



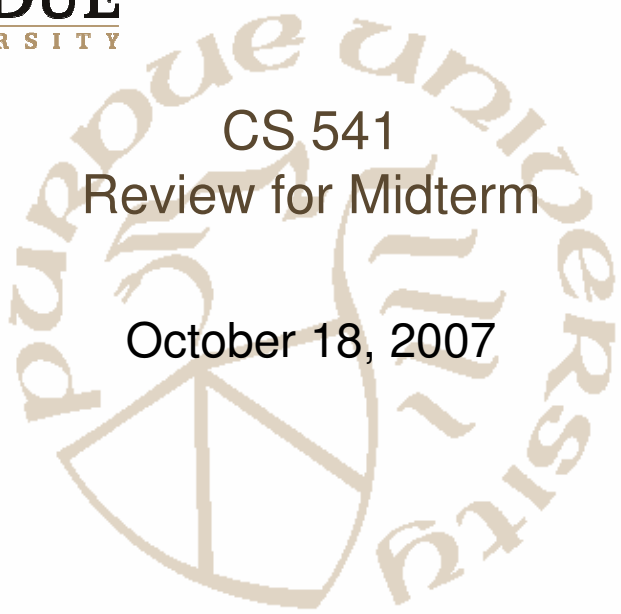
PURDUE
UNIVERSITY

CS 541
Review for Midterm

October 18, 2007

Fall 2007

Chris Clifton - CS541



Course Outline

1. Course Introduction
 - Intro / history lesson
 - Relational Model
2. Data Modeling
 - Entity-Relationship Data Model
 - Constraints and Constraint Modeling
3. Relational Theory
 - Relational Algebra and Calculus
 - Keys and Dependencies
 - Normalization
4. Using a Relational Database
 - Views
 - Constraints
 - Triggers
5. Storage mechanisms
6. Putting the Data on Disk
7. Indexing
8. Hashing / Bitmap Indexes
9. Query Processing
10. Query Optimization
11. Handling Failure
12. Concurrency Control
13. Transaction Management
14. Research topics
15. Review

Fall 2007

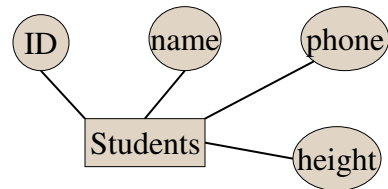
Chris Clifton - CS541

2

Entity/Relationship Model

Diagrams to represent designs.

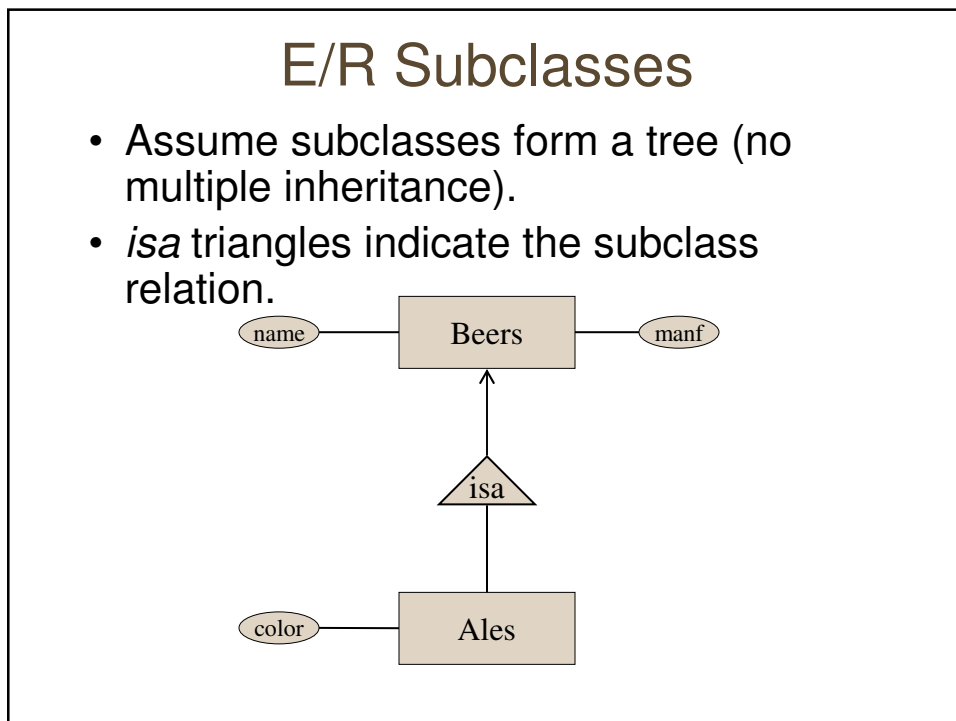
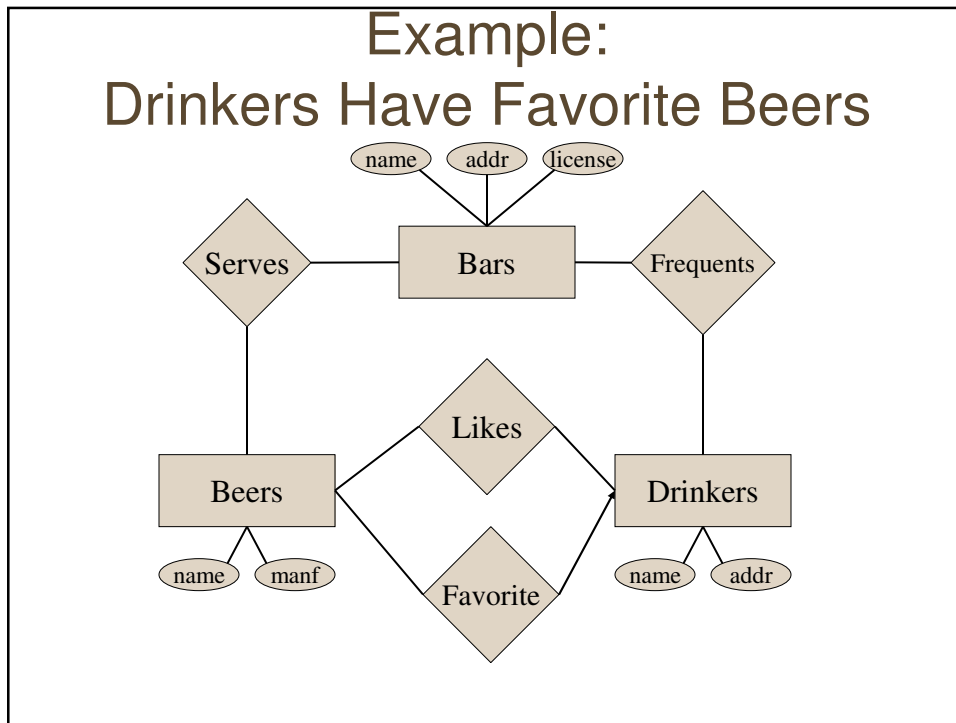
- *Entity* like object, = “thing.”
- *Entity set* like class = set of “similar” entities/objects.
- *Attribute* = property of entities in an entity set, similar to fields of a struct.
- In diagrams, entity set → rectangle; attribute → oval.

