INTRODUCTION TO DATABASE
- RELATIONAL ALGEBRA

College (cName, state, enrollment)
Student(sID,sName, GPA,sizeHS)
Apply(sID,cName, major, decision)

Select-condition operator:
Picks certain rows based on a condition
ex: Student with GPA>3.7 \select_{GPA>3.7} (Student)
← SQL: select * from … where ...

Project operator:
Takes certain columns based on attribute names
ex: TT_{sID,dec} Apply \project_{sID,dec} (Apply)

Composition
TT \project_{sID,dec} (\select_{GPA>3.7} (Student))

!!! In relational algrebra, we eliminate duplicate !!!
!!! Relation algrebra is <> from SQL on that fact !!!

Cross-product: Student x Apply
If Student has 8 tuples, and Apply has 4 ⇒ total number of tuples in Student x Apply = 8x4=32
!!! For attributes which are the same, they are renamed and prefixed by the original relation !!!
⇒ useful when combined with filtering (select operator)
ex: Student x Student ← self join !
= Student ← Use rename operator to differential attribute

Natural join (⋈ bowtie operator ⋈)
Cross-product where same attributes are the same
← ex: … where Student.sID = Apply.sID …is implied !
!!! Eliminate the duplicate attributes, there value is always going to be equal !!!
← number of column ? smaller than in cross-product
!!! Doesn't add expressive power, can be rewritten usnig select, project, cross-product !!!

Theta join (⋈_{theta} operator) ← theta is a condition
!!! Doesn't add expressive power to the language !!!
!!! In SQL, this is 'join' operation !!!

Union operator (∪)

Vertical union (of same attribute) <> join ← Attributes need to have the same name
← See ‘ϱ’ or ‘as’

ex: return Student names and College names in one column

!!! Attributes of union needs to have the same name !!!

ex: ϱ_{C(name)} [TT_{cName} (College)] ∪ ϱ_{c(name)} [TT_{sName} (Student)]

Difference operator (-)

← Attributes need to have the same name

ex: TT_{sName} ((TT_{sID} Student - TT_{sID} Apply) ⋈ Student)

Intersection operator (∩)

← Attribute needs to have the same name
← See ‘ϱ’ or ‘as’

ex; names that are both a college name and a student name

TT_{cName} (College) **∩** TT_{sName} (Student)

!!! Doesn’t add any expressive power: E1 ∩ E2 == E1 - (E1 - E2) == E1 ⋈ E2!!!

Rename operator (ϱ)

ϱ_{R(A1,...An)} (E) ← rename table and attributes of E
ϱ_{R} (E) ← rename table
ϱ_{A1,...An} (E) ← rename attributes of E
ex: \select_{s1=s2} [ϱ_{c1(n1,s1,e1)} [College] x ϱ_{c2(n2,s2,e2)} [College]]
← disambiguation in self-join

INTRODUCTION TO DATABASE
- SQL 2 (92)

Data Definition Language (DDL)
create tables, views,
drop table

Data Manipulation Language (DML)
Use to query and update the database
select
insert
delete
update

INTRODUCTION TO DATABASE
- SQL
- SELECT STATEMENT

Student(sID, sName, GPA, sizeHS) ← underlined are the keys

College(<u>cName</u>, state, entrollment)
Apply(<u>sID, cName, major</u>, decision)

basic select statement
select distinct A1, …, An
from R1,... , Rm ← table variables ← make cross product
where condition
=
\project_{A1,...An} (\select_{condition}(R1 \cross … \cross Rm)

ex:
select sName, major
from Student,Apply
where Student.sID = Apply.sID ← required to simulate natural join (⋈)
← number of column still same, but sID are ==

!!! In SQL we can have duplicate entries in the result <> relational algebra !!!
!!! if you want to prevent this behavior, use the ‘distinct’ keyword !!!

select **distinct** sName, major …. ← distinct operator

select with compounded conditions
select College.cName
from College, Apply
where College.cName = Apply.cname and enrollment > 2000 and major = ‘CS’;

!!! In the projection, we had to pick College.cName instead of just cName, because could also have been Apply.cname !!!

select with sort results ← SQL doesn’t order resulting tuple
select …. order by GPA desc, enrollment;

!!! ascending is default, so when nothing ⇒ ascending !!!
!!! sort first based on descending GPA, then on ascending enrollment !!!

