Database: Data – Info – Knowledge. **Data model**: a collection of concepts for describing data. **Schema**: a description of a particular collection of data using a given a data model. **Database**: the data as stored and managed by the DBMS. **DBMS**: software package that adds functionality (MYSQL). Database System: hardware/software to support DBMS.

Relational Models: relations; attributes; tuples; domain: 1) data type 2) value range; cardinality: # of tuples; degree: number of attributes. **Relational Schema**: table consists of a set of attr. Table consist of a set of tuples, DB consists of a set of tables. **Relational data instance**: ordering does not matter. NULL values.

Integrity constrains: constrains defined on DB schema to regulate what is consider valid instance in the DB, e.g., domain constrains.

Key Constrains: super key (1) set of attributes (2) uniquely identify each tuple; candidate key: (1),(2),(3) minimum; primary key: one of the candidate key. **Entity Integrity**: (1) each table must have a primary key. (2) no tuple can have NULL value on key. (3) no to tuples have same value on primary key. **Foreign key**: attribute(s) in one table that refers to PK in another table. **Enforcing Integrity Constrain**: cascading/non-cascading.

SQL: DDL (Create, alter, Drop) table; DML (Insert, Delete, Update). Other Data Models: XML (semi-structured), RDF (Graph model)

DB Application: **transaction**: is a sequence of database operations that are packed into a unit to which the DBMS offers certain quality guarantees (ACID). **Properties of a transaction**: ACID, atomicity, consistency, isolation, durability. A: all or nothing (unit), C: check before and after, I: concurrency control, each user does not feel the impact of others, D: log/recovery system. **Programming with transactions**: start/commit (durability) /rollback(atomicity) (all of these need to happen linearly in one function).

ER model: **entity:** set of similar objects that can be described by common attributes. Attributes: may be composed, may be represented as multi-valued (phone number). Kyes: real-key (SSN) artificial-key (student-id) (a key is real or artificial relative to the kind of system). **Relationship**: rules of correspondence between set of entities. **Constrains**: regulate involvement of instances in an entity relationship. (1) key constrains: at most one ➔ (2) participation constrain at least one (full participation, bold line w/o arrows) (1) and (2) can be combined to represent exactly one. **Class hierarchy**: super class, sub class (Person ISA Director, Actor, etc). **Weak entity:** its key is a combination of its parent key and its own key, must have exact participation (bold line) in the relationship with its parent.

Functional Dependencies: R<U,F> A->B, A in U B in U, if ∀r1,r2 in R, r1(A) = r2(A) => r1(B) = r2(B)

R<U,F> X->Y, X subset U Y subset U, if ∀r1,r2 in R, r1(X) = r2(X) => r1(Y) = r2(Y) . **Trivial FD** Y subset X, **No Trivial FD** Y not subset X, **complete nontrivial FD** X intersect Y = empty, **Full FD**: does not exists X' contained in X: X'->Y. **X is superkey** if X->U. **X is candidate key** if X->U is a Full FD, **X is a full key** if: X->U=>X=U

Properties of FD: **Reflexivity** if Y subset X then X->Y, **Transitivity** if X->Y and Y->Z then X->Z, **Augmentation** if X->Y then XZ->YZ where XZ=X Union Z. **Union rule**: if X->Y,X->Z then X->YZ, **Decomposition Rule**: if X->YZ then X->Y,X->Z, **pseudo-transitivity**: if X->Y and WY->Z, then XW->Z, **set accumulation**: if X->YZ, Z->W then X->YZW.

Closure: Let F be a set of functional dependencies, the closure of F, F^+, is the set of all FDs that are implied by F.
Equivalent and Cover: Given a schema R and two functional dependencies F and G, if $F^+=G^+$ over R, F≡G. F is a cover of G. (1) F cover G ⇔ (2) every g in G is implied by F ⇔ (3) G in F+. F+=G+ ⇔ F ≡ G ⇔ F cover G and G cover F.
Minimum cover: Given functional dependencies G, F is a minimum cover of G if: (1)$F^+=G^+$ (2)the right hand side of each FD in F is a single attribute (3) F is minimal, that is, if we remove any attribute from an FD in F or remove any FD from F, then F^+ will no longer equal G^+
Minimum Coverage Algorithm: F ≡Fc (1) X-> A (2) all X->A, As minimum (3) all X->A is required. (1) for every rule X-> Y where Y=A1,A2,…,An, then X->A1, X->A2,…,X->An (2) for every rule f=X->A, where X=B1,…,Bn, consider f'=B1,…,Bn-1->A. F'=(F-{f}) Union {f'}. If there exist a rule in F such that B1…Bn-1-> Bn then replace X->A with B1…Bn-1->A (3) for each rule X->A make F' = F-{X->A}, F'≡F, if A in(X)+ then remove X->A.
Normal Forms: 1st NF: every attribute cannot be further divided**. 2nd NF**: all nonkey attribute should totally depend on PK. **3rd NF**: for any non-trivial FD, X->A, one of the following must be true (1) X is a super key (2) A is a primary attribute (part of a candidate key) **BCNF**: (1) An attribute cannot be determined by an attribute that is not a key (2) all left hand side of all FD must contain key. (Alternatively, for each Ā->B, Ā is a key). Note: If a set of attr. Is a key then $(attr)^+$ must be equal to all attributes in the relation.

BCNF Decomposition Algorithm:

```
Let R be the initial table with FDs F
S={R}
Until all relation schemes in S are in BCNF
  for each R in S
    for each FD X → Y that violates BCNF for R
      S = (S – {R}) ∪ (R-Y) ∪ (X,Y)
Enduntil
```

This is a simplified version. In words:
When we find a table R with BCNF violation X→Y we:
1] Remove R from S
2] Add a table that has the same attributes as R except for Y
3] Add a second table that contains the attributes in X and Y

Relational Decomposition: take R<U,F> and break it into R1<U1,F1> … Rn<Un,Fn> so that: U1 union U2 union…union Un = U. A well-behaved decomposition meets: (1) lossless (don't loss any info) (2) preserve FD (F1 union F2 union … union Fn equivalent F.

