|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Key | - PK constraint: attributes cannot be NULL.  If (A,B,C) is definitely a key/superkey, then (A,B,C,D) is a superkey | | | | | | | | | | | | | | | | If (A,B) is definitely a superkey, then (A,B,C) is not a key  If (A,B) is definitely a key, then it is possible that (B,C,D) is also a key | | | | | | | | | |
| Foreign Key | | | FK = subset of attributes of relation R1 that refers to the PK of relation R2  R1 is the referencing relation; R2 is the referenced relation  Constraint: each FK in R­1 must be either PK in R2 or NULL value (or tuple w at least 1 NULL value) | | | | | | | | | | | | | | | | | | | | | | | (R1.Ai1, R1.Ai2, ...) (R2.Aj1, R2.Aj2, ...)  Referencing, referenced relation can be the same relation |
| Three-Valued Logic | | Boolean values = {True, False, NULL}  NULL return NULL | | | | | | | | a b: If false -> false, elif contain NULL -> NULL  If both true, return true | | | | | | | | | | | | | a b: If true -> true, elif contain NULL -> NULL  If both false, return FALSE | | | |
| Any relational/arithmetic operation w NULL produces NULL values | | | | | | | | | | | | | | | | (,): treat NULL as values. If both not NULL, default to = or <> | | | | | | | | |
| Unary Ops | | Selection: select all tuples from r that satisfy c, , | | | | | | | | | | | | | Operator Precedence: (), **op**, , , with **op** {=, <>, <, ≤, ≥, >, , } | | | | | | | | | | | |
| Projection: keeps only the column/project specified in and in that order  , where is an ordered list of attributes | | | | | | | | | | | | | | | | | Don't allow operations (e.g. ) or duplicates (e.g. ) on | | | | | | | |
| Renaming: renames attributes listed in ,  (order of attribute don't matter, new old) | | | | | | | | | | | | | | | - No 2 diff attributes may be renamed to the same name  - No attributes may be renamed more than once in a single op | | | | | | | | | |
| Set Operators | | | R and S are union-compatible if: - R and S have same num of attrs - The corresponding attributes have the same or compatible domains | | | | | | | | | | | | | | | | | | | | | | | |
| Cross Product of 2 relations (R S). Let R(A1, A2, ..., An) and S(B1, B2, ..., Bn), then R S produce a schema (R S)(A1, A2, ..., An, B1, B2, ..., Bn)  Set of attributes in R and S must be disjoint, i.e. Attr(R) Attr(S) = . Note, size of cross product is |R| \* |S|. | | | | | | | | | | | | | | | | | | | | | | | |
| Inner Joins | Includes only tuples that satisfy the condition  If no common attributes, same as (empty set = vacuously true) | | | | | | | | : -join. R S = , where (any op) can use attributes that appears in R or S | | | | | | | | | | | | | | | | | |
| : equi join. Special -join where the only relational opearator that can be used is equality (= or ) | | | | | | | | | | | | | | | | | |
| : natural join. Special equi join: combine only if value of common attributes are equal, and output relation only show common attributes of R and S once | | | | | | | | | | | | | | | | | |
| Outer Joins | | | dangle(R S) = set of dangling tuples in R w.r.t R S | | | | | | | | | | | | null(R) = n-component tuple of NULL values (n = num of attributes in R) | | | | | | | | | | | |
| ⟕[θ]: left outer join. R ⟕[θ] S = R S (dangle(R S) {null(S)}) | | | | | | | | | | | | | | | | | | | | | | | |
| ⟖[θ]: right outer join. R ⟖[θ] S = R S ({null(R)} dangle(R S)) | | | | | | | | | | | | | | | | | | | | | | | |
| ⟗[θ]: full outer join. R ⟗[θ] S = R S (dangle(R S) {null(S)}) ({null(R)} dangle(R S)) | | | | | | | | | | | | | | | | | | | | | | | |
| Natural left outer join: ⟕. Natural right outer join: ⟖. Natural full outer join: ⟗ | | | | | | | | | | | | | | | | | | | | | | | |
| Complex Expressions | | | | | | | | (Strongly) Equivalence: both produce error OR both produce the same result | | | | | | | | | | | | | | | | Weak: if no error, then both have same result | | |
