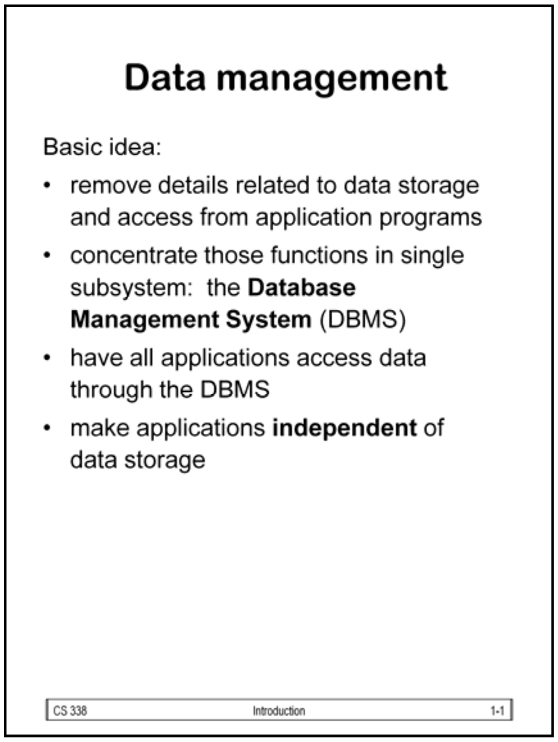


Data management

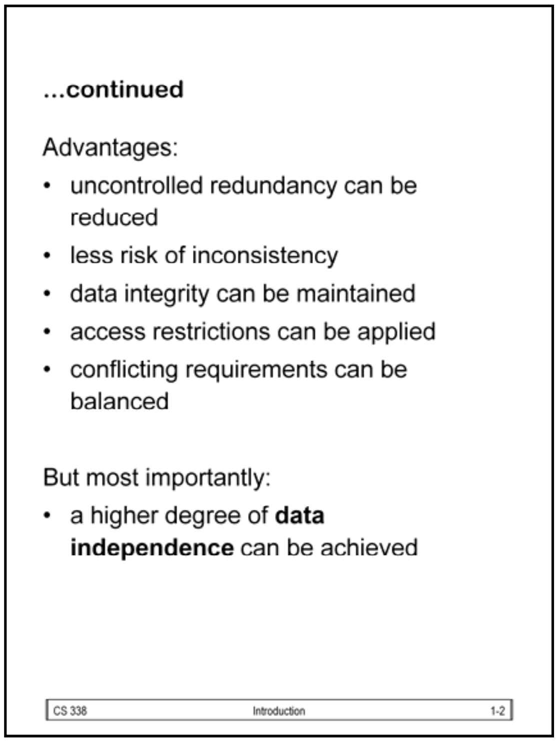
Basic idea:

- remove details related to data storage and access from application programs
- concentrate those functions in single subsystem: the **Database Management System** (DBMS)
- have all applications access data through the DBMS
- make applications **independent** of data storage

CS 338 Introduction 1-1

So as noted: remove, as much as practical, data storage and retrieval from individual applications.
Concentrate this functionality into a single program (collection of programs).
Refer to this as a DB mgmt sys.
Then, make application programs get at data through the DBMS.

Important to understand that we want a “clean break” between app and DBMS.
In current op-sys, DBMS implemented as a system service. Like to think of sending a message (eg a request for some data) and receiving a reply (eg with the data we requested)



...continued

Advantages:

- uncontrolled redundancy can be reduced
- less risk of inconsistency
- data integrity can be maintained
- access restrictions can be applied
- conflicting requirements can be balanced

But most importantly:

- a higher degree of **data independence** can be achieved

Separation facilitates some immediate advantages:

DBMS can reduce redundancy by creating “see elsewhere” notes (think of manual filing system)

Reduced redundancy means less inconsistency: Inconsistency arises when data duplication for retrieval convenience -- because data stored in a single place, no change of being inconsistent