Algorithm for well-behaved decomposition: input R<U,F> => R1,…,Rn (1) Fc (2) for every X->Y create a table (3) if no table contains any candidate key then create a table that contains a PK.

Formal Query Languages: Relational Algebra, Relational Calculus.

RA: (1) **Relation** R (2) **Selection** σ_cond(R), **input** R, parameter cond (col op value/col op col), **output** schema same as R, instance tuple in R that satisfy condition. (3) **Projection** π_col1,col2,...,coln(R), **input** R, parameter col1,col2,...coln, **output** schema (col1,col2,...,coln), instance tuples without repetition. (4) **Set operation** R U S, R ∩ S, R-S, **input** R,S (with exactly the same schema) **output** schema same as R or S. (5) **Cross Product** RxS **input** R,S, col(R) ∩ col(S) = ∅, **output** col(R) U col(S). (6) **θ Join** R bowtie θ S (from implementation point of view it is only one operator, from a logical point of view it can be defined from other operations) **input** R,S, parameter θ -> col1 op col2, col1 in R, col2 in S, **output** schema col(R) U col(S), instance R bowtie σ S = σ_θ (RxS) (7) Natural Join R bowtie S = for each pair of attribute in R and S R.a = S.a where attributes names are the same.

RC: Declarative formal query language; basic ingredients: variable, constants, comparison operators, logical operators, quantifiers, formula. **TRC**: All variables represent tuples, constants: 1,2,"CS", Comparison <> = ... logical operators AND, OR, NOT, quantifiers, forall, exists. Example {t| t ∈ Student} (get all students), {t| ∃s ∈ Student(s.name=t.name)} (equivalent to projection in RA) (t is called de free variable not quantified)

SQL: Select (1) attr(s) (2) * (3) expression +,-,*,/,... (4) aggregate function count() max() min() avg() sum() (5) distinct(attr1,attr2,...) note: avg(age) ≠ avg(distinct(age)); **From** T1,T,2,... e,g, Enrollment as E; **WHERE** a Boolean expression; **Group By** attr(s) **Having** aggregate function (column_name) operator value. Whenever you have the group by clause, involve the attr(s) in Select.

SQL Union, Intersect, Except: (S F W) Union <All> (S F W). the default behavior in not to keep duplicates. Use keyword All to enforce keep duplicates.

SQL Nested Queries: Where Clause (1) attr IN (S F W) must return a valid set and only one column and data type comparable (2) Exists (S F W) boolean expression, return true if (S F W) is non-empty, false otherwise (3) attr = > <,..., <ALL|ANY> (S F W)

EXAMPLES:

RA: **Find the students who have never taken any course that's not required by his department.**

Students - (Students ⋈ π_{sid}(Take - $\pi_{sid,cid,term,year,grade}$(Students ⋈ Take ⋈ RequiredCourse)))

π_{sid}(Student) – π_{sid}($\pi_{sid,cid}$(Take) – $\pi_{sid,cid}$(Students ⋈ RequiredCourses))

RC: Find the names of all students who are not in Informatics or CS.

{t| ∃s ∈ Student(s.name=t.name ∧ ¬(s.dept='CS' ∨ s.dept='Info')}

Find the names of students who took B561 and got A

{t| ∃s ∈ Student(s.name=t.name ∧ ∃e ∈ Enrollment(e.sid=s.sid ∧ e.grade='A' ∧ e.cid='B561'))}

SQL: **Find the youngest students**

SELECT ID FROM Student WHERE age = (SELECT MIN(age) FROM Student);

Find the department(s) where there are more male students than female students:

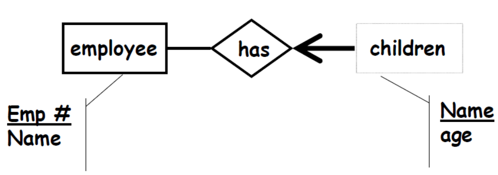
SELECT distinct F.dept FROM (SELECT dept,count(*) as CantF From Students WHERE gender = "F" Group By dept) As F,
(SELECT dept,count(*) as CantM From Students WHERE gender ="M" Group By dept) As
WHERE M.dept = F.dept AND F.cantF<M.cantM

Find the students who took all courses taken by Andrew S. (Find the students who did not took any course taken by A.S.)

SELECT Name FROM Students As S WHERE S.name !='AS' AND NOT EXISTS
((SELECT cid FROM Students as A, Enrollment as E WHERE A.sid = E.sid and A.name = 'AS')
EXCEPT (SELECT cid FROM Enrollment as E WHERE S.sid = E.sid))

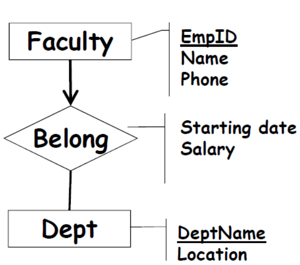
Find the students whose applied universities overlap with the universities applied by 0001

SELECT DISTINCT ID, name FROM Student, Application
WHERE ID=studentID and Univ in (
SELECT Univ
FROM Student, Application
WHERE ID = 0001 and ID=studentID);



Employee(<u>Emp</u>,Name)

hasChildren(<u>Emp REF. Employee, childName</u>,childAge)



Faculty_Belong(<u>EmpID</u>,Name,Phone,DeptName REF Dept, Start_date,Salary)

Dept(<u>DeptName</u>,Location)