| Properties for both equivalence | | | | | | |  | | | | | | R S S R (diff col order)  R S S R (non-commutative) | | | | | | | | | R (S T) (R S) T  R (S T) (R S) T (associative) | | | | |
| Strong Equivalence | | | | | | (unless c uses only attributes in )  (unless c uses only attributes in R) | | | | | | | | | | | | | (unless )  R (S T) (R S) T (if c uses T and d uses R) | | | | | | | |
| Weak Equivalence | | | | | |  | | | | |  | | | | | | | | R (S T) (R S) T | | | | | | | |
| Strong (Tutorial) | | | | | |  | | | | | |  | | | | | | | R ((S) T) (R (S)) T (associative)  (R and S are union compatible) | | | | | | | |
| Data types | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | BOOLEAN | true/false | FLOAT8 | double precision 8-bytes floating point | CHAR(n) | Fixed-length char string | | INTEGER | signed 4-bytes int | NUMERIC[(p,s)] | Exact numeric of selectable precision | VARCHAR(n) | Variable-length char string | | DATE | year, month, day | TIMESTAMP | Date and time | TEXT | Variable-length char string | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Semantics | | | INSERT INTO: - either all inserted or none inserted - attrs can be specified out of order  - missing values are replaced w NULL (if allowed) or default (if specified) | | | | | | | | | | | | | | | | | | | | | DELETE FROM: - condition is optional (if unspecified, equivalent to always true); | | |
| Principle of Acceptance: perform op if condition evaluates to True (used in WHERE clause and relational algebra) | | | | | | | | | | | | | | | | | | | | | | | |
| Integrity Constraint | | | Principle of Rejection: Reject insertion if condition evaluates to False (used in integrity constraints)  IS [NOT] NULL: use , . IS [NOT] DISTINCT FROM: use , | | | | | | | | | | | | 1) NOT NULL. 2) UNIQUE. 3) PRIMARY KEY. 4) FOREIGN KEY. 5) General Constraint  Named Constraint: <attr> <type> CONSTRAINT <name> <constraint> | | | | | | | | | | | |
| Foreign Key Constraint | | | Specify behavior when data in referenced table is deleted or updated  1) NO ACTION: reject delete/update if it violates constraint (default)  2) RESTRICT: similar to NO ACTION but check of constraint cannot be deferred  3) CASCADE. 4) SET DEFAULT 5) SET NULL: updates FK of referencing tuples to NULL value | | | | | | | | | | | | | | | | | | | | | col1 TEXT DEFAULT 'string' ON DELETE SET DEFAULT ON UPDATE CASCADE | | |
| Modifying DB | | | ALTER TABLE <table> ADD CONSTRAINT <constraint> FK (col) REFERENCES <table2> (col);  ALTER TABLE <table> DROP CONSTRAINT <constraint> <col>  ALTER TABLE <table> ALTER COLUMN <col> TYPE <type>; | | | | | | | | | | | | | ALTER TABLE <table> ALTER COLUMN <col> SET DEFAULT <value>;  ALTER TABLE <table> ALTER COLUMN <col> DROP DEFAULT;  ALTER TABLE <table> ADD COLUMN <col> <type> DEFAULT <value>;  ALTER TABLE <table> DROP COLUMN <col>; | | | | | | | | | | |
| Deferrable Constraint | | | | Default: constraints check immediately at end of statement  True even for transaction containing multiple SQL statement  Relaxed Constraint: check only at end of transaction - for UNIQUE, PK, FK | | | | | | | | | | | | | | | | ALTER TABLE ... FK ... (col) NOT DEFERRABLE  ALTER TABLE ... FK ... (col) DEFERRABLE INITIALLY DEFERRED  ALTER TABLE ... FK ... (col) DEFERRABLE INITIALLY IMMEDIATE | | | | | | |
| NOT DEFERRABLE (default for foreign key)  BEGIN; -- start of transaction  ...  COMMIT; -- end of transaction  #transaction aborted as 1st insert statement fail due to foreign key constraint violated | | | | | | | | | | INITIALLY DEFERRED  BEGIN; -- start of transaction  ...  COMMIT; -- end of transaction  #transaction successful (constraint check at commit) | | | | | | | | | | | INITIALLY IMMEDIATE  BEGIN;  SET CONSTRAINTS <constraint> DEFERRED  ... COMMIT; -- end of transaction  #transaction successful (same as not deferrable but can manually override) | |