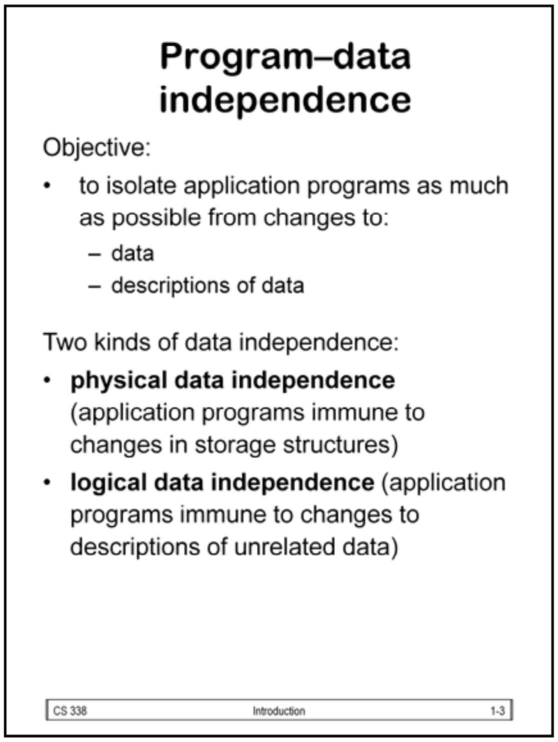
Integrity: DBMS can create backups, checkpoints, logs transparently to apps (in waiting until needed). Individual apps could do this, but centralized in DBMS means can leverage the investment into all apps.

Restrictions. DBMS is a single point of access, so can add security easily (eg user-names and passwords, encryption).

Conflicts: eg concurrency, performance. Multiple simultaneous applications through single DBMS: can handle concurrency by serializing (only allowing one app access). performance: DBMS can guarantee certain levels of response to apps, facilitate priorities.

Also (and maybe most importantly): DBMS lets us achieve data-independence: separation of users of data from definition and storage of data.

Look at this idea of data independence in more detail...



Program–data independence

Objective:

- to isolate application programs as much as possible from changes to:
  - data
  - descriptions of data

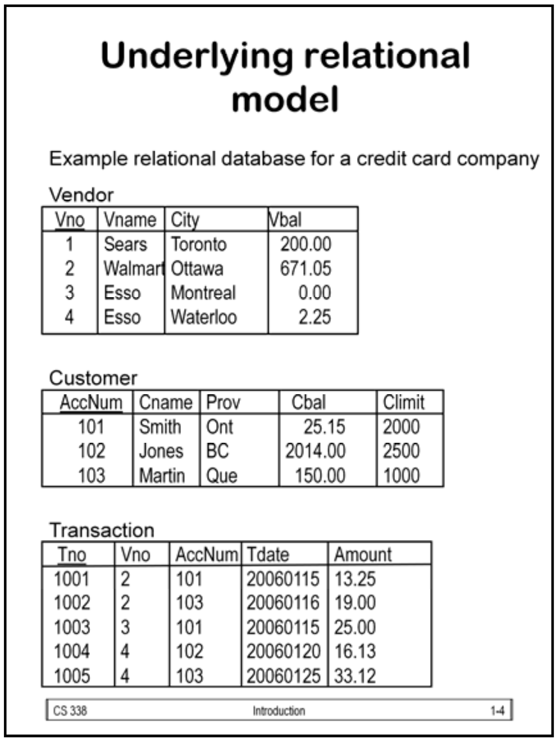
Two kinds of data independence:

- **physical data independence** (application programs immune to changes in storage structures)
- **logical data independence** (application programs immune to changes to descriptions of unrelated data)

CS 338 Introduction 1-3

Seems like separating data from application code is good, has many advantages (really just an extension of the concept of separating code and data that is preached by many programming philosophies)
But, there are different aspects of independence:
- the data itself. apps shouldn’t need to worry about how to access data (eg finding it, getting it, For a table, is it stored row-by-row or column by column? How many columns. column ordering. Disk blocking factors)
- descriptions of the data ie the format/type of the data. Eg is the data characters, numbers, bitmaps? Clearly, if the description of a given datum used by a pgm changes, the pgm must change. But, other pgms need not be aware. Eg if our db contains both payroll and inventory information, payroll applications should be completely independent changes to inventory data organization.

