|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Definitions | | | Database: collection of organized data, information and records | | | | | | |
| Challenges | | | Efficiency: fast access to information in huge volumes of data  Transactions: "All-or-nothing" changes to data | | | | Data Integrity: parallel access and changes to data  Recovery: fast and reliable handling of failures  Security: fine-grained data access rights | | |
| Data Management | | | File-Based: complex, low-level code. Have similar requirements across diff programs | | | | | High development effort, long development time, higher risk of errors | |
| DBMS: application logic moved to DBMS. Other set of universal and powerful functionalities | | | | | Faster application development, increased productivity, higher stability | |
| Transactions | | | - finite seq of database ops and constitutes the smallest logical unit of work from the application perspective  ACID property of transaction T:  Atomicity: either all effects of T reflected of none. Consistency: T guarantess to yield the correct state of the database  Isolation: T is isolated from the effects of concurrent transactions. Durability: after a commit of T, its effects are permanent  Transition graph: initial state -> begin T -> internal state -> commit T -> result state  Internal state <-> error state -> abort T -> initial state. Commit T -> error state -> abort T -> initial state  Only see the initial and result state as users. DBMS guarantees that state we see is consistent | | | | | | |
| Sequential vs Concurrent transactions: | | Concurrent transactions might result in lost update, dirty read, unrepeatable read | | | | |
| Serialization | | | Equivalent Transaction: 2 executions are equivalent if they have the same effect on the data  Serializability: concurrent execution of a set of transaction is serializable if this execution is equivalent to some serial execution of the same set of transactions | | | | | DBMS - support concurrent executions of transactions to optimize performance  - Enforce serializability of concurrent executions to ensure the integrity of data | |
| Architecture | | | Data Abstraction:  1) External Schema – user or group-specific view on the data  2) Logical Schema – logical organization of data (relations/tables, graphs); unified representation of all data; support of logical and physical data independence  3) Physical Schema – organization of data on disk and in memory; database as collection of fields, arrays, records, files, pages, etc | | | | | | |
| Logical Data Independence: ability to change logical schema w/o affecting external schemas (adding/deleting/updating attributes, changing data types, changing data model)  Physical Data Independence: - Representation of data being independent from physical schema  - Physical schema can be changed w/o affecting logical schema (creating indexes, new caching strategies, diff storage devices) | | | | | | |
| Relational Model | | | Relation Schema specifies the attributes (columns) + data constraints (domain constraints) + relation name  Domain = set of atomic values (int, numeric, text). Domain of attribute Ai is dom(Ai).  Each value v of attribute Ai is either: v dom(Ai) or v = NULL  Relation/table = set of tuple/records/row. Simplified notation of relation schema where domain is clear from context is R(A1, A2,..., An)  Instance of schema R = relation which is a subset of {(a1, a2, ..., an)|ai dom(Ai) {NULL}}  R.Ai refer to attribute Ai of relation R  Cardinality = number of rows  Note theoretically, relation/table cannot contain duplicate rows. But SQL allows this, then relation becomes a multiset.  Relational Database Schema = set of relation schemas + data constraints  Relation Database = collection of relations  DBS = DBMS + (n \* DB) (n > 0), DBS = database systems, DB = database, DBMS = database management system | | | | | | |
| Data Integrity | Integrity constraints = conditions that restrict what ocnstitutes valid data  Structural: - indep of application, - domain constraints (correct data type), - key constraints, - foreign key constraints  General: - dependent of application, - check constraints (e.g. int > 0), - triggers | | | | | | | | |
| Key | Superkey = subset of attributes that uniquely identifies a tuple in a relation  Key = superkey that is also minimal (i.e. no proper subset of the key is a superkey)  Candidate keys = set of all keys of a given relation  Primary key = selected candidate key (cannot be repeated)  - primary key constraint: attributes cannot be NULL. Key(Schema) = (...)  In Schema notation: Table(id: **INT**, title: **TEXT**) (underlined = primary key)  Prime Attribute = attributes of a candidate key | | | | | If (A,B,C) is definitely a superkey, then (A,B,C,D) is a superkey  If (A,B,C) is definitely a key, then (A,B,C) 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 | | Foreign key = subset of attributes of relation R1 that refers to the primary key of relation R2  R1 is the referencing relation; R2 is the referenced relation  We write (R1.Ai1, R1.Ai2, ...) (R2.Aj1, R2.Aj2, ...)  - foreign key constraint: each foreign key in R­1 must either appear as a primary key in R2 or be a NULL value (or a tuple containing at least 1 NULL value) | | | | | | | Referencing relation and referenced relation can be the same relation |
