CSCI 127
Introduction to Database Systems

Integrity Constraints and Functional Dependencies

Integrity Constraints

Purpose:

Prevent semantic inconsistencies in data

e.g.:

cname	svngs	check	total
Joe	100	200	250

total ≠ savings + checking

e.g.:

cname	bname
Joe	Waltham

bname	bcity
Dntn	Bkln
…	…

No entry for Waltham

Integrity Constraints

What Are They?

- *Predicates on the database*
- *Must always be true (checked whenever db gets updated)*

The 4 Kinds of IC's:

1. *Key Constraints (1 table)*

   e.g.: 2 `accts` can't share same `acct_no`

2. *Attribute Constraints (1 table)*

   e.g.: `accts` must have nonnegative balance

3. *Referential Integrity Constraints (2 tables)*

   e.g.: `bnames` associated with loans must be names of real branches

4. *Global Constraints (n tables)*

   e.g.: all `loans` must be carried by at least 1 customer with a savings account

Key Constraints

Idea:

Specifies that a relation is a set, not a bag

SQL Examples:

1. *Primary Key*

```
CREATE TABLE branch(
   bname CHAR(15) PRIMARY KEY
   bcity CHAR (50),
   assets INTEGER);
```

OR

```
CREATE TABLE depositor(
   cname CHAR(15),
   acct_no CHAR(5),
   PRIMARY KEY (cname, acct_no));
```

Key Constraints (cont.)

Idea:

Specifies that a relation is a set, not a bag

SQL Examples (cont.):

2. *Candidate Key*

```
CREATE TABLE customer(
   ssn CHAR(19),
   cname CHAR(15),
   address CHAR(30),
   city CHAR(10),
   PRIMARY KEY (ssn),
   UNIQUE (cname, address, city));
```

Key Constraints (cont.)

Effect of SQL Key Declarations

`PRIMARY` $(A_1, \ldots, A_n)$ OR `UNIQUE` $(A_1, \ldots, A_n)$

1. Insertions:

   Check if inserted tuple has same values for $A_1, \ldots, A_n$ *as any previous tuple. If found, reject insertion*

2. Updates to any of $A_1, \ldots, A_n$:

   Treat as insertion of entire tuple

Key Constraints (cont.)

Effect of SQL Key Declarations (cont.)

`PRIMARY (`A_1`,…,`A_n`)` OR `UNIQUE (`A_1`,…,`A_n`)`

Primary vs. Unique (candidate):

1. *One primary key per table.*
   Several unique keys allowed.

2. *Only primary key can be referenced by "foreign key"*
   (Referential integrity)

3. *DBMS may treat these differently (e.g.: Putting index on primary key)*

Attribute Constraints

Idea:

- *Attach constraints to value of attribute*
- *"Enhanced" type system*

  (e.g.: > 0 rather than integer)

In SQL:

1. NULL

```
CREATE TABLE branch(
    bname CHAR(15) NOT
NULL
    …
)
```

2. CHECK

```
CREATE TABLE depositor(
    …
    balance integer NOT NULL
    CHECK (balance ≥ 0)
    …
)
```

any WHERE *clause OK here*

⇒ *affect insertions, updates in affected columns*

Attribute Constraints (cont.)

Domains:

Can associate constraints with DOMAINS *rather than attributes*

e.g.: Instead of:

```
CREATE TABLE depositor(
    …
    balance integer NOT NULL
    CHECK (balance ≥ 0)
    …
)
```

One can write…

Attribute Constraints (cont.)

Domains (cont):

```
CREATE DOMAIN bank-balance integer(
   CONSTRAINT not-overdrawn
      CHECK (value ≥ 0),
   CONSTRAINT not-null-value
      CHECK (value NOT NULL)
)

CREATE TABLE depositor(
   …
   balance bank-balance
   …
)
```

Q: *What are the advantages of associating constraints w/ domains?*

Attribute Constraints (cont.)

Advantages of Associating Constraints with Domains:

1. *Can avoid repeating specification of same constraint for multiple columns*
2. *Can name constraints* *e.g.:*

```
CREATE DOMAIN bank-balance integer(
   CONSTRAINT not-overdrawn
      CHECK (value ≥ 0),
   CONSTRAINT not-null-value
      CHECK (value NOT NULL))
```

Allows One To:

1. *Add or remove:*

```
ALTER DOMAIN bank-balance
   ADD CONSTRAINT capped
      (CHECK value ≤ 10000)
```