Formally, we identify two kinds of data independence: physical, which lets pgms be independent of how the data is stored onto its physical media; and logical, which is lets pgms ignore organization of the data.
So, a pgm that is data independent doesn't care where DBMS stores data, or how DBMS stores or accesses, or what other data/pgms DBMS is managing



Underlying relational model

Example relational database for a credit card company

Vendor

<u>Vno</u>	Vname	City	Vbal
1	Sears	Toronto	200.00
2	Walmart	Ottawa	671.05
3	Esso	Montreal	0.00
4	Esso	Waterloo	2.25

Customer

<u>AccNum</u>	Cname	Prov	Cbal	Climit
101	Smith	Ont	25.15	2000
102	Jones	BC	2014.00	2500
103	Martin	Que	150.00	1000

Transaction

<u>Tno</u>	Vno	AccNum	Tdate	Amount
1001	2	101	20060115	13.25
1002	2	103	20060116	19.00
1003	3	101	20060115	25.00
1004	4	102	20060120	16.13
1005	4	103	20060125	33.12

CS 338 Introduction 1-4

sidetrack for a moment: have mentioned the notion of "tables" as the basis for relational DB. Will clarify these ideas now.

Shown is an eg of a relational database:

- 3 tables, each has a name

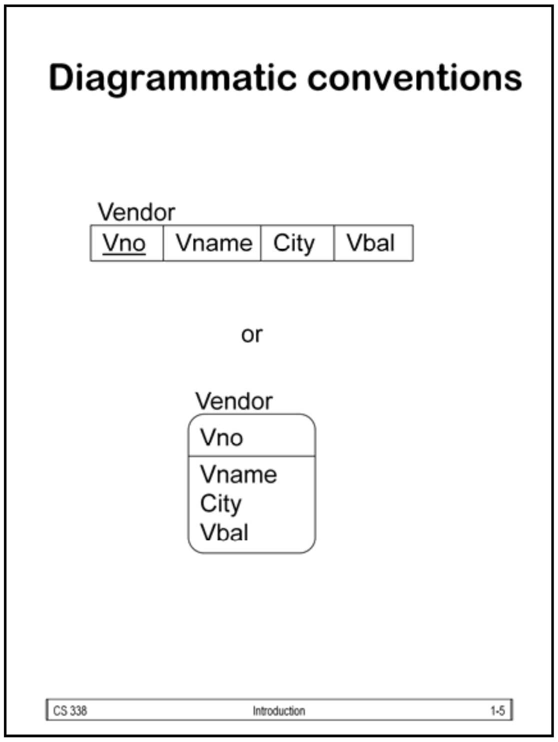
- Tables are composed of rows and columns. each column has a name. each row can be disringuished from each other, somehow (some combinations of the columns is unique). the set of columns (might be only one) is called the primary key, and its name is underlined.

The number of columns in a table is fixed; the number of rows varies.

columns from one table might be used in another table (eg vno in vendor to vno in transaction). if we assume that the "vendor" table is the defining point, would want to restrict contents of the vno column in "transaction" so that the values exist in "vendor" [this is referential integrity]

—>Observe the foreign keys

—>multi-column keys



Diagrammatic conventions

Conventional to use diagrams to describe databases. Describes the schema, not the instance

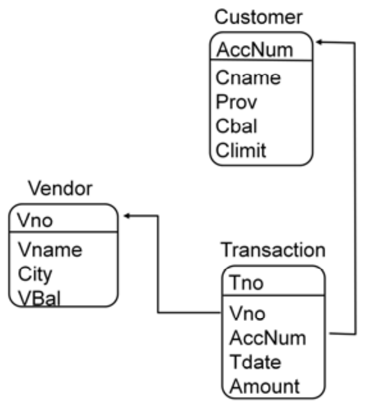
Preceding table diagrams are handy:
table name, attribute names, underline the attribute(s) (one or more) that is guaranteed to be unique for the tuples.

another style of diagram that often is used to show the dependencies between attributes in relation (the connections between columns in tables). The primary key (the collection of unique attributes) is shown above the line; remaining are below.

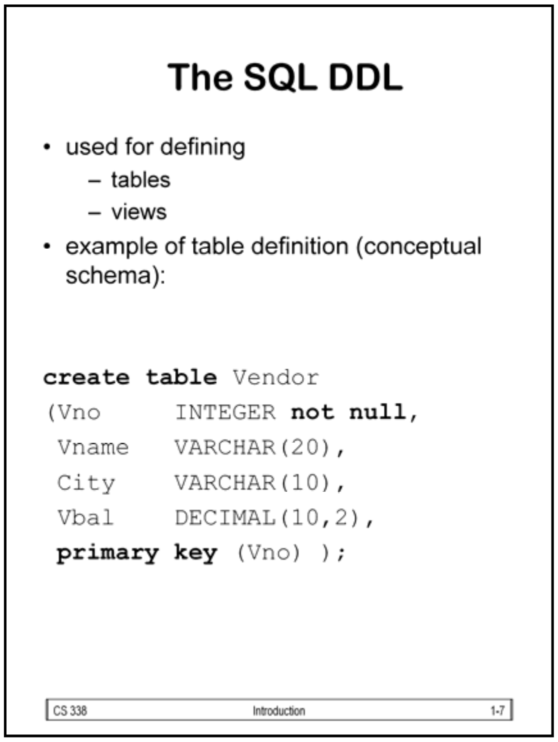
These two show the same relation.