| Summary | | Advantage of DBMS over files system  - transactions w ACID properties  - level of abstraction for data indep | | Relational Model  - Unified representation of all data as relations/tables  - (Structural) integrity constraints to specify restrictions on what constitutes correct/valid data | | | | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Relational Algebra | | Algebra is a mathematical system consisting of operands and operators  Relational Algebra: operands = relations (or vars representing relations);  operators = transformation from ≥ 1 input relations into 1 output relation | | | | | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Unary | Selection | | Projection | | Renaming | | | Binary | Cross product | union | | intersection | | difference – | | | |
|  | | Algebra allows for query optimization (rewrite query to another equivalent 1, and can be evaluated faster)  Closure Property: set of values is closed under set of operators if any combination of the operators produces only values in the given set  Thrm: Relations are closed under Relational Algebra  Implication: can chain operations as all inputs and outputs are relations (output of 1 operator can be input of another) | | | | | | | |
| Three-Valued Logic – Logical | | | Boolean values = {True, False, NULL}   |  |  |  |  | | --- | --- | --- | --- | | a | False | NULL | True | | a | True | NULL | False | | | |  |  |  |  |  | | --- | --- | --- | --- | --- | | a b | | a | | | | False | NULL | True | | b | False | False | False | False | | NULL | False | NULL | NULL | | True | False | NULL | True | | | | | |  |  |  |  |  | | --- | --- | --- | --- | --- | | a b | | a | | | | False | NULL | True | | b | False | False | NULL | True | | NULL | NULL | NULL | True | | True | True | True | True | |
| Three-Valued Logic – Relational & Arithmetic | | | Any relational/arithmetic operation w NULL produces NULL values   |  |  | | --- | --- | | Operation | Result | | NULL < 0 | NULL | | NULL + 1 | NULL | | NULL = NULL | NULL | | NULL <> NULL (<> : not equal) | NULL | | | | | Special operators to treat NULL as values   |  |  |  |  | | --- | --- | --- | --- | | a | b | a b | a b | | NULL | NULL | True | False | | x | NULL | False | True | | NULL | y | False | True | | x | y | x = y | x <> y | | | |
| Unary Ops | Selection: select all tuples from r that satisfy c  , where c is a condition that returns a boolean value, R is relation  Condition c must specify only attributes in R | | | | | | Operator Precedence: (), **op**, , ,  with **op** {=, <>, <, ≤, ≥, >, , } | | |
| Projection: keeps only the column/project specified in and in that order  , where is an ordered list of attributes  Order list of attributes must specify only attributes in R | | | | | | For simplicity, do not allow operations (e.g. )and duplicates (e.g. )on | | |
| Renaming: renames attributes listed in  , where is a collection of attribute renaming which is either:  - No 2 diff attributes may be renamed to the same name  - No attributes may be renamed more than once in a single op | | | | | 1. (order of attribute don't matter, new old) 2. (order of attribute matter; must specify all attributes, if not renamed, set new = old)   Follow option 1, (unless renaming many cols) | | | |
| Set Operators | | Union Compatibility: 2 relations R and S are union-compatible if: - R and S have same num of attributes  - The corresponding attributes have the same or compatible domains | | | | | | | |
| Cross Product of 2 relations (R S) is a relation formed by combining all pairs of tuples from the 2 input relations  Let R(A1, A2, ..., An) and S(B1, B2, ..., Bn), then R S produce a schema (R S)(A1, A2, ..., An, B1, B2, ..., Bn)  (R S) = {(a1, ..., an, b1, ..., bn) | (a1, ..., an) R (b1, ..., bn) S}  Set of attributes in R and S must be disjoint, i.e. Attr(R) Attr(S) =  Note, size of cross product is |R| \* |S|. Often also require selection/projection. Hence join operator created. | | | | | | | |
| Inner Joins | | Includes only tuples that satisfy the condition  If no common attributes, same as (empty set = vacuously true) | | : -join. R S = , where can use attributes that appears in R or S  -join can use any arbitrary relational operator (=, <>, <, ≤, ≥, >, , ) | | | | | |
| : equi join. Special -join where the only relational opearator that can be used is equality (= or )  Can use hash tables to speed up equi join | | | | | |