string matching (like operator)
select * where major like ‘%bio%’;

renaming of column label (‘as’ operator)
select last_name as name from Student;

renaming of relations/table vars
select * from Student S, College C, Apply A where A.sID = S.sID

with on the fly arithmetic
select sId,SName, GPA, sizeHS, GPA*(sizeHS/1000.0) as scaledGPA from Students;

INTRODUCTION TO DATABASE
- SQL
- TABLE VARS AND SET OPERATORS

set operators: Union, Intersect, Except

select distinct A1, …, An
from R1,... , Rm ← table variables, join of tables (cross product)
where condition

aliasing table vars
ex: Every pair of student who have same GPA
select S1.sName, S2.sName
from Student S1, Student S2 ← no 'as' !!!!
where S1.GPA = S2.GPA and S1.sID <S2.sID;
← remove match to a-a, a-b, and b-a matching
← 'as' notation is used for attribues only

union without duplicates (union)
select cName as name from College
union
select sName as name from Student;

!!! Union in SQL removes duplicate (and sorts) !!!
!!! The sorting is because it is looking for duplicate !!!

ordered union with duplicates (union all)
select cName as name from Colleg
union all
select sName as name from Student
order by name;
!!! Union all do not sort, eliminate duplicate !!!
!!! Sorting as to be set explicitly !!!

intersection (intersect)
select sID from Apply where major = 'CS'
intersect
select sID from Apply where major = 'EE';

!!! intersect operation do not add expressive power !!!

When intersect is not supported by database, use
select distinct A1.sID ← Don't foget the distinct !!!
from Apply A1, Apply A2
where A1.sID = A2.sID and A1.major = 'CS' and A2.major = 'EE';

except
select sID, sName from Apply where major = 'CS'
except ← except: but did not
select sID, sName from Apply where major = 'EE';

!!! except operation do not add expressive power !!!

When except is not supported by database, instead use
select distinct A1.sID, A1.sName
from Apply A1, Apply A2 ← Join with self !
where A1.sID = A2.sID and A1.major = 'CS' and A2.major <> 'EE';
← !!! match CS, CS !!!

better
select sID, sname
from Student
where sID in (select sID from Apply where major = 'CS')
and sID not in (select sID from Apply where major = 'EE');

INTRODUCTION TO DATABASE
- SQL
- SUBQUERIES IN THE WHERE

select distinct A1, …, An
from R1,... , Rm
where condition ← can involve "subqueries": nested select statements

where … in and 'not in'
select sID, sName
from Student
where sID in (select sID from Apply where major = 'CS');
← sID match attribute in subquery results

equivalent to

select distinct student.sID, sName
from Student, Apply
where Student.sID = Apply.sID and major = 'CS';

!!! 2 students with same name are not duplicate because sID !!!

!!! Duplicate management is important: ex when calculating the average GPA !!!
← the where … in is the only way instead of using distinct
!!! Some students may have same GPA, but you don't want double counting !!!

select sID, sname
from Student
where sID in (select sID from Apply where major = 'CS')
and sID **not in** (select sID from Apply where major = 'EE');

or
…. and **not sID in** (select sID from Apply where major = 'EE');

where exists (return non empty? if exist)
ex: return college in the same state as another
select cName, state
from College C1
where exists (select * from College C2 where C2.state = C1.state
and C1.cName <> C2.cName)
← 'exists' always use a relation (C1) from outside subquery
← 'exists' always use 'select * from', because attributes not relevant

EXISTS simply tests whether for each tuple in outer query the inner query returns any row. If it does, then the outer query proceeds. If not, the outer query does not execute, and the entire SQL statement returns nothing.

where not exists
ex: return college with the highest enrollment
select cName
from College C1
where **not exists** (select * from College C2 where C2.enrollment > C1.enrollment)
⇒ maximum v1

where … all
You need to satisfy the condition with all elements of the set
ex: Find student with highest GPA
select sName, GPA
from Student
where GPA >= all (select GPA from Student);
← great to all element of subquery

!!! Beware of student with same GPA !!!

⇒ **maximum v2**

!!! where … > all (subquery) == where not … <= any (subquery) !!!

where … any

You need to satisfy the condition with at least one element of the set

```
select cName
from College C1
where not enrollment <= any (                    ← NOT
        select enrollment from College C2 where C2.cName <> C1.cName
        );
```

!!! where … > all (subquery) == where not …. <= any (subquery) !!!

