worth = worths.getInt(1);`

- **Functional Dependencies**
  - Armstrong's Axioms
    - Let X, Y, & Z denote sets of attributes over a relation schema R
      - Reflexivity: If $Y \subseteq X$, then $X \Rightarrow Y$
      - Augmentation: If $X \Rightarrow Y$, then $XZ \Rightarrow YZ$
      - Transitivity: If $X \Rightarrow Y$ and $Y \Rightarrow Z$, then $X \Rightarrow Z$
    - Completeness: If a set $\mathcal{F}$ of FDs implies F, then F can be derived from $\mathcal{F}$ by applying Armstrong's axioms (if $\mathcal{F}$ implies F, then $\mathcal{F}$ generates F).
    - Soundness: If F can be derived from a set of FDs $\mathcal{F}$ through Armstrong's axioms, then $\mathcal{F}$ implies F (if $\mathcal{F}$ generates F, then $\mathcal{F}$ implies F).
    - With Soundness and completeness, we know that $\mathcal{F}$ implies F *iff* $\mathcal{F}$ generates F.
  - Union, Decomposition, and Pseudo-Transitivity Rules
    - Union: If $X \Rightarrow Y$ and $X \Rightarrow Z$, then $X \Rightarrow YZ$.
      Since $X \Rightarrow Z$, we get $XY \Rightarrow YZ$ (augmentation). Since $X \Rightarrow Y$, we get $X \Rightarrow XY$ (augmentation). Therefore, $X \Rightarrow YZ$ (transitivity).
    - Decomposition: If $X \Rightarrow YZ$, then $X \Rightarrow Y$ and $X \Rightarrow Z$.
      $X \Rightarrow YZ$ (given). $YZ \Rightarrow Y$ (reflexivity). $YZ \Rightarrow Z$ (reflexivity). Therefore, $X \Rightarrow Y$ and $X \Rightarrow Z$ (transitivity).
    - Pseudo-Transitivity: If $X \Rightarrow Y$ and $WY \Rightarrow Z$, then $XW \Rightarrow Z$.
      $XW \Rightarrow WY$ (augmentation). $WY \Rightarrow Z$ (given). Therefore, $XW \Rightarrow Z$ (transitivity).
  - Algorithm for FD's
    - To determine if an FD $X \Rightarrow Y$ is implied by $\mathcal{F}$, compute X^+ and check if $Y \subseteq X^+$
    - Algorithm can be modified to compute candidate keys
      - Compute the closure of a single attribute in X^+. Then compute the closure of 2 attributes, 3 attributes, and so on.
      - If the closure of a set of attributes contains all the attributes of the relation, then it is a superkey.
      - If no proper subset of those attributes has a closure that contains all attributes of the relation, then it is a key.
  - R is in BCNF if for every FD $X \Rightarrow A$ in $\mathcal{F}$, one of the following is true:
    - $X \Rightarrow A$ is a trivial FD ($A \in X$) , or
    - X is a superkey

    ** Any binary relation is in BCNF
  - R is in 3NF if for every FD $X \Rightarrow A$ in $\mathcal{F}$, one of the following is true:
    - $X \Rightarrow A$ is a trivial FD ($A \in X$) , or
    - X is a superkey , or
    - A is a part of some key of R
  - Lossless Join Decomposition
    - Let R be a relation and be a set of FDs that hold over R. A decomposition of R into relation schemas R_1 and R_2 is lossless if $\mathcal{F}^+$ contains either:
      1. $R_1 \cap R_2 \Rightarrow R_1$, or
      2. $R_1 \cap R_2 \Rightarrow R_2$
    - If the decomposition is being split into for more than two relations, the Chase method must be used.
  - Decomposition and Normalization
    - It is always possible to decompose schema into a set of BCNF relations that *eliminates anomalies* and is a *lossless join decomposition*. Though, the schema might not always be *dependency-preserving*.
    - It is always possible to decompose schema into a set of 3NF relations that is a *lossless join decomposition* and is *dependency preserving*. Though, the schema might not always *eliminate anomalies*.
- **OLAP (On-Line Analytic Processing)**
  - Dimension Tables, Fact Tables. Dimension Attr.: a key of dimension table. Dependent Attr.: Fact value determined by dimension table. Data Cubes: keys of dimension tables are dim.
  - Measures: Aggregation values along data cube. Roll-up: Aggregate along one or more dimensions. Drill-down: Break down aggregate into its respective parts.
- **XML and DTDs**
  - Well Formed XML: `<? xml version="1.0" encoding="utf-8 standalone="yes"?>`
    - Namespaces: `<md: StarMovieData xmlns:md = "http://[…]/movies"></md:StarMovieData>`
      - Any nested tags that adhere to this namespace must be prefixed with md:
  - Valid XML: involves DTD that defines allowed tags and how they may be nested
  - `<!ELEMENT Movie EMPTY>` `<Movie title="Star" year="3005 genre="SciFi />`

```
<!ATTLIST Movie
    title CDATA #REQUIRED
    year CDATA #REQUIRED
    genre (comedy | drama | sciFi) #IMPLIED    >
```

  - `<!DOCTYPE Stars []`

```
    <!ELEMENT Stars (Star*)>
    <!ELEMENT Star(Name, Address+, Movies)>
    <!ELEMENT Name(#PCDATA) […] ]>
```