| : natural join. Special equi join where we combine only if value of common attributes are equal, and output relation only show common attributes of R and S once  Let R(A1, ..., Ai, B1, ..., Bj) and S(B1, ..., Bj, C1, ..., Ck).  R S = {(a1, ..., ai, b1, ..., bj, c1, ..., ck) | (a1, ..., ai, b1, ..., bj) R (b1, ..., bj, c1, ..., ck) S}  OR, R S = , where = Attr(R) Attr(S) and = | | | | | |
| Outer Joins | | Include tuple that satisfy and some that don't satisfy condition (dangling tuple) | | 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 Outer Joins (analogous to natural inner join): condition not specified & equality op performed over common attributes of R and S  Natural left outer join: ⟕. Natural right outer join: ⟖. Natural full outer join: ⟗ | | | | | |
| Complex Expressions | | | Can define intermediate table. e.g. M := , N := , P := M N | | | | | Operator Tree: work from bottom up | |
| (Strongly) Equivalence: 2 relational algebra expressions Q1 and Q2 are equivalent (Q1 Q2) if for any input relations either: both produce error OR both produce the same result  Weakly equivalent: if there is no error, then both produce the same result  Multiplicity: multiple ways to formulate query to get same result (order of join, order of selection, having add projections)  Query optimization: find "best" relational algebra expression (handled by DBMS transparent to user) | | | | | | |
| 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) | | |
| Invalid expr | | | Attribute no longer available after projection:  Attribute no longer available after renaming: | | | | Incompatible attribute types: (but c and d of diff types; assume no implicit type conversion) | | |
| Redundant expr | | | Cross product + attribute selection join:  R  Unnecessary ops: | | | | Unoptimized query: | | |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SQL | | Structured Query Language. Is a domain-specific & declarative language (focus on what to compute, instead of how)  Unlike python which is general purpose, & relational algebra expression which is imperative  Based on multi-set/bag (may contain duplicate rows) | | | | DDL (data definition): CREATE, ALTER, DROP, RENAME, TRUNCATE  DML (data manipulation): INSERT, UPDATE, DELETE, MERGE  DQL (data query): SELECT  TCL (transaction control): BEGIN, COMMIT, ROLLBACK, SAVEPOINT  DCL (data control): GRANT, REVOKE | | | | |
| 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 | | Document | XML, JSON | Spatial | Point, Line, Polygon, Circle, Box, Path | Special | Money, MAC/IP Address | | | | | | | | | | |
| Semantics | | For INSERT INTO: - either all inserted or none inserted  - attributes can be specified out of order  - missing values are replaced w NULL (if allowed) or default (if specified) | | | | | For DELETE FROM: - condition is optional (if unspecified, equivalent to always true); - if specified condition must evaluate to Boolean, and can only involve this row and this table | | | |
| 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 ,  Col Constraint: applies to single col, specified at col definition  Table Constraint: applies to 1 or more col, specified after col definition | | | | | | 1) NOT NULL. 2) UNIQUE. 3) PRIMARY KEY. 4) FOREIGN KEY. 5) General Constraint  Named Constraint: name specified by user  <attr> <type> CONSTRAINT <name> <constraint>  Unnamed Constraint: name assigned by DBMS | | |
| 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 except check of constraint cannot be deferred  3) CASCADE: propagates delete/update to referencing tuples  4) SET DEFAULT: updates foreign key of referencing tuples to some default value (value must still be primary key in referenced table)  5) SET NULL: updates foreign key of referencing tuples to NULL value (col must allow NULL) | | | | | | | e.g.  CREATE TABLE Table1 (  name TEXT DEFAULT 'string',  FOREIGN KEY (name) REFERENCES Table2 (name) ON DELETE SET DEFAULT ON UPDATE CASCADE  ); | |
| SET NULL has issues w prime attributes, SET DEFAULT has issues w default value not in reference relations, CASCADE may create a chain of propagation (can affect performance) | |
| General Constraint | | Not a structural integrity constraint. Allow us to specify that col values must satisfy an arbitrary Boolean expr  Can place constraint anywhere (under col or table\_constraint) | | | | | | CHECK (  (name = '' AND age >= 30) OR (name <> '' AND age > 0)  ) ); | | |
| Modifying DB | | Can add/drop cols/constraints, change data type/default value of col  ALTER TABLE <table> ADD CONSTRAINT <constraint> FOREIGN KEY (col) REFERENCES <table2> (col);  ALTER TABLE <table> DROP CONSTRAINT <constraint> <col> | | | ALTER TABLE <table> ALTER COLUMN <col> TYPE <type>;  AT <table> ALTER COLUMN <col> SET DEFAULT <value>;  AT <table> ALTER COLUMN <col> DROP DEFAULT;  AT <table> ADD COLUMN <col> <type> DEFAULT <value>;  AT <table> DROP COLUMN <col>; | | | | | |