When ‘where … any’ subquery is not supported, instead use ‘exists’:

ex: Returns strudents NOT from the smallest high school

```
select sID, sname, sizeHS
from Student
where sizeHS > any(select sizeHS from Student);
==
select sID, sName, sizeHS
from Student S1
where exists (select * from Student S2 where S2.sizeHS < S1.sizeHS);
```

← ‘exists’ always use a relation (S1) from outside subquery

← ‘exists’ always use ‘select * from’, because attributes not relevant

!!! avoid using ANY and ALL because can be tricky, rather use EXISTS !!!

```
select * from R1, …,Rn
where    (select count(*) … )    =    (select count(*) ….);
```

INTRODUCTION TO DATABASE
- SQL
- SUBQUERIES IN FROM CLAUSE

Use a query to generate a table, as if it was a table in the database

more arithmetic

```
select sID, sName, GPA, GPA*(sizeHS/1000.0) as scaledGPA
from Student
where abs(GPA*(sizeHS/1000.0)-GPA) >1.0;
```

only one time equation
Instead of doing it in the from and the where clause

```
select *
from (select sID, sName, GPA, GPA*(sizeHS/1000.0) as scaledGPA from Student) G
where abs(G.scaledGPA - GPA) > 1.0
```

INTRODUCTION TO DATABASE
- SQL
- SUBQUERIES IN SELECT CLAUSE

example:
Find the highest GPA of Students in each college

```
select distinct College.cName, state, GPA
from College, Apply, Student
where College.cName = Apply.cName
        and Apply.sId = Student.sID
        and GPA >= all (        select GPA from Student, Apply
                                where Student.sID = Apply.sID
                                        and Apply.cName = College.cName_;
```

…. or ....

Find the highest GPA of Students in each college

```
select cName, state, ( select distinct GPA
                       from Apply, Student
                       where College.cName = Apply.cName
                               and Apply.sID = Student.sID
                               and GPA >= all (        select GPA from Student, Apply
                                                       where Student.sID = Apply.sID
                                                               and Apply.cName = College.cName))
                       as GPA
from College
```

!!! select in select must return exactly one value (not a tuple or a column of rows!) !!!
← which is the case here (1 GPA for each college) !!!
← append a column

INTRODUCTION TO DATABASE
- SQL
- SQL AGGREGATION
- MIN, MAX, SUM, AVG, COUNT

```
select distinct A1, …, An
from R1,... , Rm
where condition
group by columns                  ←NEW only used in aggregation
having condition                  ←NEW, apply to group created by ‘group by’ clause
                                  ←Condition apply to the groups
```

avg, min

select * from Student; ←several columns

select avg(GPA) from Student; ←one value only ← aggregate

select min(GPA) from Student, Apply where Student.sID = Apply.sID and major = ‘CS’;
←!!! students may have applied to several CS college

count

select count (*) from Apply where cName = ‘Cornell’;
← overcounting student who applied several times at Cornell

select count (distinct sID) from Apply where cName = ‘Cornell’;
← no overcounting !!!

```
select CS.avgGPA - NonCS.avgGPA
from    ( select avg(GPA) as avgGPA
        from Student
        where sID in ( select sID
                        from Apply
                        where major = ‘CS’
                        )
        ) as CS,
        (select avg(GPA) as avgGPA
        from Student
        where sID not in (      select sID
                                from Apply
                                where major = ‘CS’
                                )
        ) as nonCS;
```

==

```
select distinct ( select avg(GPA) as avgGPA                ← distinct required for 'from Student'
        from Student
        where sID in ( select sID
                       from Apply
                       where major = 'CS'
                       )
        )
        -
        (select avg(GPA) as avgGPA
        from Student
        where sID not in (     select sID
                               from Apply
                               where major = 'CS'
                               )
        ) as d
from Students;          ← from always needed, calculate once for each student entry!!!
                               ← we get same result regardless of student tuple
                               ← Result is unique cell with 'distinct'
```

group by

← only used with aggregates

??? represent the scope of *

```
select cName, count(*)
from Apply
group by cName;
```

⇒ count the number of applicants for each college

⇒ order by?

```
select state, sum(enrollment)
from College
group by state;
```

⇒ Calculate total enrollment per state

```
select cName, major, min(GPA), max(GPA)
from Student, Apply
where Student.sID = Apply.sID
group by cName, major;
```

⇒ college name, major, min, max GPA

```
select Student.sID, sName, count(distinct cName)
from Student, Apply
where Student.sID = Apply.sID
group by Student.sID;
```