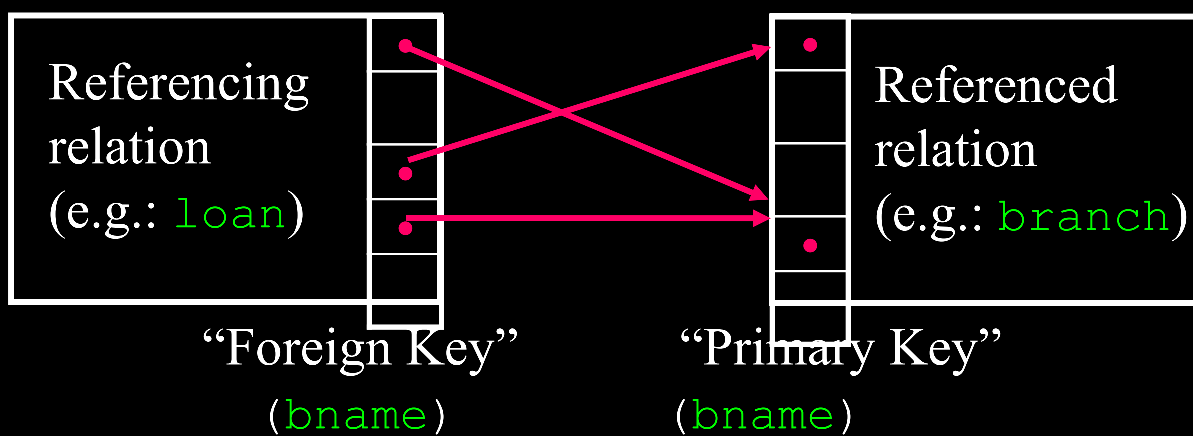
2. *Report better errors (know which constraint violated)*

Referential Integrity Constraints

Idea:

Prevent "dangling tuples" (e.g.: A loan with `bname`*,* Waltham *when no* Waltham *tuple in* `branch`*)*

Illustrated:



Referential Integrity:

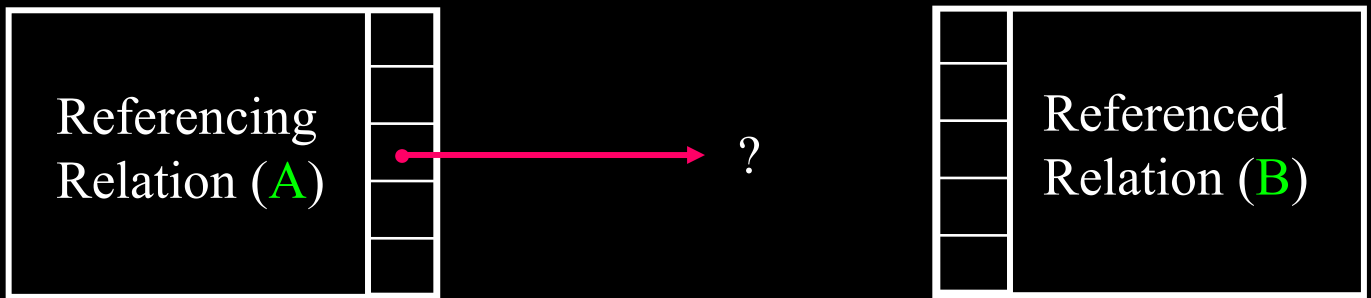
Ensure that: Foreign Key $\xrightarrow{\text{Corr. to}}$ *Primary Key value*

Note: Need not ensure $\longleftarrow$

(i.e.: Not all branches must have loans)

Referential Integrity Constraints

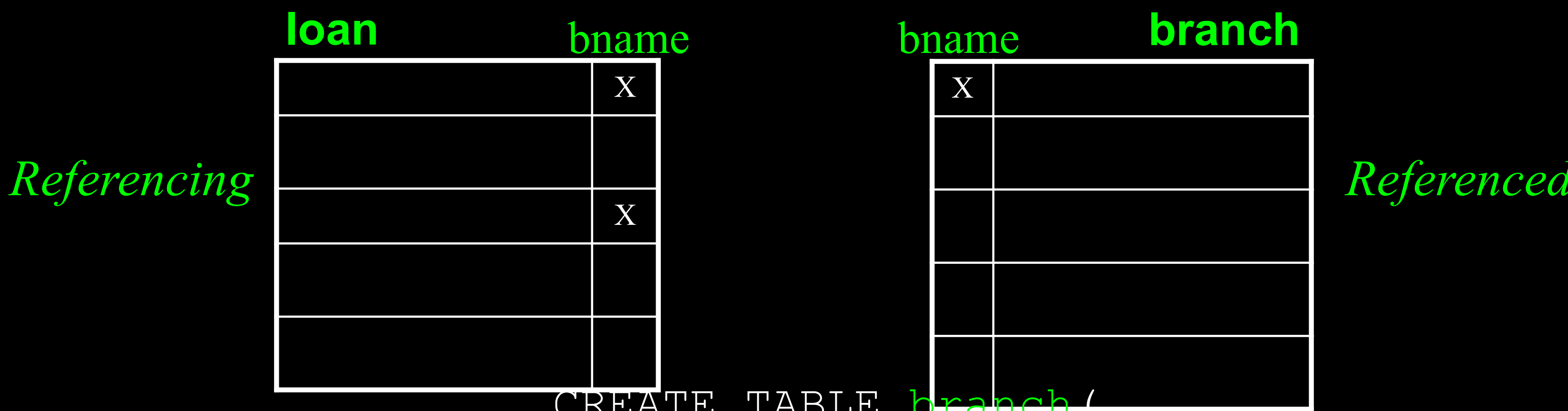
Q: Why are dangling references bad?