| Deferrable Constraint | | Default: constraints check immediately at end of statement  True even for transaction containing multiple SQL statement (violation will cause statement/transaction to be rolled back)  Relaxed Constraint: check only at end of transaction  - can use for UNIQUE, PRIMARY KEY, FOREIGN KEY | | Circular Reference: Tbl1 foreign key reference Tbl2 primary key & Tbl2 foreign key reference Tbl1 primary key  ALTER TABLE ... ADD CONSTRAINT ... FK ... (col) NOT DEFERRABLE  AT ... ADD CONSTRAINT ... FK ... (col) DEFERRABLE INITIALLY DEFERRED  AT ... ADD CONSTRAINT ... FK ... (col) DEFERRABLE INITIALLY IMMEDIATE | | | | | | |
| NOT DEFERRABLE (default for foreign key)  BEGIN; -- start of transaction  INSERT INTO Tbl1 VALUES (101,1001)  INSERT INTO Tbl2 VALUES (1001,101)  COMMIT; -- end of transaction  #transaction aborted as 1st insert statement fail due to foreign key constraint violated | INITIALLY DEFERRED  BEGIN; -- start of transaction  INSERT INTO Tbl1 VALUES (101,1001)  INSERT INTO Tbl2 VALUES (1001,101)  COMMIT; -- end of transaction  #transaction successful (constraint check at commit) | | | | | | | INITIALLY IMMEDIATE  BEGIN; -- start of transaction  SET CONSTRAINTS <constraint> DEFERRED  INSERT INTO Tbl1 VALUES (101,1001)  INSERT INTO Tbl2 VALUES (1001,101)  COMMIT; -- end of transaction  #transaction successful (same as not deferrable but can manually override) |
|  | | No need to care about order of SQL statements within a transaction  Allows for cyclic foreign key constraints  performance boost when constraint checks are bottle neck | | | Troubleshooting more difficult  Data definition no longer unambiguous  May incur performance penalty when performing queries | | | | | |
| Sql syntax | | Specification actions in case of foreign key constraint violations (ON UPDATE/ON DELETE)  Relaxed checks of violations w deferrable constraints. Convention: Keywords in uppercase, try to align elements.  [] are optional. Col constraint: DEFAULT <value> | | | | | | | | |
| CREATE TABLE <table\_name> (  <attr> <type> [<column\_constraint>],  <attr> <type> [<column\_constraint>],  :  [<table\_constraint>], -- comment  [<table\_constraint>], /\* comment \*/  :  ); | | INSERT INTO <table\_name> [(<attr>, ...)]  VALUES (<values>, ...) [, (<values>, ...)]; | | | | | | |
| DELETE FROM <table\_name> [WHERE <condition>]; | | | | | | |
| UPDATE <table\_name>  SET <attr> = <value> [, <attr> = <value>]  [WHERE <condition>]; | | | | | | |
| DROP TABLE [IF EXISTS] <table\_name> [, <table\_name> [...]]] [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) | FOREIGN KEY (A1, A2, ...) REFERENCES R1 (B1, B2, ...) | Tuple exists in R1 as Primary key or tuple contains NULL value | | General | CHECK (c) | CHECK (c) | Condition c does not evaluate to False | | | | | | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Overview | Conceptual DB Design: Capture requirements using conceptual model (Entity Relationship Model / ER Model  Logical DB Design: Map conceptual model to logical schema (ER Model -> Relational Schema) | | | |
| Entity Sets & Attributes | An entity is a representation of real world objs that are distinguishable from other objs  An entity set is a collection of entities of the same type (rectangles, nouns)  - Should not be proper nouns (e.g., Albert Einstein) As a good practice, make the name plurals | | | |
| Attributes are specific info describing an entity (oval)  1. Key Attributes (underlined): uniquely identify entity [uid]  2. Composite Attributes (composed of other ovals): composed of other attrs [address]  3. Multivalued Attributes (double-lined): 1 or more attr for a given entity [phone]  4. Derived Attributes (dashed line): derived from other attrs [age]  Attr in ER diagram should have same name iff they are semantically equivalent | | | |
| R/s sets | A r/s is an association among 1 or more entities  A r/s set is a collection of relationships of the *same type* (diamond, verbs)  Represented by a diamond in ER diagrams May have their own attributes *(further describe the relationship)*  Degree/arity of r/s sets = num of entity sets involved in the r/s set  - n-ary = n entity (in principle, no limitation on num of entity sets involved)  Binary r/s set (n = 2; e.g. on right); Ternary (n = 3)  Can add label on line btw entity and r/s set to further describe (use for ambiguities)  Each user can make 0 or more bookings and each booking can be made by at most 1 user and at least 1 user (i.e. exactly 1 user) | | |  |
| R/s Constraints | 1. Cardinality Constraints: upper bound to num of times an entity can participate in a r/s (either 1 or ∞)  - Upper limit of 1 = key constraints  2. Participation Constraints: lower bound to num of times an entity can participate in a r/s (either 0 or 1)  - Lower limit of 0 = partial participation constraints. - Lower limit of 1 = total participation constraints  3. Dependency constraints: Entity set that does NOT have its own key (its keys are called partial keys)  - Must have identifying r/s set connecting to the owning entity set | | | |