⇒ sID, sName, number of colleges he or she applied to

```
select Student.sID, count(distinct cName)
from Student, Apply
where Student.sID = Apply.sID
group by Student.sID;
union
select sID, 0                    ← arithmetic zero!!!
from Student
where sID not in (select sID from Apply);
```

having <group-condition>

← only used with aggregates

```
select cName
from Apply
group by cName
having count(*)<5;          ← condition apply to the entire group
                            ← 'where' clause, condition applies to one tuple at a time
```

⇒ find college with fewer than 5 applicants

equivalent to

```
select distinct cName
from Apply A1
where 5 > (     select count(*)
                from Apply A2
                where A2.cName = A1.cName
                ) ;
```

```
select major
from Student, Apply
where Student.sID = Apply.sID
group by major
having max(GPA) < (select avg(GPA) from Student );
```

INTRODUCTION TO DATABASE
- SQL
- NULL VALUES

NULL value means, it is undefined or unknown

In GUI, NULL is often representaed as BLANK
ex: no decision on the admission has been done

select sID, sName, GPA
from Student
where GPA > 3.5 or GPA <=3.5;

!!! NULL are not matched in condition !!!

…. where GPA >3.5 or GPA <= .35 or GPA is NULL; ← not GPA = NULL !!!

!!! matches everything !!!
!!! conditions are true, false, or unknown !!!

not null
select count(*)
from Student
where GPA is not null; ← not written as GPA = not null !!!
← beware of syntaxe errors !

what is null ?
select distinct GPA from Student; ← returns NULL entry
select count(distinct GPA) from Student; ← count doesn't includes NULL entry

---

INTRODUCTION TO DATABASE
- SQL
- DATA MODIFICATION STATEMENTS

Insert

insert into <table> values (A1,...An)
or
insert into <table> select-statement ← produce a set of entries/table

example:
insert into College
values ('Carnegie Mellon','PA',11500);

example:
Student who haven't applied anywhere are going to apply to Carnergie Mellon

```
insert into Apply
select sID, 'Carnegie Mellon', 'CS', null        ← null: Y or N for admission decision
from Student
where sID not in (select sID from Apply)         ← sID of students who haven't applied anywhere!
```

delete

```
delete from table
where condition
```

example:

```
delete from Student
where sID in ( select sID
               from Apply
               group by sID
               having count(distinct major)>2
               );
```

!!! If you delete from a table, you may have to delete from other tables as well i.e. Apply !!!
← possible inconsistency

!!! Before you delete, select the rows that you are going to delete and check ok !!!

Update

```
update table
set attribute = expression, A2 = Expr2          ← expression can be subqueries
where condition                                 ← where to update
```

advice: First use a 'select * where … ' statement and then change the 'select *' with 'update table set attr =... ', i.e. keep same where statement

examples:

```
update Apply
set decision = 'Y',major = 'economics'
where cName = 'Carnergie Melon'
      and sID in (select sID from Student where GPA <3.6);

update Apply
set major = 'CSE'
where major = 'EE'
      and sID in (select sID
                  from Student
```

```
                    where GPA >= all ( select GPA
                                       from Student
                                       where sID in (select sID
                                                     from Apply
                                                     where major = 'EE'
                                                     )
                                     )
               ) ;
```

```
update Student
set GPA = (select max(GPA) from student), sizeHS = (select min(sizeHS) from Student);
```

INTRODUCTION TO DATABASE
- RELATIONAL DESIGN THEORY

Create best schema
How to choose?
Design tools?

Bad design
Apply(SSN, studentName, collegeName, highSchool, highSchoolCity, hobby)
← 1 table to capture everything

Issues:
* capture redundancy of information
    - applications to <> colleges capture relationships between SSN and studentName
* update anomaly
    - partial update (due to redundancy)
    - update not at every place ← possible inconsistent database
* deletion anomaly
    - delete hobby Surfing delete students that have only this hobby

Better design
Student(SSN, sName)
Apply(SSN, cName)
HighSchool(SSN, highSchool)
Located(HighSchool, HighSchoolCity)
Hobbies(SSN, hobby)
← smaller tables and many more tables

!!! issues with 2 highSchool with same name in different city!!!
HighSchool(SSN, highSchool, highSchoolCity) and no Located table
← relationships are key with key

!!! All hobbies are disclosed to every college !!!
To change that
Apply(SSN, cName, hobby), and no Hobbies table

In general, better to create object tables and relation/verb tables.