Note that the domains (types) of the attributes are not shown in this diagram. This is a common convention.

Pictorial schema



arrows point towards relation (table) where the attribute domain is (column values are) defined
This servers as a constraint on the source relation: it says that the value in the source relation must occur in the destination relation. In terms of domains, it says that the domain of the source attribute is defined by the set of values in the destination relation.
Officially, this connection is called a foreign-key constraint. The word “foreign” is has the meaning “elsewhere” (ie we might just as well call it an “elsewhere” key constraint).



The SQL DDL

- used for defining
  - tables
  - views
- example of table definition (conceptual schema):

```
create table Vendor
(Vno      INTEGER not null,
 Vname    VARCHAR(20),
 City     VARCHAR(10),
 Vbal     DECIMAL(10,2),
 primary key (Vno) );
```

CS 338 Introduction 1-7

Back to SQL

one of the things is must do is be a DDL for DB, to define conceptual and external schemas. here is the SQL DDL for the credit-card-company database.

DDL to define the tables in the conceptual schema, explain:

keywords and identifiers,

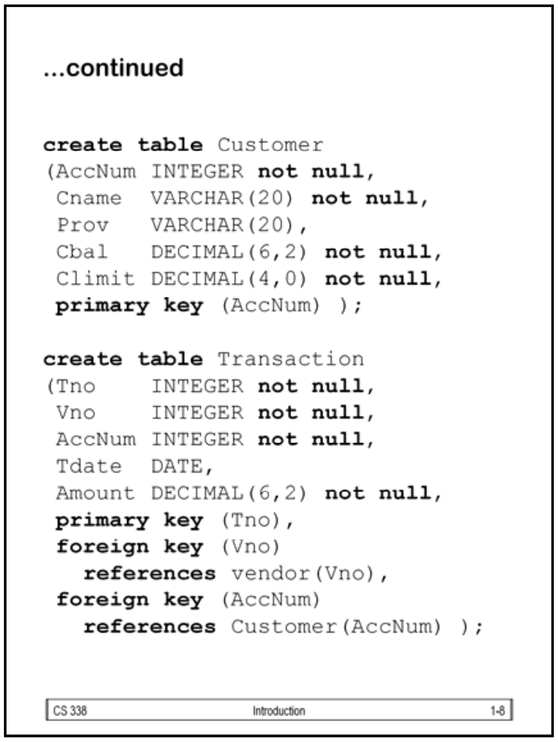
column names

data type keywords

primary key says the name of the column(s) that distinguish the row

"not null" unless otherwise specified, column values can be omitted (ie null), "not null" means cannot be omitted - common for primary keys. Strictly, “null” has a very complicated meaning that we needn’t get into just now.

Appearance is much like a programming language struct/record definition. Remember terminology, though: columns are attributes of the tuple, so this defines a tuple. datatypes are really attribute domains (set of possible values that an attribute can be).