| Cardinality | Constraint only for binary r/s:  1. Many-to-Many (e.g. a flight can be performed by diff aircrafts; an aircraft can perform diff flights)  2. Many-to-One (e.g. user can make many bookings, but each booking done by 1 user)  3. One-to-One (e.g. person has 1 set of credit card details, vice versa)  ER Diagram: default upper limit is ∞, specified by a line. Upper limit of 1 is specified by an arrorhead () | | | |
| Participation | An entity does not have to participate in a relation  Default lower limit is 0, specified by a line. Lower limit of 1 specified by double line (=) | | | |
| Dependency | Weak Entity Set: Entity set that does NOT have its own key (double line)  - Its own key is called partial key (but cannot uniquely identify entity; can only uniquely identify w help of primary key from owner entity)  - Its existence is dependent on existence of its owner entity - It is connected to owner entity via identifying r/s set (double line)  Many-to-one r/s from weak entity set to owner entity set  Weak entity set must have total participation constraint in identifying r/s  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 | Name of entity set | Attr of entity set | Key attr of entity set | Derived/Composite attr of entity set | | Schema | Name of table | Attr/Col of table | PK of table | should not appear |   ER diagram limitations: - don't capture data type (assume logical/common sense) - 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)  Composite: We assume table only hold atomic values. i.e. Composite attr in ER diagram only need set of single-valued attr in Schema  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 relationship, CREATE TABLE for r/s but use FK to REFERENCE key in Entity Table | | | |
| CREATE TABLE WeakEntity (  fid INT REFERENCES Owner,  fdate DATE,  PRIMARY KEY (fid, fdate)  );  CREATE TABLE Entity (  eid INT PRIMARY KEY); | | CREATE TABLE Relationship (  fid INT,  fdate DATE,  *eid INT REFERENCES Entity,*  *FOREIGN KEY (fid, fdate) REFERENCES WeakEntity,*  *PRIMARY KEY (fid, fdate, eid)*  ); | |
| Many-to-One  (many users, 1 booking)  – <- | CREATE TABLE Users (  uid INT PRIMARY KEY);  CREATE TABLE Bookings (  bid INT PRIMARY KEY,  bdate DATE); | | CREATE TABLE Makes (  uid INT REFERENCES Users,  bid INT REFERENCES Bookings,  *PRIMARY KEY (bid)* ); | |
| This approach remove Bookings and combine w Makes into 1 table  CREATE TABLE Users (  uid INT PRIMARY KEY); | | CREATE TABLE MakesBookings (  uid INT REFERENCES Users,  bid INT PRIMARY KEY,  bdate DATE ); | |
| One-to-One  –> <– | CREATE TABLE Users (  uid INT PRIMARY KEY,  ccnum INT UNIQUE REFERENCES CreditCards);  -- Assume CreditCards is not combined | | CREATE TABLE CreditCards (  ccnum INT PRIMARY KEY,  uid INT UNIQUE REFERENCES Users);  -- Assume Users is not combined | |
| CREATE TABLE Users (  uid INT PRIMARY KEY,  ccnum INT UNIQUE); | | OR CREATE TABLE Users (  uid INT UNIQUE,  ccnum INT PRIMARY KEY); | |
| Key + Total Participation  – <= | (exactly 1 booking)  CREATE TABLE Users (uid INT PRIMARY KEY); | | CREATE TABLE MakesBookings (  bid INT PRIMARY KEY,  uid INT NOT NULL REFERENCES Users); | |
| Weak Entity  – <<>> <= | CREATE TABLE Flights (fnum INT PRIMARY KEY); | CREATE TABLE FlightInstances (  fnum INT REFERENCES Flights ON DELETE CASCADE ON UPDATE CASCADE,  fdate DATE,  PRIMARY KEY (fnum, fdate)); | | |
| ISA Hierachies | Special type of r/s: 'is a'. (i.e. superclass/subclass)  Attr already shown in superclass should not be shown again in subclass  Diagram  Description automatically generated  Overlap Constraint: Can a superclass entity belong to multiple subclasses?  Covering Constraint: Must a superclass entity belong to at least 1 subclass?  TRUE covering constraint = total participation constraint  FALSE overlap constraint = key constraint  Conventionally, superclass is above ISA, and subclass below  To avoid ambiguity, use dashed line to connect superclass when overlap = TRUE and covering = FALSE  SQL: FK in subclass table reference PK in immediate superclass table (use ON DELETE CASCADE)  If subclass can belong to 2 superclass (use col1 <type> PK REFERENCES superclass1 REFERENCES superclass2) | | | |
| Aggregation | Want to capture r/s btw entity sets and r/s sets (for some proj worked on by students, GPU may be used)  Aggregation treats r/s as higher-level entities  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), hours NUMERIC,  PRIMARY KEY (gid, sid, name),  FOREIGN KEY (sid, name) REFERENCES Works (sid, name)); | | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SQL vs RA | | | |  |  |  | | --- | --- | --- | |  | Relational Algebra | SQL | | Paradigm | Imperative (query language using operators) | Declarative (query language built on top of RA) | | Semantic | Set | Multiset/Bag | | Query | RA expression | SELECT |   SELECT [DISTINCT] <columns/attributes> FROM <tables/relations> WHERE <conditions>  SELECT DISTINCT a1, ..., am FROM r1, ..., rn WHERE c translated to  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  Note cannot have same table name in FROM clause (unless u rename) | | | | | |