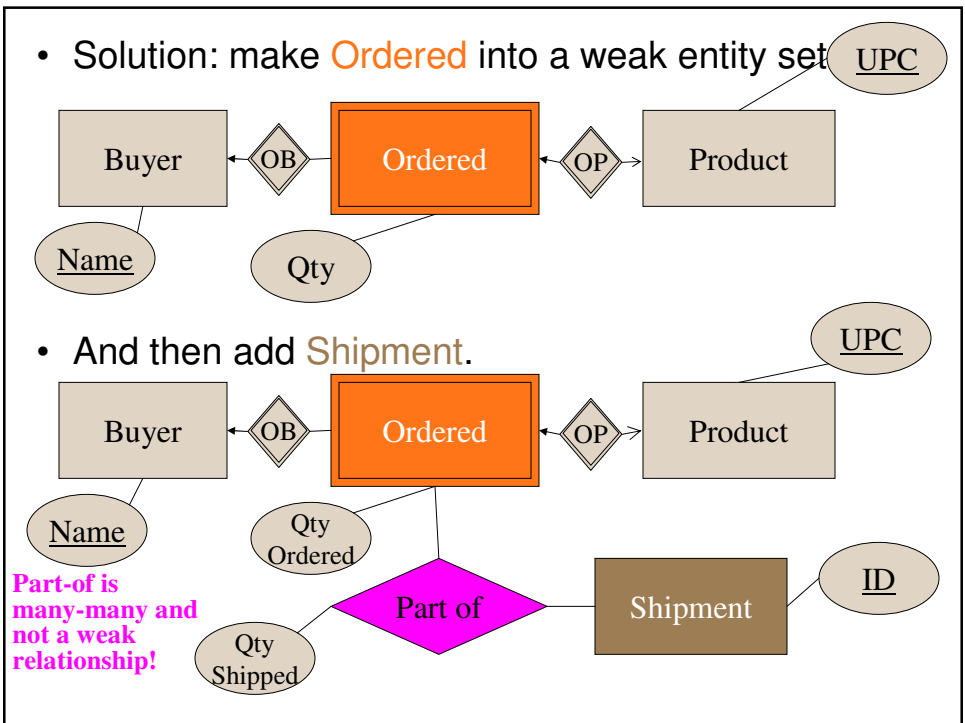
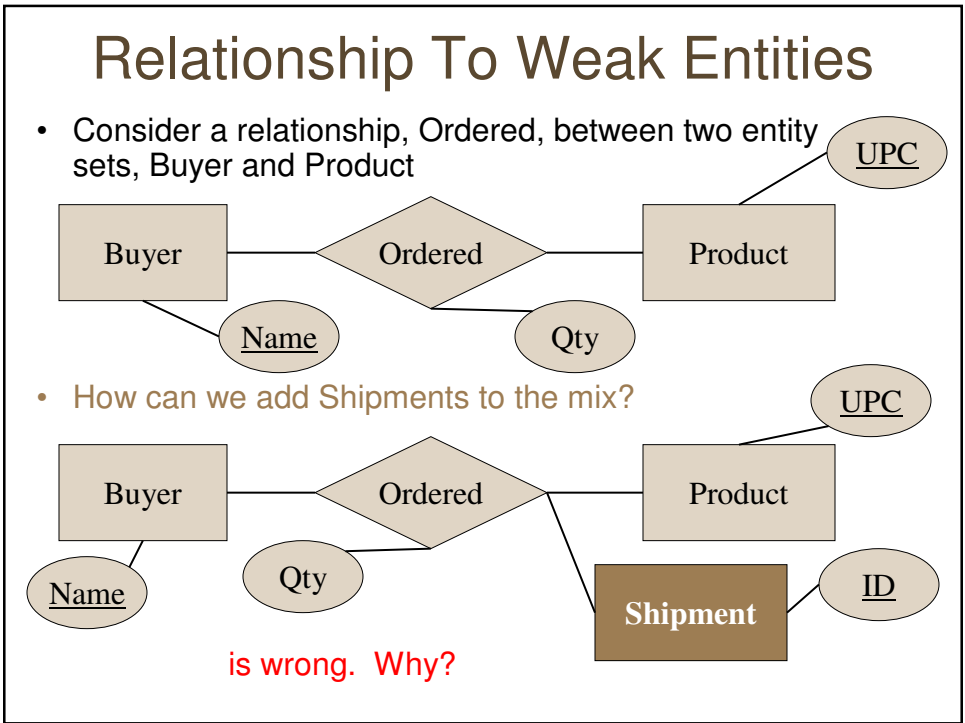


Relationships

- Connect two or more entity sets.
- Represented by diamonds.







Design Principles

Setting: client has (possibly vague) idea of what he/she wants. You must design a database that represents these thoughts and only these thoughts.

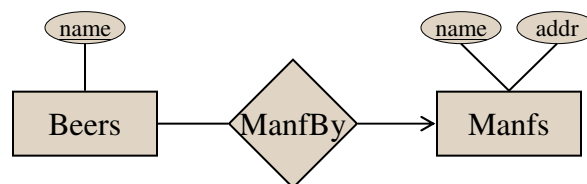
Avoid redundancy

= saying the same thing more than once.

- Wastes space and encourages inconsistency.

Example

Good:



Use Schema to Enforce Constraints

- The design *schema* should enforce as many constraints as possible.
 - Don't rely on future data to follow assumptions.

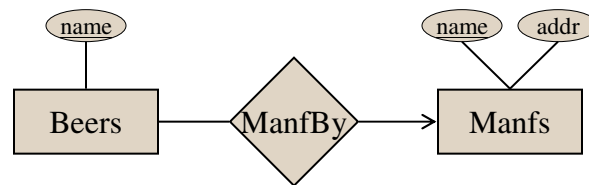
Example

- If registrar wants to associate only one instructor with a course, don't allow sets of instructors and count on departments to enter only one instructor per course.