INTRODUCTION TO DATABASE
- RELATIONAL DESIGN THEORY
- DESIGN BY DECOMPOSITION

Relational design by decomposition
1/ mega relations
2/ system decomposes based on properties ⟵ a.k.a constraints, world model
    - Functional dependencies ⟶ Boyce-Codd Normal Form
    - Multivalued dependencies ⟶ Fourth Normal From
3/ final set of relations satisfies normal form
    - no anomalies
    - no lost information

Functional dependency ⟵ Notation: SSN ⇒ sName
⟵ the notiation '⇒' means 'fully determine(s)'
    - Same SSN always has same sName
    - But sName doesn't necessarily leads to one SSN
    - Should store each SSN's sName only once

example:
    SSN ⇒ sName ⟵ 1 student has one name
    SSN ⇒ address ⟵ 1 student live at one address (doesn't move!)
    ⟵ function of our world model

    SSN, highSchool ⇒ sName ⟵ SSN and highSchool FULLY determine sName
    ⟵ SSN and highSchool are independent

    HSname, HScity ⇒ HScode ⟵ key determines other key !
    HScode ⇒ HSname, HScity

    HSname, HScity ⇒ HScode ⟵ HSname and HScity FULLY determine HScode
    ⟵ HSname and HScity are independent
    ⟵ the notiation '⇒' means 'fully determine(s)'

(In) Boyce-Codd Normal Form (if follows) ⟵ aka BCNF

- Remove functional dependencies
- If A ⇒ B then A is key ← key means no duplicate in R table
- To fix: Break mega table in two + continue until you have identified all the functional deps

example:

Apply(SSN, sName, cName) becomes Student(SSN, sName) and Apply(SSN, cName)

Multivalued dependency ← Notation SSN ->> cName, SSN ->>HS

- compliant with BCNF (no functional dependency)
- Multiplicative effect (due to redundance of storage)
  - → Redundance, update, deletion anomalies
  - store relations only once!
  - C * H entries (one mega table) instead of C + H entries (different tables)

!!! Multivalued dependency depends on the meaning or represented world !!!
!!! Make sure you understand the represented world !!!

(In) Fourth normal Form (if follows) ← aka FNF

- Stricter than Boyce-Codd Normal Form
- Remove Multivalued dependency
- if A ->> B then A is key ← Key means no duplicate in R table

example:

Apply(SSN, cName, HS) ← store SSN, HS for each application at cName
becomes Apply(SSN, cName) and HighSchool(SSN, HighSchool)

INTRODUCTION TO DATABASE
- RELATIONAL DESIGN THEORY
- FUNCTIONAL DEPENDENCY

Relational design by decomposition
Functional dependencies are useful concept for data storage (compression)
reasoning about queries (optimization).

FD defines keys: A ⇒ B and table is (A,B), then A is a key!!
← a key can be more than one attribute

Functional dependency (formal definition)
!!! Based on the knowledge of the real world !!!
VV t1,t2 in R , t1.[a1,..aN] = t2.[a1,...,aN] ⇒ t1.[b1,...,bM] = t2.[b1,...,bM]
← t1, t2 are tuples (entries in table)
← R a table/relation defined as R(A,B,C)

VV t1,t2 in R, t1.A = t2.A ⇒ t1.B = t2.B

← A set of attributes, B a different set of attributes
← B could be A or part of it !

!!! t1.C can be = or != to t2.C with C another set of attributes!!!

HSname, HScity ⇒ HScode ← HSname and HScity FULLY determine HScode
← HSname and HScity are independent
← the notiation '⇒' means 'fully determine(s)'

Trivial functional dependency
VV t1,t2 in R, t1.A = t2.A ⇒ t1.B = t2.B with B subset of A
← trivial because always true !