A: Think E/R Diagrams. In what situation do we create table A (with column containing keys of table B)

1. A *represents a relationship with* B*,*
   or is an entity set with an n:1 relationship with B
2. A *is a weak entity dominated by* B
   (d.r. violates weak entity condition)
3. A *is a specialization of* B *(dang.ref. violates inheritance tree)*

Referential Integrity Constraints



In SQL, Declare:

```
CREATE TABLE branch(
   bname CHAR(15) PRIMARY KEY

   ...)
CREATE TABLE loan(
   ...

   FOREIGN KEY bname REFERENCES branch)
```

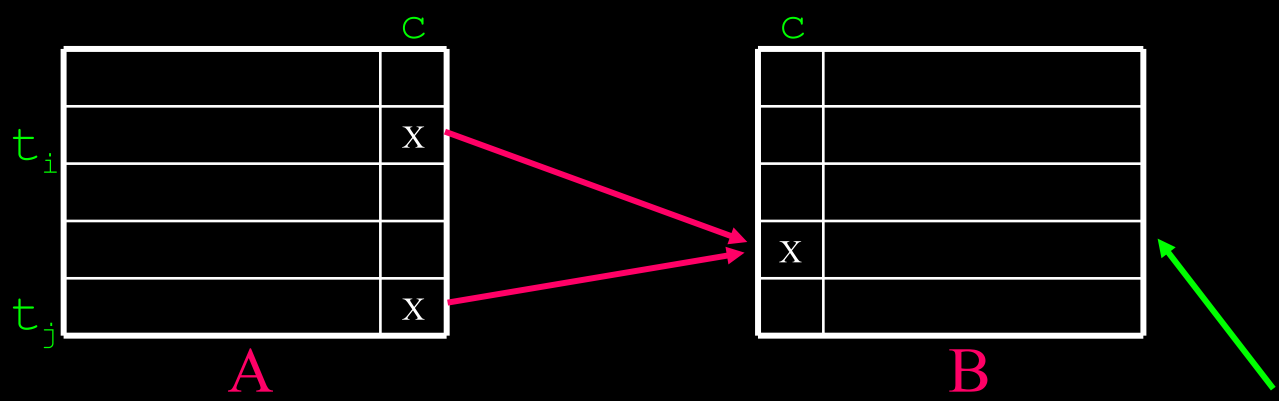
Affects:

1. *Insertions, updates of* *referencing relation*
2. *Deletions, updates of* *referenced relation*

Ensure no tuples in referencing relation left dangling

Referential Integrity Constraints

Q: What happens to tuples left dangling as a result of deletion/update of referenced relation?



What happens when we try to delete this tuple?

A: 3 Possibilities

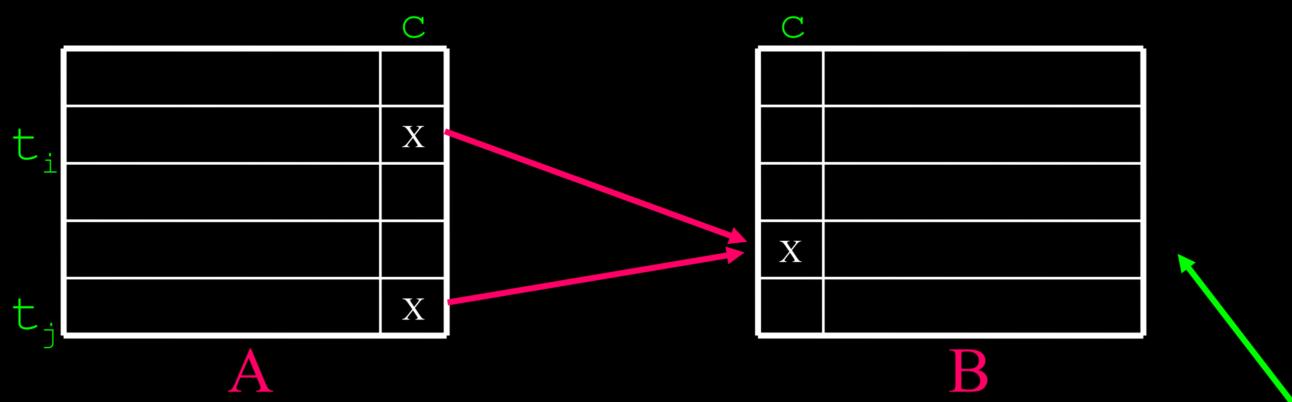
1. *Reject deletion/update*
2. *Set* `t`$_i$`[c]` and `t`$_j$`[c] = NULL`
3. *Propagate deletion/update*

`DELETE: delete` t_i, t_j

`UPDATE: set` t_i

Referential Integrity Constraints

Resolving Dangling Tuples



What happens if I try to delete/update this tuple?

In SQL:

```
CREATE TABLE A(…
    FOREIGN KEY C REFERENCES B <action>
    …)
```

Referential Integrity Constraints

Resolving Dangling Tuples (cont.)