Intuitive Rule for E.S. Vs. Attribute

Make an entity set only if it either:

1. Is more than a name of something; *i.e.*, it has nonkey attributes or relationships with a number of different entity sets, or
 - *Manfs* deserves to be an E.S. because we record *addr*, a nonkey attribute.
2. Is the “many” in a many-one relationship The following design illustrates both points:
 - *Beers* deserves to be an E.S. because it is at the “many” end.



Don't Overuse Weak E.S.

- There is a tendency to feel that no E.S. has its entities uniquely determined without following some relationships.
- However, in practice, we almost always create unique ID's to compensate: social-security numbers, VIN's, etc.
- The only times weak E.S.'s seem necessary are when:
 - a) We can't easily create such ID's; *e.g.*, no one is going to accept a “species ID” as part of the standard nomenclature (species is a weak E.S. supported by membership in a genus).
 - b) There is no global authority to create them, *e.g.*, crews and studios.

Relational Model

- Table = *relation*.
- Column headers = *attributes*.
- Row = *tuple*

name	manf
WinterBrew	Pete's
BudLite	A.B.
...	...

Beers

- *Relation schema* = name(attributes) + other structure info., e.g., keys, other constraints. Example: Beers (name , manf)
 - Order of attributes is arbitrary, but in practice we need to assume the order given in the relation schema.
- *Relation instance* is current set of rows for a relation schema.
- *Database schema* = collection of relation schemas.

Relational Data Model

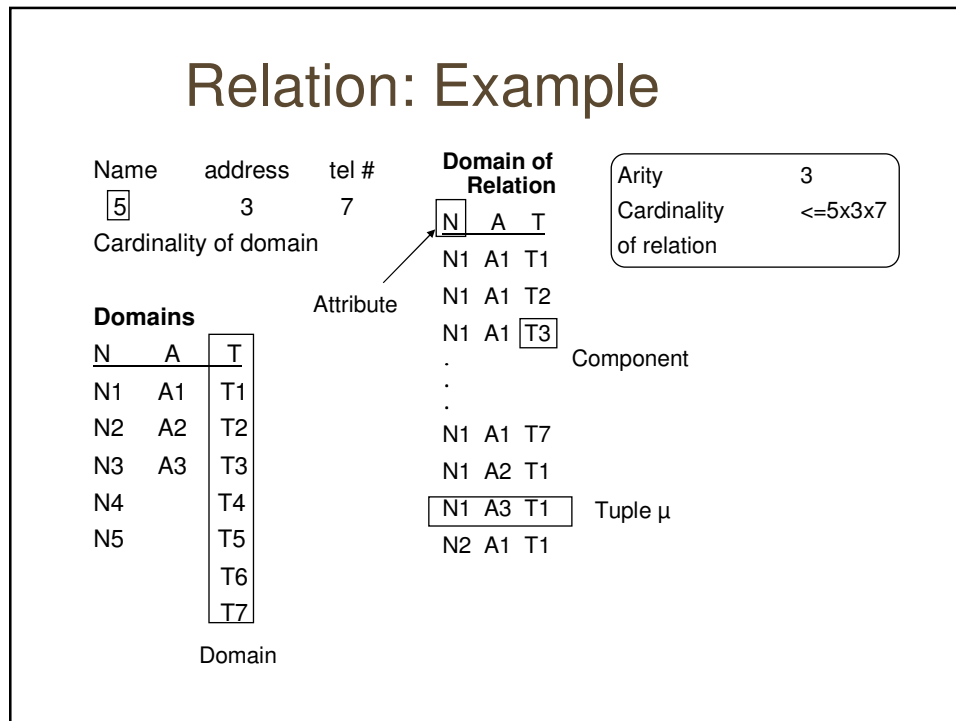
Relation as table

- Rows = tuples
- Columns = components
- Names of columns = attributes
- Set of attribute names = schema
- REL (A1,A2,...,An)

Set theoretic

- Domain — set of values like a data type
- Cartesian product (or product) $D1 \times D2 \times \dots \times Dn$
- n-tuples $(V1, V2, \dots, Vn)$
- s.t., $V1 \in D1, V2 \in D2, \dots, Vn \in Dn$
- Relation-subset of cartesian product of one or more domains
- FINITE only; empty set allowed
- Tuples = members of a relation inst.
- Arity = number of domains
- Components = values in a tuple
- Domains — corresp. with attributes
- Cardinality = number of tuples

C	↑	A1	A2	A3	...	An	← Attributes
a	↓	a1	a2	a3	...	an	← Tuple
b	↓	b1	b2	a3	...	cn	
a	↓	a1	c3	b3	...	bn	← Component
.	↓	
x	↓	x1	v2	d3	...	wn	
y	↓	← Arity					



About Relational Model

Order of tuples not important
Order of attributes not important (in theory)

Collection of relation schemas (intension)
Relational database schema

Corresponding relation instances (extension)
Relational database

intension vs. extension
schema vs. data

metadata
includes schema

Why Relations?

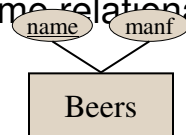
- Very simple model.
- *Often* a good match for the way we think about our data.
- Abstract model that underlies SQL, the most important language in DBMS's today.
 - But SQL uses “bags” while the abstract relational model is set-oriented.

Relational Design

Simplest approach (not always best): convert each E.S. to a relation and each relationship to a relation.

Entity Set → Relation

E.S. attributes become relational attributes.



Becomes:

`Beers(name, manf)`

Keys in Relations

An attribute or set of attributes K is a *key* for a relation R if we expect that in no instance of R will two different tuples agree on all the attributes of K .

- Indicate a key by underlining the key attributes.
- Example: If `name` is a key for `Beers`:
`Beers(name, manf)`

E/R Relationships → Relations

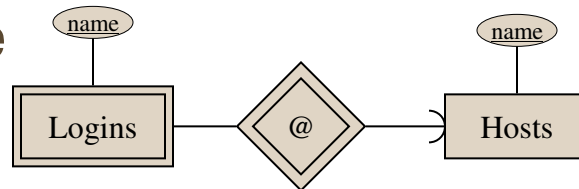
Relation has attribute for *key* attributes of each E.S. that participates in the relationship.

- Add any attributes that belong to the relationship itself.
- Renaming attributes OK.
 - Essential if multiple roles for an E.S.

Weak Entity Sets, Relationships → Relations

- Relation for a weak E.S. must include its full key (*i.e.*, attributes of related entity sets) as well as its own attributes.
- A supporting (double-diamond) relationship yields a relation that is actually redundant and should be deleted from the database schema.