| Sql syntax | | | INSERT INTO <table\_name> [(<attr>, ...)] VALUES (<values>, ...) [, (<values>, ...)]; | | | | | | | | | | | | | | | | | | DELETE FROM <table\_name> [WHERE <condition>]; | | | | | |
| UPDATE <t1> SET <attr> = <value> [, <attr> = <value>] [WHERE <condition>]; | | | | | | | | | | | | | | | | | | DROP TABLE [IF EXISTS] <t1> [, <t2> [...]]] [CASCADE]; | | | | | |
| Integrity constraints | | | | | |  |  |  |  | | --- | --- | --- | --- | | Type | column\_constraint | table\_constraint | Condition | | Not-NULL | NOT NULL | - | IS NOT NULL | | Unique | UNIQUE | UNIQUE (A1, A2, ...) | x.Ai <> y.Ai | | Primary Key | PRIMARY KEY | PRIMARY KEY (A1, A2, ...) | UNIQUE & NOT NULL | | Foreign Key | REFERENCES R1 (B) | FK (A1, A2, ...) REFERENCES R1 (B1, B2, ...) | Tuple exists in R1 as PK or contains NULL value | | General | CHECK (c) | CHECK (c) | Condition c does not evaluate to False | | | | | | | | | | | | | | | | | | | | | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Entity Sets (rectangle, nouns) | | Attributes are specific info describing an entity (oval)  1. Key Attributes (underlined): uniquely identify entity  2. Composite Attrs (composed of other ovals): composed of other attrs | | | | | 3. Multivalued Attributes (double-lined): 1 or more attr for a given entity  4. Derived Attributes (dashed line): derived from other attrs | |
| R/s sets | | (diamond, verbs). May have own attr. Degree/arity = num of entity sets involved. - n-ary = n entity. Binary r/s set (n = 2); Ternary (n = 3) | | | | | | |
| R/s Constraints | | 1. Cardinality Constraints: upper bound (either 1 or ∞) - Upper limit of 1 = key constraints (). Default is ∞ (---)  2. Participation Constraints: lower bound (either 0 or 1) - Lower limit of 0/1 = partial/total participation constraints. (---/===)  3. Dependency constraints: Entity set that does NOT have its own key (its keys are called partial keys) | | | | | | |
| Dependency | | Weak Entity Set: Entity set that does NOT have its own key. [[weak entity]] ==> <<>> --- []  - Must have total participation constraint in identifying r/s. Many-to-one r/s from weak entity set to owner entity set  Partial Key: set of attributes of weak entity set that uniquely identifies a weak entity for a given owner entity | | | | | | |
| ER diagram to Schema | | ER diagram limitations: - don't capture data type - many constraints cannot be encoded except w complicated tricks (NOT NULL: use unary; UNIQUE: one-to-one) - General constraint cannot be encoded  Key: ER diagram can only encode 1 set of key for each entity set (exception one-to-one r/s)  Multivalued: 1) Convert multivalued attr into set of single-valued attr (cons: can only have fixed num of attr)  2) Create additional table w foreign key constraint (cons: retrieval requires querying 2 tables)  Things to consider: 1) Can PK uniquely identify rest of attributes? 2) Lower bound? 3) Upper bound? | | | | | | |
| Many-to-Many – – | | | For r/s, CREATE TABLE for r/s but use FK to REFERENCE PK in Entity Table. PK = all PK of entity | | | | | |
| Many-to-One – <-  (many users, 1 booking) | | | | 1) For r/s, CREATE TABLE for r/s but use FK to REFERENCE PK in Entity Table. PK = PK of unique entity (i.e booking here) | | | | |
| 2) Combine r/s w unique entity (i.e. combine makes and booking). PK = bid. FK REFERENCES Users | | | | |
| One-to-One  –> <– | | 1) Combine r/s with either table. Then FK UNIQUE REFERENCES other entity. | | | | | | |
| 2) For r/s, CREATE TABLE for r/s but use PK REFERENCES 1 entity, FK UNIQUE NOT NULL REFERENCES other entity | | | | | | |
| Key + Total Participation – <= | | | | | | Combine r/s with unique entity. PK = PK of unique entity. FK NOT NULL REFERENCES other entity | | |
| Weak Entity – <<>> <= | | | | | Combine weak + r/s. PK = PK of weak + PK of owner. FK REFERENCES owner ON DELETE CASCADE ON UPDATE CASCADE | | | |