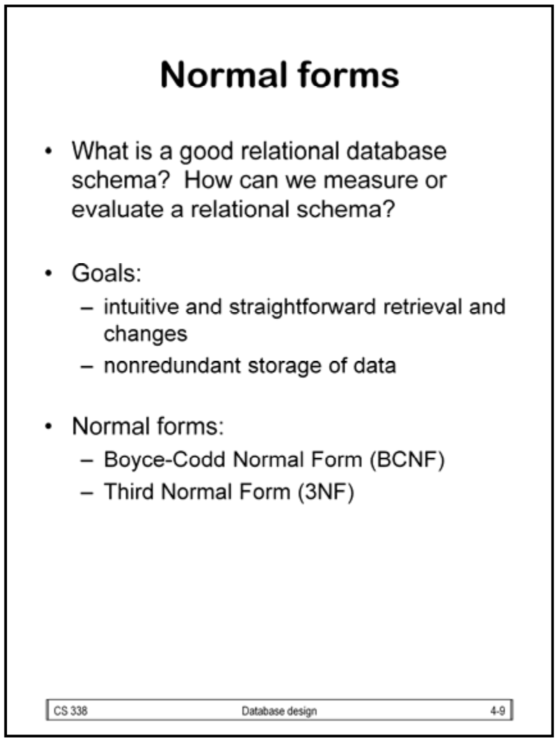


...continued

```
create table Customer
(AccNum INTEGER not null,
 Cname  VARCHAR(20) not null,
 Prov   VARCHAR(20),
 Cbal   DECIMAL(6,2) not null,
 Climit DECIMAL(4,0) not null,
 primary key (AccNum) );

create table Transaction
(Tno    INTEGER not null,
 Vno    INTEGER not null,
 AccNum INTEGER not null,
 Tdate  DATE,
 Amount DECIMAL(6,2) not null,
 primary key (Tno),
 foreign key (Vno)
   references vendor(Vno),
 foreign key (AccNum)
   references Customer(AccNum) );
```

more of the same.
note that "not null" can be used on any attribute, not just primary keys.
"foreign key" identifies the arrows that we saw in the relation diagram a couple of slides back. The "references" clause names other table and other column



Normal forms

- What is a good relational database schema? How can we measure or evaluate a relational schema?
- Goals:
  - intuitive and straightforward retrieval and changes
  - nonredundant storage of data
- Normal forms:
  - Boyce-Codd Normal Form (BCNF)
  - Third Normal Form (3NF)

CS 338 Database design 4-9

Now, turn our attention to different principles of DB design. Called normalization or "normal form" construction. Normal forms are standardized ways of constructing schemas and tables so that they conform to some set of rules. For example, we've already mentioned 1st NF: it's a rule that says that attribute instances must be single values.

Much of the interest in normal forms is motivated by the need to understand what constitutes a "good" schema. If we have two alternative designs, how can we decide which one is better? We need a way of precisely measuring the "goodness" of a schema, so that we can compare designs.

In order to measure quality, we must have some idea of what we want to achieve. DB design quality imvolves two conflicting goals: we want to avoid redundancy, ie don't store the same data more than once. We also want to have a schema that is a reasonable representation of the underlying enterprise rules, is understandable and easy to manipulate. There's no point in haveing a perfect design if it is untenable.

DB theorists have, over time, devised several normal forms that have many of the desired features.. Normal forms have rules that reduce or minimize redundancy, and, at the same time, represent all enterprise rules. The two best-known normals forms are 3NF and BCNF. Other exist (2nd, 4th, 5th) but we won't look at them. So, one way to measure the quality of a schema is to see if it conforms to one of the normal forms -- if it does, it is deemed to be "good" (acceptable, at least).

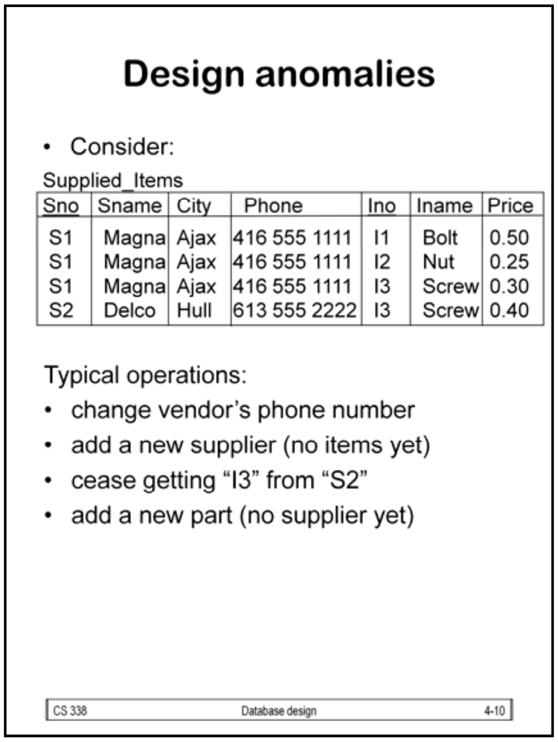
The real beauty of normal forms is that if a schema doesn't conform, the theory tells us how to rearrange the schema so that it does conform to a normal form. Doing so is called Normalization -- the act of taking a scehma and transforming it into a normal form. Thus, a discussion of normal forms as a quality measure implicitly includes a design methodology, since the two are really the same topic.