Example



Hosts(hostName)

Logins(loginName, hostName)

At(loginName, hostName, hostName2)

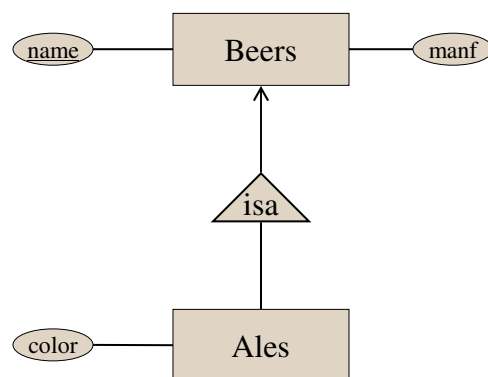
- In At, hostName and hostName2 must be the same host, so delete one of them.
- Then, Logins and At become the same relation; delete one of them.
- In this case, Hosts' schema is a subset of Logins' schema. Delete Hosts?

Subclasses → Relations

Three approaches:

1. Object-oriented: each entity is in one class. Create a relation for each class, with all the attributes for that class.
 - Don't forget inherited attributes.
2. E/R style: an entity is in a network of classes related by *isa*. Create one relation for each E.S.
 - An entity is represented in the relation for each subclass to which it belongs.
 - Relation has only the attributes attached to that E.S. + key.
3. Use nulls. Create one relation for the root class or root E.S., with all attributes found anywhere in its network of subclasses.
 - Put `NULL` in attributes not relevant to a given entity.

Example



Keys of Relations

K is a *key* for relation R if:

1. $K \rightarrow$ all attributes of R . (**Uniqueness**)
2. For no proper subset of K is (1) true.
(Minimality)
 - If K at least satisfies (1), then K is a *superkey*.

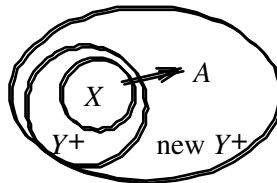
Conventions

- Pick one key; underline key attributes in the relation schema.
- X , etc., represent sets of attributes; A etc., represent single attributes.
- No set formers in FD's, e.g., ABC instead of $\{A, B, C\}$.

Inferring FDs

Define $Y^+ =$ *closure* of $Y =$ set of attributes functionally determined by Y :

- Basis: $Y^+ := Y$.
- Induction: If $X \subseteq Y^+$, and $X \rightarrow A$ is a given FD, then add A to Y^+ .



- End when Y^+ cannot be changed.

Algorithm

- For each set of attributes X compute X^+ .
 - But skip $X = \emptyset$, $X =$ all attributes.
 - Add $X \rightarrow A$ for each A in $X^+ - X$.
- Drop $XY \rightarrow A$ if $X \rightarrow A$ holds.
 - Consequence: If X^+ is all attributes, then there is no point in computing closure of supersets of X .
- Finally, project the FD's by selecting only those FD's that involve only the attributes of the projection.
 - Notice that after we project the discovered FD's onto some relation, the eliminated FD's can be inferred *in the projected relation*.

FDs: Armstrong's Axioms

- Reflexivity:
 - If $\{B_1, B_2, \dots, B_m\} \subseteq \{A_1, A_2, \dots, A_n\} \Rightarrow$
 $A_1A_2 \dots A_n \rightarrow B_1B_2 \dots B_m$
 - Also called “trivial FDs”
- Augmentation:
 - $A_1A_2 \dots A_n \rightarrow B_1B_2 \dots B_m \Rightarrow$
 $A_1A_2 \dots A_nC_1C_2 \dots C_k \rightarrow B_1B_2 \dots B_mC_1C_2 \dots C_k$
- Transitivity:
 - $A_1A_2 \dots A_n \rightarrow B_1B_2 \dots B_m$ and $B_1B_2 \dots B_m \rightarrow$
 $C_1C_2 \dots C_k \Rightarrow A_1A_2 \dots A_n \rightarrow C_1C_2 \dots C_k$

Example – Functional Dependencies

In ABC with FD's $A \rightarrow B$, $B \rightarrow C$, project onto AC .

1. $A^+ = ABC$; yields $A \rightarrow B$, $A \rightarrow C$.
2. $B^+ = BC$; yields $B \rightarrow C$.
3. $AB^+ = ABC$; yields $AB \rightarrow C$; drop in favor of $A \rightarrow C$.
4. $AC^+ = ABC$ yields $AC \rightarrow B$; drop in favor of $A \rightarrow B$.
5. $C^+ = C$ and $BC^+ = BC$; adds nothing.
 - Resulting FD's: $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow C$.
 - Projection onto AC : $A \rightarrow C$.

Normalization

Goal = BCNF = Boyce-Codd Normal Form =
all FD's follow from the fact "key \rightarrow
everything."

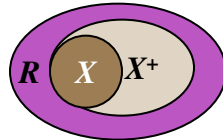
- Formally, R is in BCNF if for every nontrivial FD for R , say $X \rightarrow A$, then X is a superkey.
 - "Nontrivial" = right-side attribute not in left side.

Why?

1. Guarantees no redundancy due to FD's.
2. Guarantees no *update anomalies* = one occurrence of a fact is updated, not all.
3. Guarantees no *deletion anomalies* = valid fact is lost when tuple is deleted.

Decomposition to Reach BCNF

- 1. Compute X_+ .
 - Cannot be all attributes – why?
- 2. Decompose R into X_+ and $(R - X_+) \cup X$.



- 3. Find the FD's for the decomposed relations.
 - Project the FD's from F = calculate all consequents of F that involve only attributes from X_+ or only from $(R - X_+) \cup X$.

3NF

One FD structure causes problems:

- If you decompose, you can't check all the FD's only in the decomposed relations.
- If you don't decompose, you violate BCNF.

Abstractly: $AB \rightarrow C$ and $C \rightarrow B$.

- Example 1: title city \rightarrow theatre and theatre \rightarrow city.
- Example 2: street city \rightarrow zip, zip \rightarrow city.

Keys: $\{A, B\}$ and $\{A, C\}$, but $C \rightarrow B$ has a left side that is not a superkey.

- Suggests decomposition into BC and AC .
 - But you can't check the FD $AB \rightarrow C$ in only these relations.

“Elegant” Workaround

Define the problem away.

- A relation R is in 3NF iff (if and only if) for every nontrivial FD $X \rightarrow A$, either:
 1. X is a superkey, or
 2. A is *prime* = member of at least one key.
- Thus, the canonical problem goes away: you don't have to decompose because all attributes are prime.