| ISA Hierachies | | 'is a' r/s. Superclass above, and subclass below. Attr already shown in superclass should not be shown again in subclass  Overlap Constraint: Can a superclass belong to multiple subclasses? Covering Constraint: Must a superclass belong to at least 1 subclass?  Cover F; Over F; -->. Cover F; Over T; --- (dashed). Cover T; Over F; ==>. Cover T; Over T; ===.  Cover T = total participation constraint. Over F = key constraint  SQL: FK in subclass table REFERENCES PK in immediate superclass ON DELETE CASCADE  If subclass can belong to 2 superclass (use col1 <type> PK REFERENCES superclass1 REFERENCES superclass2) | | | | | | |
| Aggre-gation | Want to capture r/s btw entity sets & r/s sets (for some proj doned by students, GPU may be used)  Aggregate is a rectangle (entity set) around the diamond (r/s set)  Note rectangle should not touch diamond. Entity sets forming the aggregate touches the diamond  When using the aggregate as an entity set, the line touches the rectangle  FK constraint: (Works.sid) (Students.sid). (Works.name) (Projects.name) | | | | | | | CREATE TABLE Uses (  gid INT REFERENCES GPUs,  sid INT, name VARCHAR(50),  PRIMARY KEY (gid, sid, name),  FK (sid, name) REFERENCES Works); |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SQL vs RA | | By default, duplicates are not eliminated (use DISTINCT to remove duplicates). Conditions based on principle of acceptance  Order-independent: Order of rows in results does not matter. For CS2102, assume output should not contain duplicate & is order-indep | | | | | | | |
| SELECT | | Complex condition: use OR and AND. Operators: {=, <>, <, <=, >=, >}  String op: {|| (concatenate), LOWER(s), UPPER(s)}. Datetime: {+, NOW(), ...} | | | | Mathematical op: {+, \*, -, /, ^, |/ (√), ...} | | | |
| DISTINCT keyword check for distinct rows using IS DISTINCT FROM (i.e. ), opp is IS NOT DISTINCT FROM () | | | | | | | |
| WHERE | | Pattern matching ([NOT] LIKE): \_ (any character); % (0 or more characters). | | | | | WHERE col IS NULL AND col LIKE 'abc%yz' | | |
| Set Ops | | UNION, INTERSECT, EXCEPT (set difference) where both tables are union-compatible (remove duplicate rows, col names follow left table)  use UNION ALL, INTERSECT ALL, EXCEPT ALL to keep duplicate rows (where each element is now treated as distinct element) | | | | | | | |
| Join Ops | JOIN refers to INNER JOIN, but INNER is optional  - NATURAL JOIN (auto join on common attributes) | | | - LEFT JOIN Table2 t2 ON t1.col1 = t2.col2 #LEFT JOIN same as LEFT OUTER JOIN  - RIGHT [OUTER] JOIN, FULL [OUTER] JOIN, NATURAL LEFT JOIN (no need specify ON) | | | | | |
| Subqueries | | | Subqueries are queries within another queries: Can appear in SELECT, FROM and WHERE clause  Subquery expressions: IN/NOT IN, EXISTS/NOT EXISTS, ANY (equivalent to SOME in other DBMS), ALL | | | | | | |
| Scalar subquery | | Scalar subquery = query that returns at most a single value (i.e. table with ≤ 1 row and 1 col)  - Dynamically checked at run time, can be used as a value in queries. If result return 0 row, it is treated as NULL | | | | | | | |
| SELECT (SELECT col1 FROM Table1 t1 WHERE t1.col2 = t2.col2), col3  FROM Table2 t2; | | | SELECT \* FROM Table1 t1  WHERE col1 < (SELECT t2.col1 FROM Table1 t2 WHERE ... AND ...); | | | | |
| [NOT] IN | | WHERE <expr> [NOT] IN [<subquery> | <tuple>];  - Subquery must return exactly 1 col. IN return TRUE if row <subquery>: row = <expr>  NOT IN return TRUE if row <subquery>: row <> <expr>  - Tuple specification is (value1, value2, ...). IN vacuously F, NOT IN vacuously T | | | | | | | SELECT DISTINCT col1 FROM Table1  WHERE col2 IN (SELECT col3 FROM Table2 WHERE col4 = 'abc'); |