| SELECT | | | Complex condition: use OR and AND. Operators: {=, <>, <, <=, >=, >}  SELECT col1, 'S$' || (col2 \* 1.5) AS new\_col  String op: {||, LOWER(s), UPPER(s)}. Datetime: {+, NOW(), ...} | | Mathematical op: {+, \*, -, /, ^, |/ (√), ...}  #Concatenate 'S$' and values in each row after col2 \* 1.5  #rename this new col as new\_col | | | |
| DISTINCT keyword check for distinct rows using IS DISTINCT FROM (i.e. ), opp is IS NOT DISTINCT FROM () | | | | | |
| WHERE | | | x IS NULL (True if x is null) vs x = NULL (always False). Same for IS NOT NULL & <> NOT NULL  Pattern matching ([NOT] LIKE): \_ (any character); % (0 or more characters)  Note %\_ is equivalent to % | | | | | WHERE col IS NULL  WHERE col LIKE 'abc%yz' |
| Set Ops | | | UNION, INTERSECT, EXCEPT (i.e. set difference) where both tables are union-compatible  Note, set ops remove duplicate rows by default & col names follow left relation  use UNION ALL, INTERSECT ALL, EXCEPT ALL to keep duplicate rows (where each element is now treated as distinct element) | | | | | |
| Join Ops | A table name (after renaming) can appear at most once in the FROM clause  Can rename table to allow same table to appear more than once in the FROM clause but w diff name  Note: AS is optional in SQL statement. Can replace WHERE with AND as well. JOIN refers to INNER JOIN, but INNER is optional  - FROM Table1 t1 NATURAL JOIN Table2 t2 WHERE (auto join on common attributes)  - Based on universal schema assumption (attrs have same name if they refer to the same thing)  - FROM Table1 t1 LEFT JOIN Table2 t2 ON t1.col1 = t2.col2 #LEFT JOIN same as LEFT OUTER JOIN  There is also RIGHT [OUTER] JOIN, FULL [OUTER] JOIN, NATURAL LEFT JOIN (no need specify ON) | | | | | | | SELECT t1.col1, t2.col1  FROM Table1 AS t1 JOIN Table2 t2  ON t1.col2 = t2.col2  WHERE t1.col2 < t2.col2 |
| 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, ...)  Rule of thumb: IN subqueries can usually be replaced w inner joins, NOT IN w outer joins | | | | | 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  Subquery must return exactly 1 col. <op>: {=, <>, <, <=, >=, >}. <expr> is compared to each row from subquery using <op>. 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>;  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 1 (note 1 is just a placeholder, since we only care about the num of rows) | | | | SELECT DISTINCT col1 FROM Table1 t1  WHERE EXISTS (SELECT 1 FROM Table2 t2  WHERE t1.col2 = t2.col2); | |
| Scoping | | Correlated subqueries: inner query uses value from outer query (uncorrelated subqueries for [NOT] EXISTS are usually wrong or unnecessary)  Naming ambiguities: use table aliases: e.g. t1.col1, t2.col2  Scoping rules: A table alias declared in (sub-)query Q can only be used in Q or subqueries nested within Q  - If same table alias used in Q1 and outer query Q0, declaration in Q1 is used  General rule: from inner to outer queries | | | | | | |
| Remarks | | | WHERE <expr> IN <subquery> is equivalent to WHERE <expr> = ANY <subquery>  WHERE <col1> <op> ANY (SELECT <col2> FROM <rel> WHERE <cond>) is equivalent to  WHERE EXISTS (SELECT \* FROM <rel> WHERE <cond> AND <col1> <op> <col2>) | | | | | |
| Ordering | | | Soring of rows by a single attribute can be done by: ORDER BY <attr> ASC (note ASC is default, so is optional). or ORDER BY <attr> DESC  Can also sort by multiple attr: 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 | Aggregate fn compute a single value from a set of tuples: MIN(), MAX(), AVG(), COUNT(), SUM()  SELECT MIN(col) AS LOWEST, FROM Table;   |  |  |  |  | | --- | --- | --- | --- | | Query  R(A); | Interpretation | If R is empty relation | S is non-empty relation with attr A that only has NULL values | | MIN(A) | Min non-NULL value in A | NULL | NULL | | MAX(A) | Max non-NULL value in A | NULL | NULL | | AVG(A) | Avg of non-NULL values in A | NULL | NULL | | SUM(A) | Sum of non-NULL values in A | NULL | NULL | | COUNT(A) | Count of non-NULL values in A | 0 | 0 | | COUNT(\*) | Count of rows in A | 0 | n | | AVG(DISTINCT A) | Ave of distinct non-NULL values in A |  |  | | SUM(DISTINCT A) | Sum of distinct non-NULL values in A |  |  | | COUNT(DISTINCT A) | Count of distinct non-NULL values in A |  |  |   Function signatures: MIN/MAX: any comparable type same as input.  SUM: INT BIGINT OR REAL REAL. COUNT: any data BIGINT | | | | | | | | |