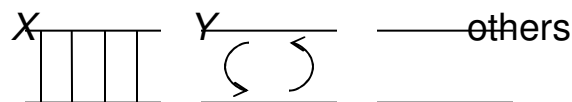
What 3NF Gives You

There are two important properties of a decomposition:

1. We should be able to recover from the decomposed relations the data of the original.
 - Recovery involves projection and join, which we shall defer until we've discussed relational algebra.
2. We should be able to check that the FD's for the original relation are satisfied by checking the projections of those FD's in the decomposed relations.
 - Without proof, we assert that it is always possible to decompose into BCNF and satisfy (1).
 - Also without proof, we can decompose into 3NF and satisfy both (1) and (2).
 - But it is not possible to decompose into BCNF and get both (1) and (2).
 - Street-city-zip is an example of this point.

Multivalued Dependencies

The *multivalued dependency* $X \twoheadrightarrow Y$ holds in a relation R if whenever we have two tuples of R that agree in all the attributes of X , then we can swap their Y components and get two new tuples that are also in R .



MVD Rules

1. Every FD is an MVD.

- Because if $X \rightarrow Y$, then swapping Y 's between tuples that agree on X doesn't create new tuples.
- Example, in `Drinkers`: `name` \twoheadrightarrow `addr`.

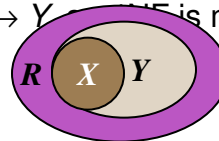
2. Complementation: if $X \twoheadrightarrow Y$, then $X \twoheadrightarrow Z$, where Z is all attributes not in X or Y .

- Example: since `name` \twoheadrightarrow `phones` holds in `Drinkers`, so does `name` \twoheadrightarrow `addr beersLiked`.

4NF

Eliminate redundancy due to multiplicative effect of MVD's.

- Roughly: treat MVD's as FD's for decomposition, but not for finding keys.
- Formally: R is in Fourth Normal Form if whenever MVD $X \twoheadrightarrow Y$ is *nontrivial* (Y is not a subset of X , and $X \cup Y$ is not all attributes), then X is a superkey.
 - Remember, $X \rightarrow Y$ implies $X \twoheadrightarrow Y$. 4NF is more stringent than BCNF.
- Decompose R , using 4NF violation $X \twoheadrightarrow Y$, into XY and $X \cup (R - Y)$.



Why Decomposition “Works”?

What does it mean to “work”? Why can't we just tear sets of attributes apart as we like?

- Answer: the decomposed relations need to represent the same information as the original.
 - We must be able to reconstruct the original from the decomposed relations.

Projection and Join Connect the Original and Decomposed Relations

- Suppose R is decomposed into S and T . We project R onto S and onto T .

Example

$$R =$$

<u>name</u>	<u>addr</u>	<u>beersLiked</u>	<u>manf</u>	<u>favoriteBeer</u>
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	Voyager	WickedAle	Pete's	WickedAle
Spock	Enterprise	Bud	A.B.	Bud

- FDs:
 - name addr
 - name favoriteBeer
 - beersLiked manf
- Decompose:

Project onto Drinkers1 (name, addr, favoriteBeer):

<u>name</u>	<u>addr</u>	<u>favoriteBeer</u>
Janeway	Voyager	WickedAle
Spock	Enterprise	Bud

Project onto Drinkers3 (beersLiked, manf):

<u>beersLiked</u>	<u>manf</u>
Bud	A.B.
WickedAle	Pete's
Bud	A.B.

Project onto Drinkers4 (name, beersLiked):

<u>name</u>	<u>beersLiked</u>
Janeway	Bud
Janeway	WickedAle
Spock	Bud

Reconstruction of Original

Can we figure out the original relation from the decomposed relations?

- Sometimes, if we natural join the relations.

Example

Drinkers3 \bowtie Drinkers4 =

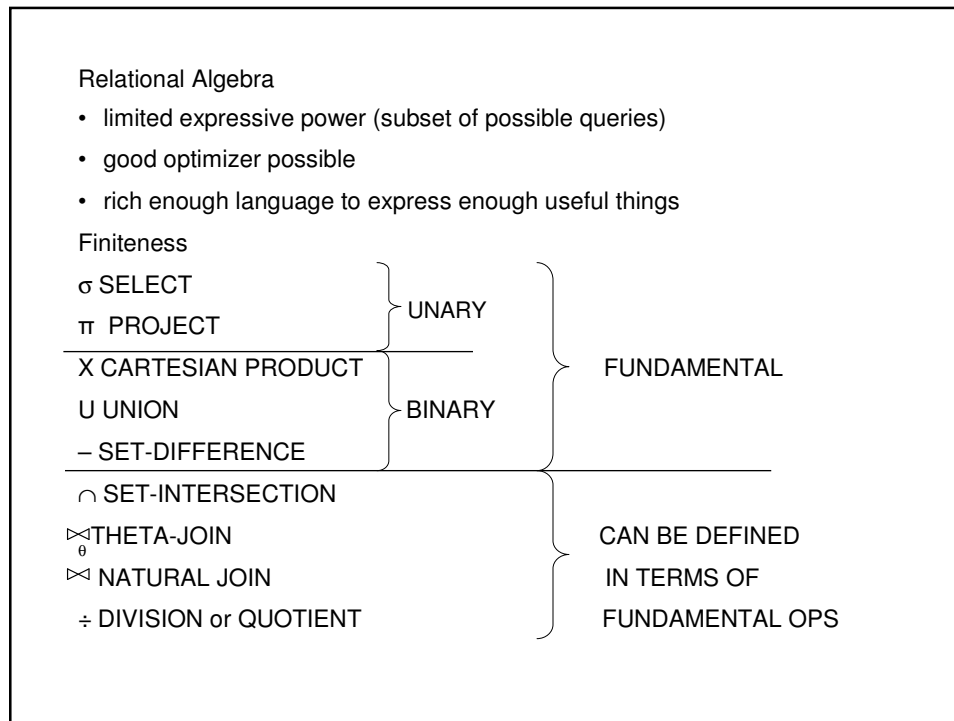
name	beersLiked	manf
Janeway	Bud	A.B.
Janeway	WickedAle	Pete's
Spock	Bud	A.B.

- Join of above with Drinkers1 = original *R*.

Theorem

Suppose we decompose a relation with schema *XYZ* into *XY* and *XZ* and project the relation for *XYZ* onto *XY* and *XZ*. Then $XY \bowtie XZ$ is *guaranteed* to reconstruct *XYZ* if and only if $X \twoheadrightarrow Y$ (or equivalently, $X \twoheadrightarrow Z$).