| ANY / SOME & ALL | | WHERE <expr> <op> ANY <subquery>; #same for AND  ANY returns TRUE if row <subquery>: (<expr> <op> row) = TRUE  ALL retuns TRUE if row <subquery>: (<expr> <op> row) = TRUE  If <subquery> has no rows: ANY always FALSE; ALL always TRUE | | | | | | | SELECT DISTINCT col1 FROM Table1  WHERE col2 < ANY (SELECT col2  FROM Table2 WHERE col3 = 'abc'); |
| [NOT] EXISTS | | WHERE [NOT] EXISTS <subquery>; SELECT 1 (note 1 is just a placeholder)  Subquery may return any num of cols. EXISTS returns TRUE if subquery returns at least 1 row  NOT EXISTS returns TRUE if subquery returns no row | | | | | | SELECT DISTINCT col1 FROM Table1 t1  WHERE EXISTS (SELECT 1 FROM Table2 t2  WHERE t1.col2 = t2.col2); | |
| Remarks | | WHERE <expr> IN <subquery> is equivalent to WHERE <expr> = ANY <subquery>  WHERE col1 <op> ANY (SELECT col2 FROM t1WHERE <cond>) WHERE EXISTS (SELECT \* FROM t1 WHERE <cond> AND col1 <op> col2) | | | | | | | |
| Ordering | | SELECT ... FROM ... WHERE ... ORDER BY col1 DESC, col2 ASC; (col1 sorting first)  LIMIT k: return 1st k rows. OFFSET i: show rows starting from i+1. Can combine: ORDER BY ... OFFSET 3 LIMIT 3; | | | | | | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Aggregation | |  |  |  |  | | --- | --- | --- | --- | | Query  R(A); | Interpretation | If R is empty relation | S is non-empty relation with attr A that only has NULL values | | MIN(A) /MAX(A) | Min non-NULL value in A | NULL | NULL | | AVG/SUM/COUNT([DISTINCT] A) | Avg/sum/count of [distinct] non-NULL values | NULL | NULL | | COUNT(\*) | Count of rows in A | 0 | n | | | | | | | | | | | |
| Grouping | GROUP BY attr1, attr2, .... GROUP BY w/o aggregation is valid but not meaningful. Aggregation fn is applied over each group.  2 tuples are in the same group if (t.attr1 IS NOT DISTINCT FROM t'.attr1) AND (t.attr2 IS NOT DISTINCT FROM t'.attr2) ... evaluate to TRUE  Restriction: If col Ai of table R appears in SELECT clause, one of the following must hold:  1. Ai appear in GROUP BY. 2. Ai appears as input of an aggregation fn in the SELECT clause 3. PK of R appears in the GROUP BY | | | | | | | | | | |
| HAVING | GROUP BY attr1, attr2, ... HAVING <condition>. HAVING must be used w GROUP BY. <condition> usually an aggregate fn  Restriction: If col Ai of table R appears in HAVING clause, one of the following must hold:  1. Ai appear in the GROUP BY. 2. Ai appears as input of an aggregation fn in the HAVING clause 3. PK of R appears in the GROUP BY | | | | | | | | | | |
| Evaluation | FROM WHERE GROUP BY HAVING SELECT ORDER BY LIMIT/OFFSET | | | | | | | | | | |
| Conditional | Only 1 of the result is returned  If <expression> is specified, similar to <condition> being defined as <expression> = <value>  ELSE is optional: return NULL if no ELSE specified | | | | | CASE  WHEN <condition1> THEN <result1>  WHEN <conditionn> THEN <resultn>  ELSE <result0> END | | | | | CASE <expression>  WHEN <value1> THEN <result1>  WHEN <valuen> THEN <resultn>  ELSE <result0> END |
| Coalesce | COALESCE(<value1>, <value2>, ...) | | | | | | | Returns the 1st non-NULL value in list of input arguments | | | |
| NULLIF | NULLIF(<value1>, <value2>) | | | | | | | Return NULL if <value1> = <value2>, else return <value1> | | | |
| Table Expressions | SELECT \* FROM (SELECT \* FROM Table1) T1; | | | | | | | "Creates" a temporary table ( no actual table created) | | | |
| WITH CTE1 AS (Q1),  CTEn AS (Qn)  Q0 | | | Each CTEi is name of temp table defined by query Qi  Each CTEi can reference any other CTE declared before it | | | | | | Main SELECT statement Q0 can reference any subset of all CTEi  Col name can be specified as CTE(attr1, attr2) AS (Q) | |
| VIEWS | Same table but diff views for diff users  A VIEW is a permanent named query (no actual table created) | | | | | | | CREATE VIEW <name> AS <query>;  Can specify col name: CREATE VIEW <name> (attr1, attr2, ...) AS ... | | | |