Deletion:

1. *(Left blank): Deletion/update rejected*

2. `ON DELETE SET NULL / ON UPDATE SET NULL`
   sets $t_i[c]$ = NULL, $t_j[c]$ = NULL

3. `ON DELETE CASCADE`
   delete t_i, *delete* t_j

   `ON UPDATE CASCADE`
   sets $t_i[c]$, $t_j[c]$ *to new Key value*

Global Constraints

Idea:

1. *Single relation (constraint spans multiple columns)*

   e.g.: `CHECK (total = svngs + check)`

   declared in `CREATE TABLE` for relation

2. *Multiple relations* `CREATE ASSERTIONS`

Global Constraints (cont.)

SQL Example (cont.):

Multiple relations: Every loan has a borrower with a savings account

```
CHECK (NOT EXISTS(
    SELECT *        EMPTY when inner query is EMPTY for all l
    FROM loan AS l
    WHERE NOT EXISTS(
        SELECT *
        FROM borrower AS b, depositor AS d, account AS a,
        WHERE b.cname = d.cname AND d.acct_no = a.acct_no
            AND l.lno = b.lno)))
```

EMPTY when JOINs are broken

```
SELECT *            FROM loan AS l          WHERE <non-c
```

Global Constraints (cont.)

SQL Example (cont.):

Multiple relations: Every loan has a borrower with a savings account (cont.)

Problem:

With which table's definition does this go?
(`loan`*?,* `depositor`*?,…)*

A: *None of the above*

```
CREATE ASSERTION loan-constraint
   CHECK (NOT EXISTS…)
```

Checked with <u>EVERY</u> DB update! VERY EXPENSIVE…

Integrity Constraints: Summary

Constraint	Where Declared	Affects…	Expense
Key Constraints	`CREATE TABLE (PRIMARY KEY, UNIQUE)`	*Insertions, updates*	*Moderate*

Integrity Constraints: Summary

Constraint	Where Declared	Affects…	Expense
Key Constraints	`CREATE TABLE (PRIMARY KEY, UNIQUE)`	*Insertions, updates*	*Moderate*
Attribute Constraints	`CREATE TABLE` `CREATE DOMAIN (NOT NULL, CHECK)`	*Insertions, updates*	*Cheap*

Integrity Constraints: Summary

Constraint	Where Declared	Affects…	Expense
Key Constraints	`CREATE TABLE (PRIMARY KEY, UNIQUE)`	*Insertions, updates*	*Moderate*
Attribute Constraints	`CREATE TABLE` `CREATE DOMAIN (NOT NULL, CHECK)`	*Insertions, updates*	*Cheap*

Referential Integrity	*Table tag*
	`(FOREIGN KEY`
	`REFERENCES …)`
		1. Insertions into referencing relation
		2. Updates of referencing relation of relevant att's
		3. Deletions from referenced relations
		4. Updates of referenced relations
			1,2: Like key constraints. Another reason to index/sort on primary keys
			3,4: Depends on
			a. update/delete policy chosen
			b. Existence of indexes on foreign keys

Integrity Constraints: Summary

Constraint	Where Declared	Affects…	Expense
Key Constraints	`CREATE TABLE (PRIMARY KEY, UNIQUE)`	*Insertions, updates*	*Moderate*
Attribute Constraints	`CREATE TABLE` `CREATE DOMAIN (NOT NULL, CHECK)`	*Insertions, updates*	*Cheap*
Referential Integrity	`(FOREIGN KEY REFERENCES …)`	*1. Insertions into referencing relation* *2. Updates of referencing relation of relevant att's* *3. Deletions from referenced relations* *4. Updates of referenced relations*	*1,2: Like key constraints. Another reason to index/sort on primary keys* *3,4: Depends on* *a. update/delete policy chosen* *b. Existence of indexes on foreign keys*
Global Constraints	*Outside tables (create assertion)*	*1. For single relation constraint, with insertions, updates of relevant att's* *2. For assertions, with every database modification*	*1. Cheap* *2. Very Expensive*

Functional Dependencies

An Example:

loan-info =

bname	lno	cname	amt
Dntn	L-17	Jones	1000
Dntn	L-17	Williams	1000
Redwood	L-23	Smith	1000
Perry	L-15	Hayes	1500
Redwood	L-23	Johnson	1000

True or False?

`amt` → `lno`?
`lno` → `cname`?
`lno` → `lno`?
`bname` → `lno`?

Can't always decide by looking at populated db's

Observe:

Tuples with the same value for `lno` *will always have the same value for* `amt`

We write: `lno` → `amt`
(`lno` *"determines"* `amt`*, or* `amt` *is "functionally determined" by* `lno`*)*

Functional Dependencies

In general:

$A_1, \ldots, A_n \rightarrow B$

Informally:

If 2 tuples "agree" on their values for $A_1, \ldots, A_n$*, they will also agree on their values for* B

Formally:

$$\forall t,u \ (t[A_1] = u[A_1] \wedge t[A_2] = u[A_2] \wedge \ldots \wedge t[A_n] = u[A_n] \Rightarrow t[B] = u[B])$$

Functional Dependencies

Another Example:

Drinkers

name	addr	likes	lmanf	fave	fmanf
Homer	WS	Bud	AB	Duff	SB
Homer	WS	Duff	SB	Duff	SB
Apu	ES	Bud	AB	Bud	AB

What are the FD's?

likes $\rightarrow$ lmanf
fave $\rightarrow$ fmanf
name $\rightarrow$ fave
name $\rightarrow$ addr (?)