- Usually, the MVD is really a FD, $X \rightarrow Y$ or $X \rightarrow Z$.
- BCNF: When we decompose *XYZ* into *XY* and *XZ*, it is because there is a FD $X \rightarrow Y$ or $X \rightarrow Z$ that violates BCNF.
 - Thus, we can always reconstruct *XYZ* from its projections onto *XY* and *XZ*.
- 4NF: when we decompose *XYZ* into *XY* and *XZ*, it is because there is an MVD $X \twoheadrightarrow Y$ or $X \twoheadrightarrow Z$ that violates 4NF.
 - Again, we can reconstruct *XYZ* from its projections onto *XY* and *XZ*.



Bag Semantics

A relation (in SQL, at least) is really a *bag* or *multiset*.

- It may contain the same tuple more than once, although there is no specified order (unlike a list).
- Example: $\{1,2,1,3\}$ is a bag and not a set.
- Select, project, and join work for bags as well as sets.
 - Just work on a tuple-by-tuple basis, and don't eliminate duplicates.

Laws for Bags Differ From Laws for Sets

- Some familiar laws continue to hold for bags.
 - Examples: union and intersection are still commutative and associative.
- But other laws that hold for sets do *not* hold for bags.

Example

$R \cap (S \cup T) \equiv (R \cap S) \cup (R \cap T)$ holds for sets.

- Let R , S , and T each be the bag $\{1\}$.
- Left side: $S \cup T = \{1,1\}$; $R \cap (S \cup T) = \{1\}$.
- Right side: $R \cap S = R \cap T = \{1\}$;
 $(R \cap S) \cup (R \cap T) = \{1\} \cup \{1\} = \{1,1\} \neq \{1\}$.

Extended (“Nonclassical”) Relational Algebra

Adds features needed for SQL, bags.

1. Duplicate-elimination operator δ .
2. Extended projection.
3. Sorting operator τ .
4. Grouping-and-aggregation operator γ .
5. Outerjoin operator \bowtie^o .

SQL

- DML
 - select, from, where, renaming
 - set operations
 - ordering
 - aggregate functions
 - nested subqueries
- other parts: DDL, embedded SQL, auth etc

DML

General form

select a1, a2, ... an

from r1, r2, ... rm

where P

[**order by**]

[**group by** ...]

[**having** ...]

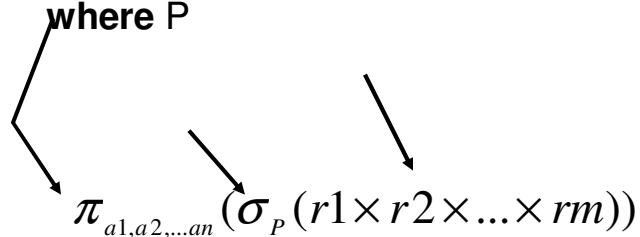
DML - observation

General form

select distinct a1, a2, ... an

from r1, r2, ... rm

where P



Aggregate functions- 'having'

find students with GPA > 3.0

select ssn, **avg**(grade)

from takes

group by ssn

having **avg**(grade)>3.0

SSN	c-id	grade
123	15-413	4
234	15-413	3

'having' <-> 'where' for groups

SSN	avg(grade)
123	4
234	3

DML - nested subqueries

find names of students of 15-415

```
select name
from student
where ssn in (
  select ssn
  from takes
  where c-id = "15-415")
```

Constraints

Commercial relational systems allow much more “fine-tuning” of constraints than do the modeling languages we learned earlier.

- In essence: SQL programming is used to describe constraints.

Outline

1. Primary key declarations.
2. Foreign-keys = referential integrity constraints.
3. Attribute- and tuple-based checks = constraints within relations.
4. SQL Assertions = global constraints.
 - Not found in Oracle.
5. Oracle Triggers.
 - A substitute for assertions.

Triggers (Oracle Version)

Often called event-condition-action rules.

- *Event* = a class of changes in the DB, e.g., “insertions into Beers.”
- *Condition* = a test as in a where-clause for whether or not the trigger applies.
- *Action* = one or more SQL statements.
- Differ from checks or SQL assertions in that:
 1. Triggers invoked by the event; the system doesn't have to figure out when a trigger could be violated.
 2. Condition not available in checks.

- Triggers are part of the database schema, like tables or views.
- Important Oracle constraint: the action cannot change the relation that triggers the action.
 - Worse, the action cannot even change a relation connected to the triggering relation by a constraint, e.g., a foreign-key constraint.

Views

An expression that describes a table without creating it.

- View definition form is:
`CREATE VIEW <name> AS <query>;`

Semantics of View Use

SQL query \longrightarrow rel. algebra \longrightarrow SQL

SQL view def. \longrightarrow rel. algebra \longrightarrow SQL

Example

$\pi_{drinker, beer}$
 \bowtie
 Frequents Sells
 CanDrink

π_{beer}
 $\sigma_{drinker \neq Sally'}$
 CanDrink
 Query

Modification to Views Via Triggers

Oracle allows us to “intercept” a modification to a view through an instead-of trigger.

Example

```
Likes(drinker, beer)
Sells(bar, beer, price)
Frequents(drinker, bar)
```

```
CREATE VIEW Synergy AS
  SELECT Likes.drinker, Likes.beer,
         Sells.bar
  FROM Likes, Sells, Frequents
  WHERE Likes.drinker = Frequents.drinker AND
         Likes.beer = Sells.beer AND
         Sells.bar = Frequents.bar;
```

Cursors

Declare by:

```
CURSOR <name> IS
  select-from-where statement
```

- Cursor gets each tuple from the relation produced by the select-from-where, in turn, using a *fetch statement* in a loop.
 - Fetch statement:


```
FETCH <cursor name> INTO
  variable list;
```
- Break the loop by a statement of the form:


```
EXIT WHEN <cursor name> %NOTFOUND;
```

 - True when there are no more tuples to get.
- Open and close the cursor with OPEN and CLOSE.

Authorization in SQL

- File systems identify certain access privileges on files, *e.g.*, read, write, execute.
- In partial analogy, SQL identifies six access privileges on relations, of which the most important are:
 1. `SELECT` = the right to query the relation.
 2. `INSERT` = the right to insert tuples into the relation
 - may refer to one attribute, in which case the privilege is to specify only one column of the inserted tuple.
 3. `DELETE` = the right to delete tuples from the relation.
 4. `UPDATE` = the right to update tuples of the relation
 - may refer to one attribute.

Granting Privileges

- You have all possible privileges to the relations you create.
- You may grant privileges to any user if you have those privileges “with grant option.”
 - You have this option to your own relations.