| SELECT Statements: no restrictions when used in SQL queries | | | Updatable View: Restrictions for INSERT, UPDATE and/or DELETE statements:  - Only 1 entry in FROM clause, - No WITH, DISTINCT, GROUP BY, HAVING, LIMIT, OFFSET  - No UNION, INTERSECT, EXCEPT or ALL variants, - No aggregate fns | | | | | | | |
| Universal Quantifica-tion (Double Negation) | | E.g. restaurant that sells all pizzas liked by 'Homer' there DNE pizza that 'Homer' likes and not sold by restaurant | | | SELECT DISTINCT S1.rname FROM Sells S1  WHERE NOT EXISTS (SELECT 1 FROM Likes L WHERE L.cname = 'Homer'  AND NOT EXISTS (SELECT 1 FROM Sells S WHERE S.pizza = L.pizza  AND S.rname = S1.rname)); | | | | | | |
| Universal Quantification (Cardinality) | | | Superset: R S R S = R |R S| = |R|  Equal: R S R S R S = R |R S| = |R S|  Subset: R S R S = R |R S| = |R| | | | | SELECT DISTINCT rname FROM Sells S  WHERE (SELECT COUNT(DISTINCT pizza) FROM (Q1 UNION Q2) = (SELECT COUNT(DISTINCT pizza) FROM Sells S1 WHERE S1.rname = S.rname); | | | | |
| Recursive Queries | WITH RECURSIVE CTE\_name AS (  Q\_1 UNION [ALL] Q\_2 (CTE\_name))  Q\_0 (CTE\_name)  Q1 is non-recursive. Q\_2 is recursive and can reference CTE\_name | | | | | | | | WITH RECURSIVE Function(to\_stn, stops) AS (  Q1 UNION ALL SELECT ... FROM Function F, MRT M  SELECT DISTINCT (to\_stn) FROM Function; | | |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PL/pgSQL Functions | CREATE OR REPLACE FUNCTION convert (mark INT) //input  RETURNS char(1) AS $$ //output  SELECT CASE  WHEN mark >= 70 THEN 'A'  ELSE 'D' END;  $$ LANGUAGE sql;  SELECT convert(90); OR SELECT Name, convert(Mark) FROM Table; | | | | | CREATE OR REPLACE FUNCTION topStudent() RETURNS Scores AS $$ //return only 1 tuple w same schema as Scores  SELECT \* FROM Scores LIMIT 1;  $$ LANGUAGE sql; | | | | | |
| //return multiple tuples w same schema as Scores  CREATE OR REPLACE FUNCTION topStudent() RETURNS SETOF Scores AS $$ SELECT \* FROM Scores;  $$ LANGUAGE sql; | | | | | |
| #return 1 tuple w diff schema  CREATE OR REPLACE FUNCTION fn (OUT var1 INT, OUT var2 INT)  RETURNS RECORD AS $$  ...; $$ LANGUAGE sql; | | | | | | #returns multiple tuples w diff schema  1) use RETURNS SETOF RECORD AS | | | | |
| 2) CREATE OR REPLACE FUNCTION MarkCnt ()  RETURNS TABLE(Mark INT, Cnt INT) AS $$ ...; $$ LANGUAGE sql; | | | | |
| SQL Procedures | | No return tuple needed  CALL transfer('Alice', 'Bob', 100); | | CREATE OR REPLACE PROCEDURE transfer (fromAcct TEXT, toAcct TEXT, amount INT) AS $$  UPDATE Table SET ...; $$ LANGUAGE SQL | | | | | | | |
| Control | | IF ... THEN ... ELSE ... END IF | LOOP ... END LOOP | | EXIT ... WHEN ... | | | WHILE ... LOOP ... END LOOP | | | FOR ... IN ... LOOP ... END LOOP |
| Cursor | | CREATE OR REPLACE FUNCTION score\_gap()  RETURNS TABLE ( name TEXT, mark INT, gap INT ) AS $$  DECLARE  curs CURSOR FOR (SELECT \* FROM Scores ORDER BY Mark DESC);  BEGIN prv\_mark := -1; OPEN curs;  LOOP  FETCH curs INTO r;  EXIT WHEN NOT FOUND; name := r.Name; mark := r.Mark;  RETURN NEXT;  END LOOP; CLOSE curs;  END; $$ LANGUAGE plpgsql; | | | | | | | | Access each individual row returned by a SELECT statement  #declares cursor variable  #point cursor to beginning of result  #read tuple from curs and put in r  #when no more tuple to be read  #insert a tuple(name, mark, gap) to output of function as defined earlier | |
| also have FETCH PRIOR/FIRST/LAST FROM cur INTO r  FETCH ABSOLUTE n FROM cur INTO r (fetch nth tuple) | |
|  | | To prevent SQL injection, use function/procedures or prepared statements | | | | | | | So anything in input as fn args is treated as a string constant | | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Triggers | | CREATE OR REPLACE FUNCTION func() RETURNS TRIGGER AS $$  BEGIN INSERT INTO Scores\_Log(Name, EntryDate) VALUES (NEW.Name, CURRENT\_DATE);  RETURN NULL;  END; $$ LANGUAGE plpgsql  CREATE TRIGGER scores\_trigger AFTER INSERT ON Scores  FOR EACH ROW EXECUTE FUNCTION func(); | | | | | CURRENT\_DATE: built in fn in SQL  #once insert into scores  #call func() for each insertion |