Back to Global Integrity Constraints

How Do We Decide What Constraints to Impose?

Consider Drinkers (name, addr, likes, lmanf, fave, fmanf)
with FD's: name → addr, …

Q: How do we ensure that name → addr?

A:

```
CREATE ASSERTION name-addr
    CHECK (NOT EXISTS
        (SELECT *
         FROM Drinkers AS d1, Drinkers AS d2
         WHERE ?))
```

? ≡ d_1.name = d_2.name AND d_1.addr <> d_2.addr

Back to Functional Dependencies

How to derive them?

1. *Key Constraints*
   (e.g.: `bname` a key for `branch`)

   Therefore: `bname → bname`
   `bname → city`
   `bname → assets`

   will instead write:
   `bname → bname bcity assets`

Q: *Define "Super Keys" in terms of FD's*

A: *Any set of attributes in a relation that functionally determines all attributes in the relation*

Q: *Define "Candidate Key" in terms of FD's*

A: *Any super key such that the removal of any attribute leaves a set that does not functionally determine all attributes*

Functional Dependencies

How to Derive Them?

1. *Key Constraints*
2. *n:1 relationships*

   e.g.: `beer → manufacturer`, `beer → price`
3. *Laws of Physics*

   e.g.: `time room → course`
4. *Trial-and-error*

Given `R = (A, B, C)`, *try each of the following to see if they make sense.*

A → B	C → A	BC → A
A → C	C → B	
B → A	AB → C	
B → C	AC → B	

What about?

AB → A

C → C

Just write: "... plus all of the trivial dependencies"

Back to Global IC's

2. Avoiding the Expense

Recall: name → addr *preserved by*

```
CHECK (NOT EXISTS
    (SELECT *
     FROM Drinkers AS d1, Drinkers AS d2
    WHERE d1.name = d2,name AND d1.addr <> d2.addr))
```

Q: *Is it necessary to have an assertion for every FD?*

A: *Luckily, no. Can preprocess FD set*
Some FD's can be eliminated
Some FD's can be combined

Functional Dependencies

Combining FD's:

a. `name → addr`

```
CREATE ASSERTION name-addr
    CHECK (NOT EXISTS
        (SELECT *
         FROM Drinkers AS d1, Drinkers AS d2
        WHERE d1.name = d2.name AND d1.addr <> d2.addr))
```

b. `name → fave`

```
CREATE ASSERTION name-fave
    CHECK (NOT EXISTS
        (SELECT *
         FROM Drinkers AS d1, Drinkers AS d2
        WHERE d1.name = d2.name AND d1.fave <> d2.fave))
```

Functional Dependencies (cont.)

Combining FD's (cont.):

Combine into: `name` → `addr fave`

```
CREATE ASSERTION name-addr
    CHECK (NOT EXISTS(SELECT *
                      FROM Drinkers AS d1, Drinkers AS d2
                      WHERE d1.name = d2.name AND ?))
```

```
? ≡ (d1.addr <> d2.addr) OR (d1.fave <> d2.fave)
```

Functional Dependencies

Determining Unnecessary FD's

Consider: name → name

```
CREATE ASSERTION name-name
   CHECK(NOT EXISTS
      (SELECT *
       FROM Drinkers AS d1, Drinkers AS d2
       WHERE d1.name = d2.name AND d1.name <> d2.name))
```

Cannot possibly be violated!

Functional Dependencies

Note:

$X \rightarrow Y$ *s.t.* Y $\supseteq$ X *is a "trivial dependency"*
(true, regardless of attributes involved)

Moral:

Don't create assertions for trivial dependencies

Functional Dependencies

Determining Unnecessary FD's

Even non-trivial FD's can be unnecessary

e.g.:

1. name → fave

```
CREATE ASSERTION name-fave
   CHECK (NOT EXISTS
      SELECT *
      FROM Drinkers AS d1, Drinkers AS d2
      WHERE d1.name = d2.name AND d1.fave <> d2.fave)
```

2. fave → fmanf

```
CREATE ASSERTION fave-fmanf
CHECK (NOT EXISTS                              SELECT *
                                                 FROM
Drinkers AS d1, Drinkers AS d2
      WHERE d1.fave = d2.fave AND d1.fmanf <> d2.fmanf)
```

Functional Dependencies (cont.)

Determining Unnecessary FD's (cont.)