Non-trivial function dependency
A ⇒ B non trivial if B not a subset of A
← intersection between A and B may not be empty

Completely nontrivial FD
A ⇒ B and intersection = 0 ← B a complete different subset than A

Special case of functional dependency
We possibly could have A ⇒ B and B ⇒ A
example:
HighSchool(HScode, HSname, HScity)
with A = HScode and B = (HSname, HScity)

Rules for functional dependencies
A ==> B, then A ⇒ b1, A ⇒ b2, etc ← split of RHS

!!! you cannot split the LHS !!!
- example: HScity, HSname ⇒ HScode

A ⇒ b1 and A ⇒ b2, then A ⇒ b1,b2 ← combining rule

A ⇒ B, then A ⇒ A union B ← add LHS on RHS
A ⇒ B, then A ⇒ A inter B ← trivial function !!!
A ⇒ B, B ⇒ C, then A ⇒ C ← transitive rule

Closure attributes ← **closure of A is A+**
Find all B such as A → B ← A and FDs are given, find B
start with { a1, ...aN}
repeat until no change:
if A ⇒ bi and A in set, then add bi in set

example:
Student(SSN, sName, address, HScode, HSname, HScity, GPA, priority)
we know:
SSN ⇒ sName, address, GPA

GPA ⇒ priority
HScode ⇒ HSname, HScity

find closure of { SSN, HScode } or {SSN, HScode}+
{ SSN, sName, address, GPA, HScode, HSname, HScode, priority}
← all attributes of the relation!!!
← { SSN, HSname} is a key of the relation!!!

!!! Closure help us determine if a set of attributes is a key for the relation !!!

Is A a key for R?
Compute A+
- if = all attributes, then A is a key

How can we find all keys given a set of FD?
- consider every subset of attributes in increasing size

!!! Trick: If one attribute is never determine by others, then it is part of the key !!!
Example: R(A,B,C,D,E) with following FDs

AB ⇒ C
AE ⇒ D ← A and E are never determined by a set of attributes
D ⇒ B

Specifying FDs for relation
S1 and S2 sets of FD
S2 'follows from' S1 if every relation instance satisfying S1 also satisfies S2
example:

S1 = { AB ⇒ C, AE ⇒ D, D ⇒ B } ← S1 ⇒ S1
S2 = { AD ⇒ C, AE ⇒ B } ← S1 ⇒ S2
S3 = { AD ⇒ C } ← doesn't have to include all
S4 = {ABC ⇒ D } ← S3 doesn't follow from S1

Two Sets of functional dependencies are equivalent
if S1 follows from S2 and S2 follows from S1

INTRODUCTION TO DATABASE
- RELATIONAL DESIGN THEORY
- FUNCTIONAL DEPENDENCY

Review: cross operation
R1 tuple x R2 tuple where attributes are same?
R1: | A | B | and R2: | B | C| produce R: | A | B | C | with 123, 125, 423, 425 entries
1 | 2 2 | 3
4 | 2 2 | 5
← Beware of lossless join!

Relational design by decomposition
R(a1, ...aN) ==
- R1(b1,...bM) and R2(c1...cP) ← or R1(B) and R2(C)
- with A =B union C ← B and C can have attributes in common (keys?)
- R1 cross R2 = R

R1 = \project_{B} (R)
R2 = \project_{C} (R)

example:
Student(SSN,sName, address, HScode, HSname, HScity, GPA, priority)
* Decomposition #1
S1(SSN, sName, addr, HScode, GPA, priority) ← R1
S2(HScode, HSname, HScity) ← R2
* Decomposition #2
S1(SSN, sName, address, HScode, HSname, HScity)
S2(sName, HSname, GPA, priority)
← Join doesn't work because not unique values
← we may have new entries appear !!!
← Bad decomposition (because join on not key)
!!! Reassembly should produce the original relation !!!

In Boyce-Codd Normal Form?
In Boyce-Codd Normal Form the LHS of all the FDs contain a key
!!! That key doesn't need to be the declared key in our database!!!
!!! A key is a set of attributes, that given FDs give all the attributes!!! ← see closures

example:
Student(SSN, sName, address, HScode, HSname, HScity, GPA, priority)
FD:
SSN ⇒ sName, address, GPA
GPA ⇒ priority
HScode ⇒ HSname, HScity
Closure (key) = { SSN, HScode }

Does every FD have a key on the LHS? No! So no in BCNF !!!
← we need to split the R in two !

example:
Apply(SSn, cName, state, date, major)

FD:

- SSN, cName, state ⇒ date, major

Ok, in BCNF ! {SSn, cName} is a key!!