|  | | CREATE OR REPLACE FUNCTION scores\_func() RETURNS TRIGGER AS $$  BEGIN IF (TG\_OP = 'INSERT') THEN  INSERT INTO Scores\_Log2 SELECT NEW.Name, 'Insert', CURRENT\_DATE; RETURN NEW;  ELSEIF (TG\_OP = 'DELETE) THEN .....  CREATE TRIGGER scores\_log2\_tigger AFTER INSERT OR DELETE OR UPDATE ON Scores  FOR EACH ROW EXECUTE FUNCTION scores\_func(); | | | | | NEW, TG\_OP (operation that activates trigger),  TG\_TABLE\_NAME (name of table that activated trigger),  OLD (for insert, OLD is NULL) |
| Before Trigger | | CREATE OR REPLACE FUNCTION for\_Elise\_func() RETURNS TRIGGER AS $$  BEGIN IF (NEW.Name = 'Elise') THEN NEW.Mark := 100; END IF; RETURN NEW;  END; $$ LANGUAGE plpgsql;  CREATE TRIGGER for\_Elise\_trigger BEFORE INSERT ON Scores  FOR EACH ROW EXECUTE FUNCTION for\_Elise\_func(); | | | | #Elise mark always 100  #if RETURN NULL, then no tuple inserted  #as long as return non-null tuple, trigger would use it to insert | |
| Returns Values | | |  |  |  |  | | --- | --- | --- | --- | |  | INSERT | UPDATE | DELETE | | BEFORE | return non-null tuple t: t inserted  null: no tuple inserted | return non-null t: t is updated  null: no tuple updated | non-null t; deletion proceed as normal  null: no deletion performed | | AFTER | return value don't matter (since trigger invoked after main operation is done) | | | | | | | | |
| INSTEAD OF | | Only for views (when someone want to operate on a view -> trigger operate on main table and update view) | | | | | |
| CREATE OR REPLACE FUNCTION update\_func() RETURNS TRIGGER AS $$  BEGIN UPDATE Scores SET Mark = NEW.Mark WHERE Name = OLD.Name;  RETURN NEW; END; $$ LANGUAGE plpgsql;  CREATE TRIGGER update\_max\_trigger INSTEAD OF UPDATE ON Max\_Score  FOR EACH ROW EXECUTE FUNCTION update\_max\_func(); | | | | | #return non-null: proceed as normal  #return NULL: ignore rest of operation on current row |
| Trigger Levels | FOR EACH ROW: execute trigger function for every tuple encountered | | | FOR EACH STATEMENT: execute trigger function only once | | | |
| CREATE ... FUNCTION del\_warn\_funct() RETURNS ...  BEGIN RAISE NOTICE 'You cannot delete from table'; RETURN NULL;...  CREATE TRIGGER del\_warn\_trigger  BEFORE DELETE ON Scores\_Log  FOR EACH STATEMENT EXECUTE FUNCTION del\_warn\_func() | | #Statement level trigger ignore value returned by fn  #So RETURN NULL would not make db ignore subsequent ops  #So instead of RAISE NOTICE (prompt users), use RAISE EXCEPTION  #BEFORE/AFTER allowed on both ROW and STATEMENT  #INSTEAD OF only for ROW | | | | |
| Trigger Condition | | CREATE TRIGGER for\_Elise\_trigger BEFORE INSERT ON Scores  FOR EACH ROW WHEN (NEW.Name = 'Elise') EXECUTE FUNCTION for\_Elise\_func()  CREATE ... FUNCTION for\_Elise\_func() ... BEGIN NEW.Mark := 100; RETURN NEW;... | | | #checking condition moved to trigger  #WHEN requirements: No SELECT/OLD for INSERT/NEW for DELETE, Not for INSTEAD OF | | |
| Deferred Trigger | | CREATE CONSTRAINT TRIGGER bal\_check\_trigger  AFTER INSERT OR UPDATE OR DELETE ON Account  DEFERRABLE INITIALLY DEFERRED FOR EACH ROW EXECUTE FUNCTION bal\_check\_func(); | | INITIALLY DEFFERED: trigger deferred by default  INITIALLY IMMEDIATE: trigger not deferred by default  Deferred triggers only work with AFTER and FOR EACH ROW | | | |
| Multiple Triggers | | Order of trigger activation: BEFORE statement level, BEFORE row-level, AFTER row-level, AFTER statement-level  Within each category, triggers activated in alphabetic order  If a BEFORE row-level trigger returns NULL, all subsequent triggers on same row are omitted | | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Functional Dependency (FD) | Let A1, ..., Am, B1, ..., Bn be some attributes. A1...Am B1...Bn if : whenever 2 objs have same values on Ai, they have same value on Bj  i.e. same input cannot map to diff outputs. Notation: with / across = not funtionally dependent  A -> B is trivial iff B ⊆ A. A -> B is non-trivial iff B ⊈ A. A -> B is completely non-trivial iff B ≠ ∅ and B ∩ A = ∅ | | | | | |