Even non-trivial FD's can be unnecessary (cont.)

e.g.:

3. name → fmanf

```
CREATE ASSERTION name-fmanf
  CHECK (NOT EXISTS
    SELECT *
        FROM Drinkers AS d1, Drinkers AS d2
    WHERE d1.name = d2.name AND d1.fmanf <> d2.fmanf)
```

Note: *If 1 and 2 succeed, 3 must also*

Functional Dependencies

Using FD's to Determine Global IC's:

Step 1: Given schema $R = \{A_1, \ldots, A_n\}$

Use key constraints, n:1 relationships, laws of physics and trial-and-error to determine an initial FD set, F

Step 2:

Use FD elimination techniques to generate an alternative (but equivalent) FD set, F'

Step 3:

Write assertions for each $f \in F'$ *(for now)*

Functional Dependencies

Using FD's to Determine Global IC's (cont.):

<u>Issues:</u>

1. *How do we guarantee that* `F = F'` *?*

   A: *Closures*

2. *How do we find a "minimal"* `F = F'` *?*

   A: *Canonical cover algorithm*

Functional Dependencies

Example:

Suppose:

$R = \{A, B, C, D, E, H\}$ *and we determine that:*

$F = \{A \rightarrow BC, B \rightarrow CE, A \rightarrow E, AD \rightarrow H, D \rightarrow B\}$

Then we determine the canonical cover of F:

$$F_c = \{A \rightarrow BH, B \rightarrow CE, D \rightarrow B\}$$

ensuring that F *and* F_c *are equivalent*

Note:

F *requires 5 assertions*

F_c *requires 3 assertions*

Functional Dependencies

Equivalence of FD Sets:

FD sets `F, G` *are equivalent if they* ***imply*** *the same set of FD's*

e.g.: $\left.\begin{array}{l} A \rightarrow B \\ B \rightarrow C \end{array}\right\}$ *Implies* $A \rightarrow C$

Equivalence usually expressed in terms of closures

Closures:

For any FD set, `F`, F^+ *is the set of all FD's implied by* `F`.

Can calculate in 2 ways:

1. Attribute closures
2. Armstrong's axioms

Both techniques are tedious → *we will do only for toy examples*

Note: `F` *equivalent to* `G` *if and only if* $F^+ = G^+$

Functional Dependencies

A	B	C	D
a	α	1	u
a	α	1	u
a	β	5	w
b	β	3	w
b	β	3	w

Shorthand:

$C \rightarrow BD$ *same as* $C \rightarrow B$

Be Careful!

C

$\rightarrow$ D

$AB \rightarrow C$ *not the same as* $A \rightarrow C$

$B \rightarrow C$

not true

Attribute Closures

Given:

```
R = {A, B, C, D, E, H}
F = {A → BC,
     B → CE,
     A → E,
     AC → H,
     D → B}
```

Q: What is the closure of CD (i.e., CD^+)?

A: The set of attributes
   that can be determined from CD.

Attribute Closures (cont.)

Q: What is the closure of `CD` (i.e., CD^+)?

A:
```
Algorithm attr-closure (X: set of attributes)
        result ← X
        repeat until stable
            for each FD in F,Y → Z, do
                if Y ⊆ result then
                    result ← result U Z
```

e.g.: `attr-closure (CD)`

Iteration	Result
0	CD

```
R = {A, B, C, D, E, H}
F = {A → BC,
     B → CE,
     A → E,
     AC → H,
     D → B}
```

Attribute Closures (cont.)

Q: What is the closure of `CD` (`CD`$^+$) ?

A:
```
Algorithm attr-closure (X: set of attributes)
    result ← X
    repeat until stable
        for each FD in F,Y → Z, do
            if Y ⊆ result then
                result ← result U Z
```

e.g.: `attr-closure (CD)`

Iteration	Result
0	CD
1	CDB

```
R = {A, B, C, D, E, H}
F = {A → BC,
     B → CE,
     A → E,
     AC → H,
     D → B}
```

Attribute Closures (cont.)

Q: What is the closure of `CD` (`CD`$^+$) ?

A:
```
Algorithm attr-closure (X: set of attributes)
    result ← X
    repeat until stable
        for each FD in F,Y → Z, do
            if Y ⊆ result then
                result ← result U Z
```

e.g.: `attr-closure (CD)`

Iteration	Result
0	CD
1	CDB
2	CDBE

```
R = {A, B, C, D, E, H}
F = {A → BC,
     B → CE,
     A → E,
     AC → H,
     D → B}
```

Attribute Closures

Q: What is ACD^+?

A: $ACD^+ \rightarrow R$

Q: How can you determine if ACD is a super key?

A: *It is if* $ACD^+ \rightarrow R$

Q: How can you determine if ACD is a candidate key?

A: *It is if:* $ACD^+ \rightarrow R$, and

None of ($AC^+ \rightarrow R$, $AD^+ \rightarrow R$, $CD^+ \rightarrow R$)

are true.

Using Attribute Closures To Determine FD Set Closures

Given:

$F = \{A \rightarrow BC,$
$\quad B \rightarrow CE,$
$\quad A \rightarrow E,$
$\quad AC \rightarrow H,$
$\quad D \rightarrow B\}$

$F^+ = \{A \rightarrow A^+, B \rightarrow B^+, C \rightarrow C^+, D \rightarrow D^+, E \rightarrow E^+, H \rightarrow H^+.$
$AB \rightarrow AB^+, AC \rightarrow AC^+, AD \rightarrow AD^+, AE \rightarrow AE^+, AH \rightarrow AH^+, BC \rightarrow BC^+, BD \rightarrow BD^+, \ldots\}$

To Decide if F, G Are Equivalent:

1. *Compute* F^+
2. *Compute* G^+
3. *Is 1 = 2?*

Expensive:

F^+ *has 63 rules (in general: $O(2^{|R|})$ rules)*

FD Closures Using Armstrong's Axioms

A. Fundamental Rules (W, X, Y, Z: sets of attributes)

1. *Reflexivity*
   If $Y \subseteq X$ then $X \rightarrow Y$

2. *Augmentation*
   If $X \rightarrow Y$ then $WX \rightarrow WY$

3. *Transitivity*
   If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

FD Closures Using Armstrong's Axioms (cont.)