| Grouping | GROUP BY attr1, attr2, .... GROUP BY w/o aggregation is valid but not meaningful  Aggregation fn is applied over each group. E.g. SELECT MIN(price) FROM Table GROUP BY name;  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 the GROUP BY clause. 2. Ai appears as input of an aggregation fn in the SELECT clause  3. Primary key of R appears in the GROUP BY clause (basically separate by PK) | | | | | | | | |
| HAVING | GROUP BY attr1, attr2, ... HAVING <condition>. HAVING must be used w GROUP BY.  Conditions check for each group defined by GROUP BY clause. Conditions typically involve aggregate fn  E.g. SELECT COUNT(col1) FROM Table GROUP BY col2 HAVING COUNT(col1) > 1  Restriction: If col Ai of table R appears in HAVING clause, one of the following must hold:  1. Ai appear in the GROUP BY clause. 2. Ai appears as input of an aggregation fn in the HAVING clause  3. Primary key of R appears in the GROUP BY clause (basically separate by PK) | | | | | | | | |
| 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 and no condition matched | | | 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  COALESCE(price, 0): return price if not NULL else 0 | | | | | | Equivalent to CASE WHEN <value1> IS NOT NULL THEN <value1>    ELSE <valuen>  END | | |
| 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 T1 (col1) AS (  SELECT DISTINCT col1 FROM Table1)  SELECT col3, col4 FROM Table2 T2, Table3 T3  WHERE T2.name = T3.name  AND T2.col3 IN (SELECT \* FROM T1) | 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  CTE is to help improved readability  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)  VIEW can be used almost like a normal table | | | | | | 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 Quantifica-tion (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|  Find name of restaurant that sells all pizzas liked by 'Homer' | | SELECT DISTINCT rname FROM Sells S  WHERE (SELECT COUNT(DISTINCT pizza) FROM (  SELECT pizza FROM Sells S1 WHERE S1.rname = S.rname  UNION  SELECT pizza FROM Likes WHERE cname = 'Homer' AS T1  ) = (  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  Query is evaluated 'lazily', stops when fixed pt is reached  E.g. Find MRT station that can be reached from NS1 in at most 3 stops | | | | | | | WITH RECURSIVE  Function(to\_stn, stops) AS (  SELECT to\_stn, 0 FROM MRT  WHERE fr\_stn = 'NS1'  UNION ALL  SELECT M.to\_stn, F.stops + 1  FROM Function F, MRT M  WHERE F.to\_stn = M.fr\_stn AND F.stops < 2)  SELECT DISTINCT (to\_stn) FROM Function; | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| SQL with other language | | - Mix language with SQL (in C, do EXEC SQL <sql command>)  - Write a program that calls some API to run SQL queries (JDBC/ODBC) | | EXEC SQL BEGIN DECLARE SECTION: char stmt[500];  EXEC SQL END DECLARE SECTION;  EXEC SQL CONNECT TO @localhost USER john;  ... //some code to assign SQL statement to stmt  1) EXEC SQL EXECUTE IMMEDIATE :stmt; #execute statement stored in stmt  OR 2) EXEC SQL PREPARE mystmt FROM :stmt;  EXEC SQL EXECUTE mystmt USING 42, 'foobar';  EXEC SQL DEALLOCATE PREPARE mystmt;  EXEC SQL DISCONNECT; | | | |
| Dynamic SQL | | Static SQL: SQL statement if "fixed" beforehand  Dynamic SQL: SQL statement to be executed is not fixed in advance. Statement is stores in a string variable and complied and executed at runtime | |
| Call-Level Interface (CLI) | | connection C("dbname = testdb user = postgres password = pw hostaddr = 127.0.0.1 port = 5432")  sql = "CREATE TABLE COMPANY("ID INT PRIMARY KEY, NAME TEXT NOT NULL");";  work W(C); //create a transactional object W.exec(sql); W.commit(); C.disconnect(); | | | | | |
| 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 Scores; | | | | CREATE OR REPLACE FUNCTION topStudent() RETURNS Scores AS $$ //return only 1 tuple w same schema as Scores  SELECT \* FROM Scores ORDER BY Mark DESC LIMIT 1;  $$ LANGUAGE sql; | | |
| //return multiple tuples w same schema as Scores  CREATE OR REPLACE FUNCTION topStudent() RETURNS SETOF Scores AS $$  SELECT \* FROM Scores  WHERE Mark = (SELECT MAX(Mark) FROM Scores);  $$ LANGUAGE sql; | | |
| #return 1 tuple w diff schema  CREATE OR REPLACE FUNCTION topMarkCnt (OUT TopMark INT, OUT Cnt INT)  RETURNS RECORD AS $$  SELECT Mark, COUNT(\*) FROM Scores  WHERE Mark = (SELECT MAX(Mark) FROM Scores)  GROUP BY Mark ;  $$ 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 $$  SELECT Mark, COUNT(\*) FROM Scores  GROUP BY Mark ;  $$ 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 $$  UPDATE Accounts SET balance = balance - amount WHERE name = fromAcct;  UPDATE Accounts SET balance = balance + amount WHERE name = toAcct;  $$ LANGUAGE SQL | | | | |
| Control Structures | | IF ... THEN ... ELSE ... END IF  LOOP ... END LOOP  EXIT ... WHEN ...  WHILE ... LOOP ... END LOOP  FOR ... IN ... LOOP ... END LOOP  ...  SELECT swap(99,22); | CREATE OR REPLACE FUNCTION swap (INOUT val1 INT, INOUT val2 INT) RETURNS RECORD AS $$  DECLARE temp\_val INTEGER;  BEGIN IF val1 > val2 THEN temp\_val := val1; val1 := val2; val2 := temp\_val; END IF;  END; $$ LANGUAGE plpgsql; | | | | |