Example

1. Here, Sally can query `Sells` and can change prices, but cannot pass on this power:

```
GRANT SELECT ON Sells,
      UPDATE(price) ON Sells
TO sally;
```

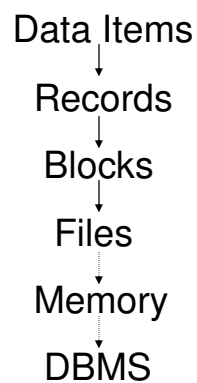
2. Here, Sally can also pass these privileges to whom she chooses:

```
GRANT SELECT ON Sells,
      UPDATE(price) ON Sells
TO sally
WITH GRANT OPTION;
```

Storage

- Secondary storage, mainly disks
- I/O times
 - Time = Seek Time +
Rotational Delay +
Transfer Time +
Other
- I/Os should be avoided,
especially random ones.....

- How to lay out data on disk

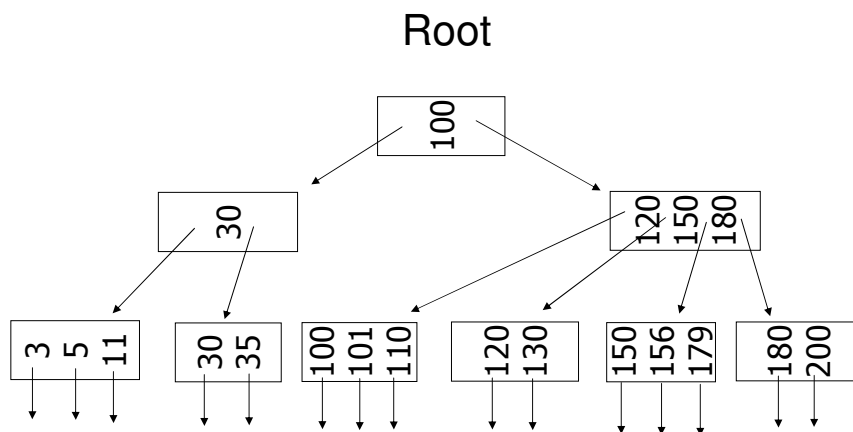


Indexing

- Index sequential file
- Search key (\neq primary key)
- Primary index (on Sequencing field)
- Secondary index
- Dense index (all Search Key values in)
- Sparse index
- Multi-level index
- Insertion/deletion

B+Tree

$n=3$



B+ tree rules

- (1) All leaves at same lowest level
(balanced tree)
- (2) Pointers in leaves point to records
except for “sequence pointer”

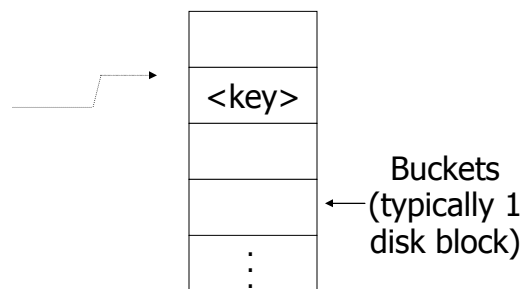
Use at least

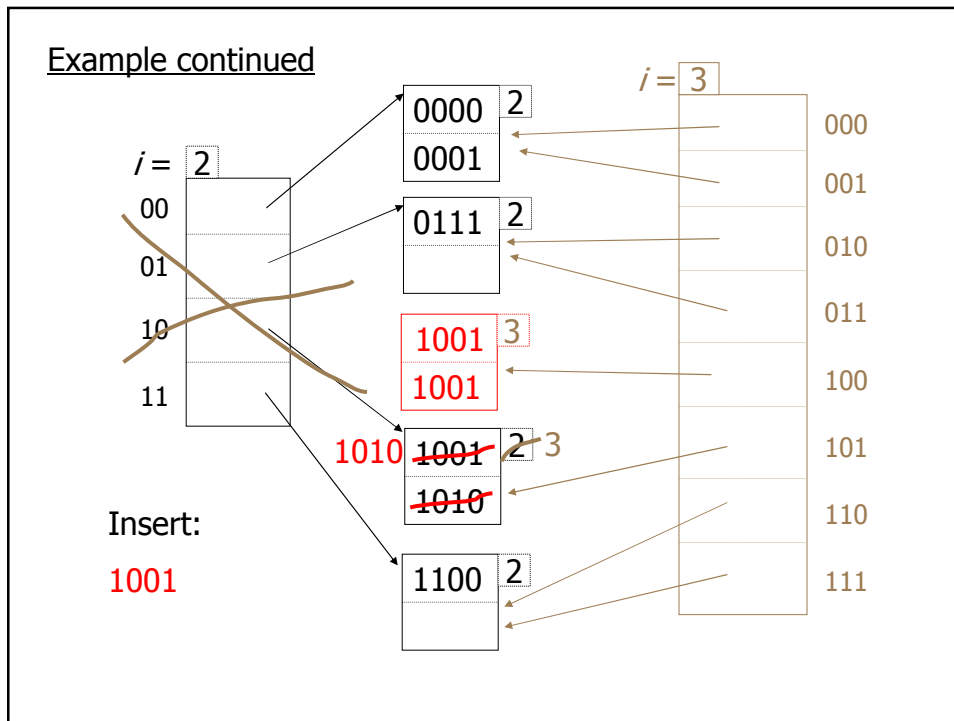
Non-leaf: $\lceil (n+1)/2 \rceil$ pointers

Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data

Hashing

key \rightarrow h(key)





Extensible hashing: deletion

- No merging of blocks
- Merge blocks
and cut directory if possible
(Reverse insert procedure)

Summary Extensible hashing

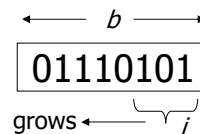
- Can handle growing files
- ⊕ - with less wasted space
- with no full reorganizations
- ⊖ Indirection
(Not bad if directory in memory)
- ⊖ Directory doubles in size
(Now it fits, now it does not)