Algorithm

input: relation R and FDs for R
output: decomposition of R into BCNF relations with 'lossless join'

Compute keys for R (using FDs) ← The FDs are constants (i.e world view)
Repeat until all relations are in BCNF:

Pick any R' with A ⇒ B that violates BCNF
Decompose R' into R1(a,B) and R2(A, rest) ← split in 2 tables
Compute FDs for R1 and R2 ← Use closure !!! Not R's FDs!!!
Compute keys for R1 and R2

Reasoning:

R: | A | B | rest | = R1: | A | B | \cross R2: | A | rest |

example:
Student(SSN, sName, address, HScode, HSname, HScity, GPA, priority)
FD:
SSN ⇒ sName, address, GPA
GPA ⇒ priority
HScode ⇒ HSname, HScity

First iteration: (start with HScode ⇒ HSname, HScity)
S1(HScode, HSname, HScity) ← S1 BCNF ok !
S2 (SSN, sName, address, HScode, GPA, priority) ← no HSname, HScity, but HScode
← S2 not in BCNF!
Second iteration: (use GPA ⇒ priority)

S1
S2 is decomposed as follow

- S3 (GPA, priority) ← S3 BCNF ok !
- S4 (SSN, sName, address, HScode, GPA) ← S4 not in BCNF !

…

S4 is decomposed as follow

S5 (SSN, sname, addr, GPA) ← S5 BCNF ok !
S6 (SSN, HScode) ← S6 BCNF ok!

So database schema, use the following relation: S1, S3, S5, S6

!!! There is not one only BCNF decomposition (function of which FD you start with) !!!

INTRODUCTION TO DATABASE
- RELATIONAL DESIGN THEORY
- MULTIVALUED DEPENDENDECIES & 4th NORMAL FORM

Relational design by decomposition (Review)
1/ mega relations
2/ system decomposes based on properties ← a.k.a constraints, world model
- Functional dependencies → Boyce-Codd Normal Form
- Multivalued dependencies → Fourth Normal From

3/ final set of relations satisfies normal form
- no anomalies
- no lost information

Example:
Apply(SSN, cName, hobby) ← reveal all hobbies to each college
FDs ? No ← No FD possible !
Keys? All attributes ← key = all attributes !!!
BCNF? Yes ← No FD, so yes ! ;-)
Good design? No, ex 1 person applies to 5 colleges and 6 hobbies = 30 tuples

⇒ separation of independent facts = 4th Normal Form
⇒ separate college and hobby

Multivalued dependency
* Based on knowledge in read world
* All instances of relation must adhere
- R A ->> B ← A multidetermines B
with A, B set of attributes ← a1,...,aN and b1, … bM
← A multidetermines B

VV t1, t2 in R : t1[A] = t2[A] ,
then E t3 in R such as t3[A] = t1[A] and t3[B] = t1[B] and t3[rest] = t2[rest]
← t1, t2, t3 : 3 tuples

	A	B	rest	
				← SSN, cName, Hobby
t1	a	b1	r1	
t2	a	b2	r2	
t3	a	b1	r2	← B independent from rest
but also				← College independent from Hobby
t4	a	b2	r1	← Guaranteed to be there !?!?!?

!!! You need at least 2 tuples in table that starts with the same A, then find different B, and do cross (check C) !!!

R(A,B,C) with multivalued dependency A ->> B,
then minimum number of tuple in R is product of :
of <> values for A * # of <> values for B * # of different values for C

!!! if A ->> B, then we also have A ->> rest !!!

Example:
Apply(SSN, cName, hobby)

	SSN	cName	hobby	
t1	123	stanford	trumpet	
t2	123	berkeley	tennis	
t3	123	stanford	tennis	← reveal hobbies to all college !
and also				
t4	123	berkeley	trumpet	

Here we have SSN ->> cName, but also SSN ->> hobby

Modified example:
Apply(SSN, cName, hobby)
(1) but reveal hobbies to colleges selectively ← small detail, big difference !!!

* Multivalued dependencies (MVDs) ? None
* Good design? Yes

Expanded example:
Apply(SSN, cName, date, major, hobby) ← date: date of application to college
(1) Reveal hobbies to college selectively ←
(2) Apply once to each college (on one day) ← <> major at <> college ok
← <> major at a single college

May apply to multiple majors

Assume major are independent from hobby as well!

FD: SSN, cName ⇒ date ← SSN, cName fully determine date
← SSN, cName ha sonly one date
← SSN and cName are indepedent
MVDs : SSN, cName, date ->> major ← rest = hobby, indep from major
A: SSN, cName, date ->> B: major ← # of entries: #A * #B * #C

Trivial multivalued dependency
A ->> B
if B included in A ← ??????
or if A union B = all attributes ← rest is EMPTY

Nontrivial MVD
all other cases ;-)

Rules for Multivalued Dependencies
FD-is-an-MVD rule
if A → B, then A ->> B ← Very important!
Intersection rule
A ->> B and A ->> C then A → B intersection C

Transitive rule

A ->> B and B ->> C then A → C - B

Every rule for MVDs is a rule for FDs

because FD-is-MVD rule!!