| CREATE OR REPLACE FUNCTION sum\_to\_x (IN x INT, OUT s INT) RETURNS INTEGER AS $$  DECLARE temp\_val INTEGER;  BEGIN s := 0; temp\_val := 1;  LOOP EXIT WHEN temp\_val > x; s := s + temp\_val; temp\_val := temp\_val + 1; END LOOP;  END; $$ LANGUAGE plpgsql; | | | | |
| Cursor | | CREATE OR REPLACE FUNCTION score\_gap()  RETURNS TABLE ( name TEXT, mark INT, gap INT ) AS $$  DECLARE  curs CURSOR FOR (SELECT \* FROM Scores ORDER BY Mark DESC);  r RECORD; prv\_mark INT;  BEGIN prv\_mark := -1;  OPEN curs;  LOOP  FETCH curs INTO r;  EXIT WHEN NOT FOUND;  name := r.Name; mark := r.Mark;  IF prv\_mark >= 0 THEN gap := prv\_mark - mark;  ELSE gap := NULL;  END IF;  RETURN NEXT;  prv\_mark := r.Mark;  END LOOP;  CLOSE curs;  END; $$ 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 FROM cur INTO r  FETCH FIRST FROM cur INTO r  FETCH LAST FROM cur INTO r  FETCH ABSOLUTE n FROM cur INTO r (fetch nth tuple) |
| SQL Injection | | Suppose we have a user-provided parameter input\_name and define stmt := SELECT \* FROM T WHERE Name = 'input\_name';  EXEC SQL EXECUTE IMMEDIATE :stmt;  But if user sets input\_name to be: a'; DROP TABLE T; SELECT 'a, then whole table is deleted | | | | | |
|  | | To prevent this, either use function/procedures or prepared statements  CREATE OR REPLACE FUNCTION queryT (IN in\_name TEXT)  RETURNS SETOF T AS $$  SELECT \* FROM T WHERE Name = in\_name;  $$ LANGUAGE sql; | | | | | EXEC SQL PREPARE mystmt FROM :stmt;  EXEC SQL EXECUTE mystmt USING :in\_name; |
| For both, SQL statement is compiled first when prepared/in function  So anything in in\_name 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(); | | | | | #NEW: new row inserted into Scores, 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, No OLD for INSERT, No NEW for DELETE, Not for INSTEAD OF | |
| Deferred Trigger | | Finish whole transaction before checking trigger  CREATE ... FUNCTION ... IF .... RAISE EXCEPTION ...  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 |
|  | | BEGIN TRANSACTION;  ...  COMMIT; #trigger only activated at COMMIT | | For INITIALLY IMMEDIATE:  BEGIN TRANSACTION;  SET CONSTRAINTS bal\_check\_trigger DEFERRED; COMMIT; | | | |
| Multiple Triggers | | Can have multiple triggers defined for same event on same table  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  FDs may be diff across tables. Must specify context  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 | Let S = {A1, ..., An} be set of attributes. Closure of S is set of attributes that can be decided by A1, ..., An (directly or indirectly). Notation: {A1, ..., An}+  To prove X Y (does not) holds, just show {X}+ (does not) contains Y | | | | | E.g. A B, C D, BC E  {A}+ = {A, B}, {A, C}+ = {A, B, C, D, E}, {B}+ = {B} | | |
| Superkeys, Keys, Prime attributes | Superkeys: set of attributes in table that decides all other attributes  Keys: a superkey that is minimal  Prime attribute: attr that appears in a key (else: non-prime) | | | To find keys: - consider every subset of attributes in; derive closure; identify superkeys based on closures; identify keys  - check small attribute sets first  - check which attr don't appear in RHS of FD (i.e. attr must be in key) | | | | |
| Normal Forms | Conditions that a "good" table should satisfy. In increasing order of strictness:  1st NF, 2nd NF, 3rd NF (3NF), Boyce-Codd NF (BCNF), 4th NF, 5th NF, 6th NF.  3NF, BCNF get rid of most redundancies, always possible to satisfy | | | | | | 1st and 2nd easy to satisfy, have high redundancy  4th, 5th, 6th very little redundancy, not always possible to satisfy | |
| 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  For all possible subset of attrs, compute closure. From each closure, remove the "trivial attr" (aka LHS) from RHS. Derive non-trivial and decomposed FDs | | | | | | | |
| BCNF | A table R is in BCNF, if every non-trivial and decomposed FD has a superkey as its LHS  If LHS not superkey = LHS can appear multiple times in table = RHS can appear multiple times = Redundancy | | | | | | | |
| BCNF Check | Compute closure for all attr subset. Check if there is a closure {A1...Ak}+ s.t. it is a "more but not all" closure (i.e. contains some attr not in {A1...Ak} but does not contain all attr in table). 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 | | | |
| Each decomposition step removes at least 1 BCNF violation | | | | | | |
| 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 | | Algo: - 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 can be derived from M, and vice versa (check using closure)  2. every FD in M is a non-trivial and decomposed FD | | | 3. if any FD is removed from M, then some FD in S cannot 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 |