B. Additional rules (can be proved from 1 through 3)

4. *Union*

   If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

5. *Decomposition*

   If $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$

6. *Pseudotransitivity*

   If $X \rightarrow Y$ and $WY \rightarrow Z$, then $WX \rightarrow Z$

FD Closures Using Armstrong's Axioms

Given:

```
F = {A → BC,  (1)
     B → CE,  (2)
     A → E,   (3)

     AC → H,  (4)

     D → B}   (5)
```

Exhaustively Apply Armstrong's Axioms to Generate F^+:

```
F+ = F ∪
   1. {(6) A → B, (7)  A → C}
      … decomposition on (1)
   2. {(8) A → CE}
      … transitivity on (6),(2)
   3. {(9) B → C, (10) B → E}
      … decomposition on (2)
   4. {(11) A → C, (12) A → E}
      … decomposition on (8)
   5. {(13) A → H}
      … pseudotransitivity on
        (1),(4)
      …
```

Functional Dependencies

Our Goal:

Given FD set, `F`*, find an alternative FD set,* `G`*, that is:*

1. Smaller
2. Equivalent

Bad News:

Testing $F \equiv G \ (F^+ = G^+)$ *is computationally expensive*

Good News: Canonical Cover Algorithm (CCA)

Given FD set, `F`*, CCA finds <u>minimal</u> FD set equivalent to* `F`

<u>minimal</u>: can't find another equivalent FD set with fewer FD's

Canonical Cover Algorithm

Given:

$$F = \{A \rightarrow BC,\ B \rightarrow CE,\ A \rightarrow E,\ AC \rightarrow H,\ D \rightarrow B\}$$

Determine canonical cover of F:

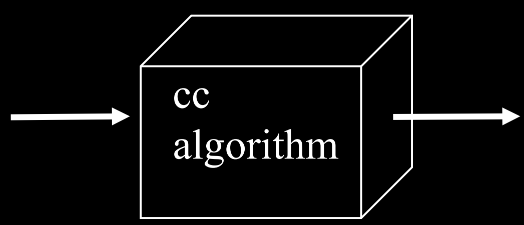
$$F_c = \{A \rightarrow BH,\ B \rightarrow CE,\ D \rightarrow B\}$$

$F_c = F$

No G that is equiv. to F is smaller than F_c

Another Example:

$$F = \{A \rightarrow BC,\ B \rightarrow C,\ A \rightarrow B,\ AB \rightarrow C,\ AC \rightarrow D\}$$



$$F_c = \{A \rightarrow BH,\ B \rightarrow C\}$$

Canonical Cover Algorithm

Basic Algorithm

```
ALGORITHM canonical-cover (X: FD Set)
   BEGIN
      REPEAT UNTIL STABLE                                    1.
         Where possible, apply UNION rule (A's
         Axioms)
            (e.g.: A → BC, A → CD becomes A → BCD)

         2. Remove "extraneous attributes" from each
              FD
            (e.g.: AB → C, A → B becomes A → B, B → C
          i.e.: A is extraneous in AB → C)
   END
```

Extraneous Attributes

1. Extraneous in RHS?

e.g.: Can we replace $A \rightarrow BC$ *with* $A \rightarrow C$*?*

(i.e.: Is B extraneous in $A \rightarrow BC$?)

2. Extraneous in LHS?

e.g.: Can we replace $AB \rightarrow C$ *with* $A \rightarrow C$*?*

(i.e.: Is B extraneous in $AB \rightarrow C$?)

Simple (but expensive) test:

1. Replace $A \rightarrow BC$ *(or* $AB \rightarrow C$*) with* $A \rightarrow C$ *in* F

Define $F_2 = F - \{A \rightarrow BC\} \cup \{A \rightarrow C\}$ OR

$F_2 = F - \{AB \rightarrow C\} \cup \{A \rightarrow C\}$

2. Test: Is $F_2^+ = F^+$*? If yes, then* B *was extraneous*

Extraneous Attributes

A. RHS: Is B extraneous in $A \rightarrow BC$?

Step 1: $F_2 = F - \{A \rightarrow BC\} \cup \{A \rightarrow C\}$

Step 2: $F^+ = F_2^+$?

To simplify step 2, observe that $F_2^+ \subseteq F^+$

(i.e.: no new FD's in F_2^+)

Why? *Have effectively removed* $A \rightarrow B$ *from* F

When is $F^+ = F_2^+$?