Linear hashing

- Another dynamic hashing scheme

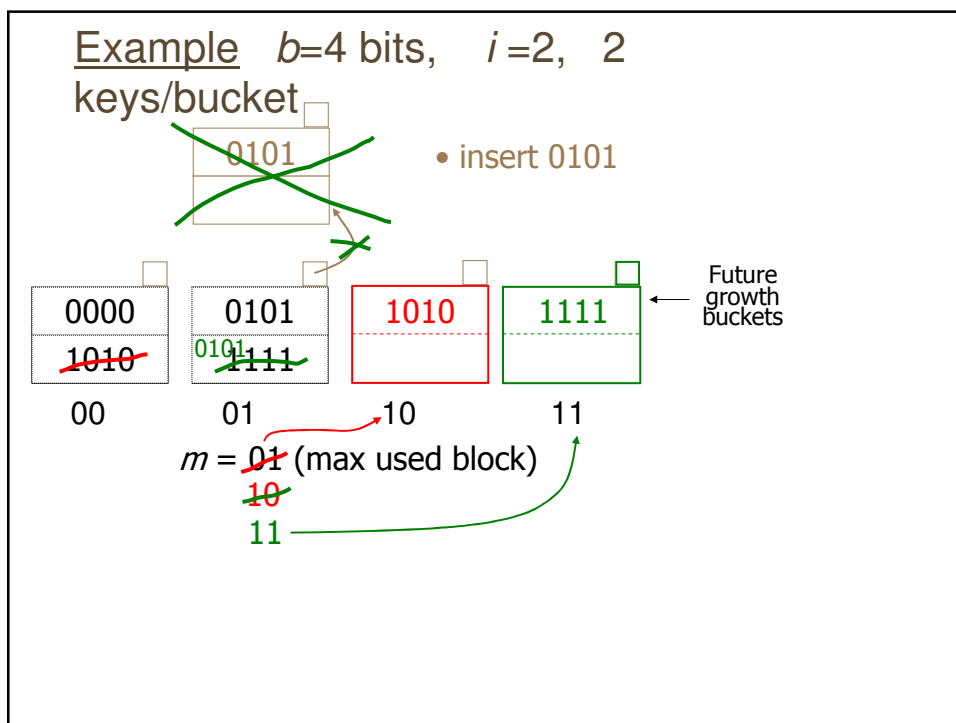
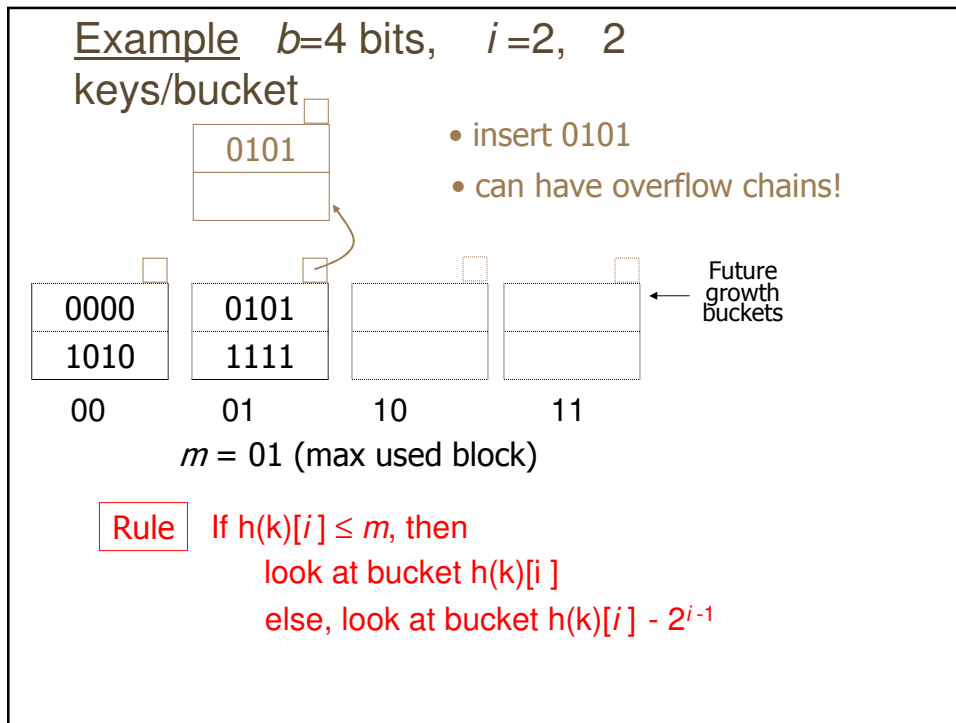
Two ideas:

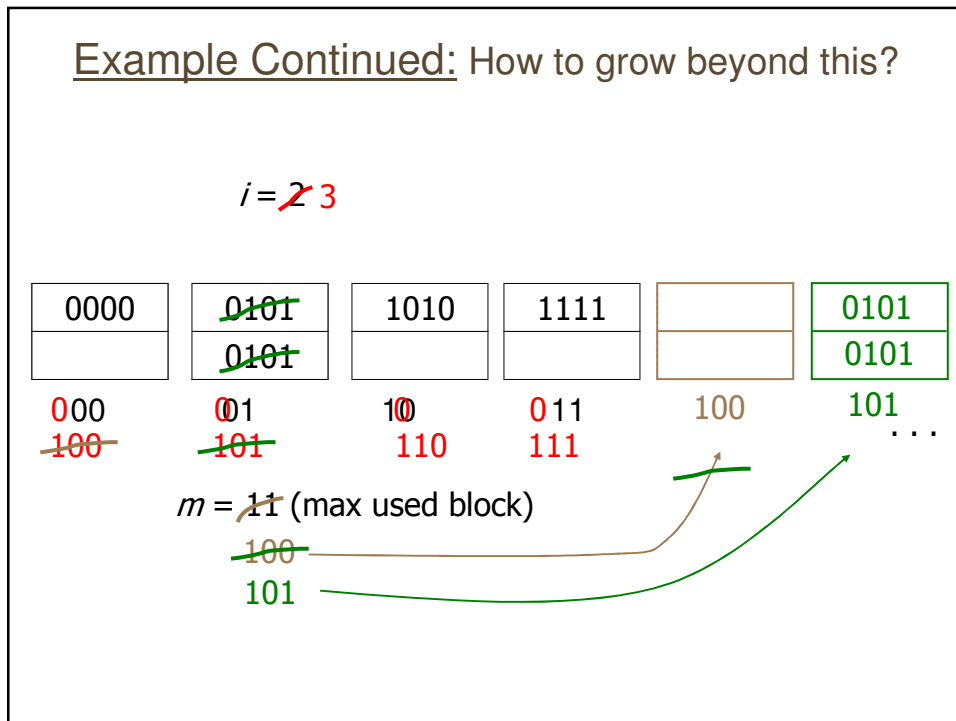
(a) Use i low order bits of hash



(b) File grows linearly







- 👉 When do we expand file?
- Keep track of:
$$\frac{\text{\# used slots}}{\text{total \# of slots}} = U$$
 - If $U > \text{threshold}$ then increase m
(and maybe i)

Summary Linear Hashing

- ⊕ Can handle growing files
 - with less wasted space
 - with no full reorganizations

- ⊕ No indirection like extensible hashing

- ⊖ Can still have overflow chains