| Armstrong's Axioms | Reflexivity: AB A  Augmentation: If A B, then AC BC | | Transitivity: If A B and B C, then A C  Decomposition: If A BC, then A B and A C | | | Union: If A B and A C, then A BC |
| Closure | To prove X Y (does not) holds, just show {X}+ (does not) contains Y | | | | E.g. A B, C D, BC D. {A}+ = {A, B}, {A, C}+ = {A, B, C, D}, ... | |
| Prime attributes | Prime attribute: attr that appears in a key (else: non-prime)  To find keys: - consider every subset of attributes in; derive closure; | | | - check small attribute sets first  - check which attr not in RHS of FD (i.e. attr must be in key) | | |
| Non-trivial & Decom-posed FD | Decomposed FD: an FD whose RHS has only 1 attr  A non-decomposed FD can always be transformed into a set of decomposed FD (by Decomposition)  Non-trivial & Decomposed FD: a decomposed FD whose RHS does not appear on the LHS | | | | | |
| BCNF | A table R is in BCNF, if every non-trivial and decomposed FD has a superkey as its LHS | | | | | |
| BCNF Check | Check if there is a closure {A1...Ak}+ s.t. it is a "more but not all" closure. If such a closure exists, then R is not in BCNF | | | | | |
| BCNF Decomposition / Normalization | | Algo: 1. Find a subset X of attrs in R, s.t. {X}+ satisfy "more but not all" property.  2. Decompose R into 2 tables R1 and R­2 s.t. (R1 contains all attrs in {X}+) and (R2 contains all attrs in X and attrs not in {X}+)  3. If R1/R2 not in BCNF, further decompose R1/R2. | | | | |
| BCNF decomposition of a table may not be unique | | If table has only 2 attrs, then is must be in BCNF | | |
| Properties of BCNF | | Good: No update/deletion/insertion anomalies. Small redundancy. Original table can be reconstructed from decomposed tables | | | | |
| Bad: Dependencies may not be preserved. | | | | |
| Lossless Join Decom-position | Decomposition guarantees lossless join, whenever the common attrs in R1 and R2 constitute a superkey of R1 or R2  (e.g. SELECT \* FROM R1, R2 WHERE R1.attr1 = R2.attr1)  BCNF Decomposition: X is the set of common attrs btw R1 and R2. X is a superkey of R1 | | | | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Dependency Preservation | | 2 sets of FDs, S and S' are equivalent if every FD in S' can be derived from S and vice versa. Just check {LHS}+ RHS can be derived  Preserving FDs makes it easier to avoid update anomalies | | | | | | |
| Third Normal Form (3NF) | | | A table is in 3NF iff for every non-trivial and decomposed FD: either the LHS is a superkey or RHS is a prime attribute  Check: compute closure for all subset of attr, find keys, check defn | | | | | |
| 3NF Decomposi-tion | | BCNF decomposition may perform 1 or more binary splits (split table into 2)  3NF decomposition only has 1 split, dividing table into 2 or more smaller tables | | | - Derive minimal basis of S - In minimal basis, combine FDs whose LHS are the same  - Create table for each FD remaining  - If none of the table contains a key of original table, create a table that contains a key  - Remove redundant tables | | | |
| Minimal basis | Let S be a set of FDs. Its minimal basis M is a set of FDs, s.t.  1. every FD in S derivable from M, and vice versa (check using closure)  2. every FD in M is a non-trivial and decomposed FD | | | | | | 3. if any FD removed from M, some FD in S can't be derived from M  4. for any FD in M, if we remove an attr from its LHS, then the FD cannot be derived from S | |
| Algo for minimal basis | | | | 1. Transform FDs, s.t. RHS contains only 1 attr | | 2. Remove redundant attrs on LHS of all FDs | | 3. Remove redundant FDs |