A: *When* $(A \rightarrow B) \in F_2^+$ (i.e., when you can deduce it from other FD's in F2)

Idea: *If* F_2^+ *includes:* $A \rightarrow B$ *and* $A \rightarrow C$, *then it includes* $A \rightarrow BC$

Extraneous Attributes

B. LHS: Is B extraneous in $AB \rightarrow C$?

Step 1: $F_2 = F - \{AB \rightarrow C\} \cup \{A \rightarrow C\}$

Step 2: $F^+ = F_2^+$?

To Simplify step 2, observe that $F^+ \supseteq F_2^+$

(i.e.: there may be new FD's in F_2^+)

Why?

$A \rightarrow C$ *"implies"* $AB \rightarrow C$.

Thus, all FD's in F^+ *also in* F_2^+.

But $AB \rightarrow C$ *does not "imply"* $A \rightarrow C$.

Thus, all FD's in F_2^+*, not necessarily in* F^+.

A → C in F_2^+ allows FD's That are not in F^+.

When is $F^+ = F_2^+$?

A: *When* $(A \rightarrow C) \in F^+$

Idea: *If* $(A \rightarrow C) \in F^+$, *then it will include all FD's of* F_2^+

Extraneous Attributes

A. RHS:

Given $F = \{A \rightarrow BC,\ B \rightarrow C\}$,
is C *extraneous in* $A \rightarrow BC$?

Why or why not?

A: *Yes, because*
$(A \rightarrow C) \in \{A \rightarrow B,\ B \rightarrow C\}^+$

Proof:
1. $A \rightarrow B$ Given
2. $B \rightarrow C$ Given
3. $A \rightarrow C$ transitivity, (1) and (2)

Use Armstrong's axioms in proof

Canonical Cover Algorithm

```
ALGORITHM canonical-cover (X: FD Set)
  BEGIN

    REPEAT UNTIL STABLE
       1. Where possible, apply UNION rule (A’s Axioms)

       2. Remove all extraneous attributes:
          a. Test if B extraneous in A → BC              (B
          extraneous if (A → B) ∈
             (F – {A → BC} U {A → C})⁺) = F₂⁺

          b. Test if B extraneous in AB → C
             (B extraneous if (A → C) ∈ F⁺)
  END
```

Canonical Cover Algorithm

Example: *Determine the canonical cover of*

$F = \{A \rightarrow BC,\ B \rightarrow CE,\ A \rightarrow E\}$

Iteration 1:

a. $F = \{A \rightarrow BCE,\ B \rightarrow CE\}$

b. *Must check for up to 5 extraneous attributes*

- B *extraneous in* $A \rightarrow BCE$? *No*
- C *extraneous in* $A \rightarrow BCE$?

  Yes: $(A \rightarrow C) \in \{A \rightarrow BE,\ B \rightarrow CE\}^+$

  1. $A \rightarrow BE$ Given
  2. $A \rightarrow B$ Decomposition (1)
  3. $B \rightarrow CE$ Given
  4. $B \rightarrow C$ Decomposition (3)
  5. $A \rightarrow C$ Trans (2,4)
- E *extraneous in* $B \rightarrow CE$? ...

Canonical Cover Algorithm

Example (cont.): $F = \{A \rightarrow BC,\ B \rightarrow CE,\ A \rightarrow E\}$

Iteration 1:

a. $F = \{A \rightarrow BCE,\ B \rightarrow CE\}$

b. *Extraneous* atts:

- B *extraneous in* $A \rightarrow BCE$? *No*
- C *extraneous in* $A \rightarrow BCE$? *Yes*...
- E *extraneous in* $A \rightarrow BCE$?

  Yes: $(A \rightarrow E) \in \{A \rightarrow B,\ B \rightarrow CE\}^+$

  1. $A \rightarrow B$ Given
  2. $B \rightarrow CE$ Given
  3. $B \rightarrow E$ Decomposition (2)
  4. $A \rightarrow E$ Trans (1,3)
- E *extraneous in* $B \rightarrow CE$? *No*
- C *extraneous in* $B \rightarrow CE$? *No*

Canonical Cover Algorithm

Example (cont.): F = {A → BC, B → CE, A → E}

Iteration 1:

a. F = {A → BCE, B → CE}

b. *Extraneous* atts:

- B *extraneous in* A → BCE? *No*
- C *extraneous in* A → BCE? *Yes*...
- E *extraneous in* A → BE? *Yes*...
- E *extraneous in* B → CE? *No*
- C *extraneous in* B → CE? *No*

Iteration 2:

a. F = {A → B, B → CE}

b. *Extraneous* atts:

- E *extraneous in* B → CE? *No*
- C *extraneous in* B → CE? *No*

DONE!

Functional Dependencies So Far…

1. Canonical Cover Algorithm

   Result (F_c) *guaranteed to be minimal FD set equivalent to* F

2. Closure Algorithms

   a. *Armstrong's Axioms:*
      More common use: test for extraneous atts in CC algorithm

   b. *Attribute closure:*
      More common use: test if set of atts is a super key

3. Purpose

   Minimize cost of global integrity constraints

   So far: min gic's = $|F_c|$

Functional Dependencies

So Far, have used for:

1. *Determining global integrity constraints*
2. *Minimizing global integrity constraints (canonical cover)*
3. *Deciding if some attribute set is a key (attribute closure)*

Next: *Influencing schema design (normalization)*