Fourth normal Form

Relation R with MVDs is in 4NF if:

for each nontrivial A ->> B, A is a key

Algorithm

input: relation R and FDs for R + MVDs for R

output: decomposition of R into 4NF relations (not BCNF relations) with ‘lossless join’

Compute keys for R (using FDs) ← The FDs are constants (i.e world view)

Repeat until all relations are in 4NF (not BCNF):

Pick any R’ with non trivial A->> B (not A ⇒ B) that violates 4NF (not BCNF)

Decompose R’ into R1(A,B) and R2(A, rest) ← split in 2 tables

Compute FDs and MVDs (!!) for R1 and R2 ← Use closure !!! Not R’s FDs!!!

Compute keys for R1 and R2

Example:

Apply(SSN, cName, hobby)

- SSN ->> cName
- no keys for R except all attributes

A1(SSN, cName)

A2(SSN, hobby)

← no FDs and no MVDs so in 4NF

Example 2:

Apply(SSN, cName, date, major, hobby)

- (FD) SSN, cName ⇒ date
- (MVD) SSN, cName, date ->> major
- no keys for R except all attributes

Using MVD, we decompose: ← decompose first using MVDs

A1(SSN, cName, date, major)

A2(SSN, cName, date, hobby)

Using FD, we decompose

A1 becomes

A3 (SSN, cName, date)

A4 (SSN, cname, major)

A2 becomes

A5 (SNN, cName, date) ← A3 == A5 !

A6 (SSN, cName, hobby)

So we have decomposed and the 4NF of R is A3, A4,A6

INTRODUCTION TO DATABASE
- RELATIONAL DESIGN THEORY
- SHORTCOMING OF BCNF AND 4NF

Review:
BCNF:
    Relation R with FDs is in BCNF if
        for each A ⇒: B, A is a key (or includes a key)

4NF:
    Relation R with MVDs is in 4NF if:
        for each nontrivial A->>B, A is a key

Example:
Apply (SSN, cName, date, major)
(1) Can apply to each college once for one major
    Student can apply to each college once
    Student can apply to each college for one major
(2) Colleges have non-overlapping application dates
    Date is unique
FDs: (1) SSN, cName ⇒ date, major
    (2) date ⇒ cName ← Not a key in the LHS, so not BCNF
Keys: SSN, cName
BCNF: No
    A1 (date, cName)
    A2 (SSN, date, major)
Good design?
    A1, A2 in BCNF, but still not necessarily a good design !
    To check (1), we will need to do a join of A1 and A2
    cName and SSN not in same table !!!!

Example II:
Apply(SSN, HSname, GPA, priority)
(1) Multiple HS okay
(2) priority determined from GPA
FDs: (1) SSN ⇒ GPA
    (2) GPA ⇒ priority
    (1) + (2) SSN ⇒ priority
Keys: SSN, HSname
BCNF? No
    S1(SSN, priority)
    S2(SSN, HSname, GPA)
        S3(SSN, GPA)
        S4(SSN, HSname)
Good design?
    S1, S3, S4 are BCNF
    But priority is not in same table as GPA !

So no, because after decomposition,
no guarantee dependencies can be checked on decomposed relations
(require join to check them !) ← Dependency enforcement

Example III
Scores(SSN, sName, SAT, ACT)
(1) Multiple SATs and ACTs allowed

FDs: SSN ⇒ sName
no key
MVDs: SSN, sName ⇒ SAT
4NF? no
S1(SSN, sName, SAT)
S3(SSN, sName)
S4(SSN, SAT)
S2(SSN, sname, ACT)
S5(SSN, ACT)
Good design?
If every wuery returns name + composite score for SSN, then we need ‘joins’ all the time!
Use a denormalized table ! ← Query workload

Example IV:
colleg(cName, state)
CollegeSize(cName, enrollment)
CollegeScores(cName, avgSAT)
CollegeGRades(cName, avgGPA)

BCNF/4NF? Yes
Good design? Not necessarily.... too decomposed!! ← overdecomposition

---