There are a number of different ways of approaching this topic. Traditional text, ours included, take a rigorous mathematical approach, based on a things like functional dependency, FD closure, minimal covers, lossy and lossless decompositions etc. This approach is necessary if one wants to work with theory, eg proofs about redundancy, deriving transformation algorithms.

Our approach will differ. Will attempt to demonstrate problems and give examples of solutions. Will define normal forms by example. We'll define some terminology and give some informal definitions. [without the underlying math, can't deal with "real" (ie math) definitions]

From a design methodology viewpoint, normalization begins with a single table, and then applies successive decompositions until desired normal form is reached.

It is interesting to note that even though ER modelling and normalization are quite different methods, they often produce very similar and even equivalent schemas. In fact the two methods reinforce each other, and knowledge of both is useful whichever one uses day-to-day.



Design anomalies

- Consider:

Supplied_Items

Sno	Sname	City	Phone	Ino	Iname	Price
S1	Magna	Ajax	416 555 1111	I1	Bolt	0.50
S1	Magna	Ajax	416 555 1111	I2	Nut	0.25
S1	Magna	Ajax	416 555 1111	I3	Screw	0.30
S2	Delco	Hull	613 555 2222	I3	Screw	0.40

Typical operations:

- change vendor’s phone number
- add a new supplier (no items yet)
- cease getting “I3” from “S2”
- add a new part (no supplier yet)

To start our discussion, let’s consider a bad design.

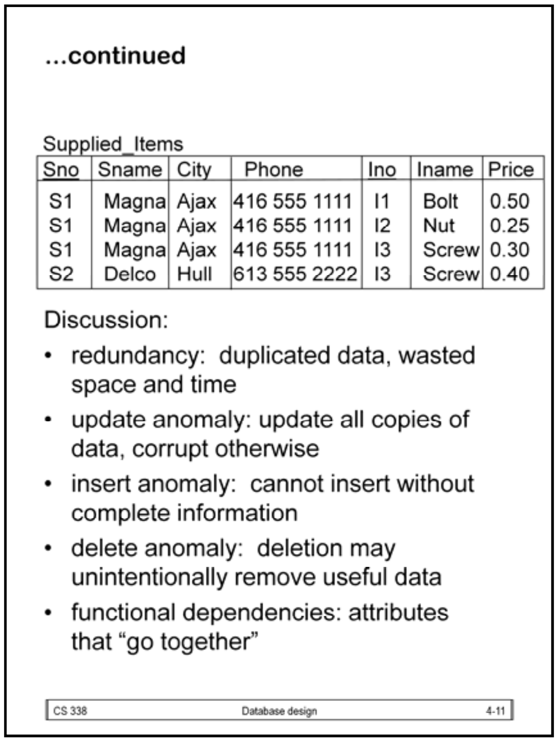
We have a database that is intended to keep track of the suppliers of parts for an parts vendor. It must track suppliers name,. location, what items we stock get from that supplier and theprice of the item.

We start out by tossing everything into a single table, formally known as a universal table. Because of the 1stNF rule, we have to duplicate indormation whenever we get more that one item from a supplier (can’t have an attribute that is a list of item) This organization is exceedingly common -- its a spreadsheet. We see this kind of data organization all the time. [any comments]

What kinds of things might we want to do? Queries -- probably OK. Changes/updates: eg changing a vendor’phone number

Adding: want to add information fo a supplier, but haven’t decided what parts to get from that supplier; add information about a part we intend to sell, but haven’t found a supplier for it yet.

deleting: decide to stop use Budd as a supplier for screws



...continued

Supplied_Items

Sno	Sname	City	Phone	Ino	Iname	Price
S1	Magna	Ajax	416 555 1111	I1	Bolt	0.50
S1	Magna	Ajax	416 555 1111	I2	Nut	0.25
S1	Magna	Ajax	416 555 1111	I3	Screw	0.30
S2	Delco	Hull	613 555 2222	I3	Screw	0.40

Discussion:

- redundancy: duplicated data, wasted space and time
- update anomaly: update all copies of data, corrupt otherwise
- insert anomaly: cannot insert without complete information
- delete anomaly: deletion may unintentionally remove useful data
- functional dependencies: attributes that "go together"

CS 338 Database design 4-11

Redundancy: lots of stuff in the table is duplicated. wasted space, takes time; If we were tempted to say that there should only be one row for each supplier, remember that 1NF precludes such an organization. Worse, it leads to...

Update anomaly: integrity problems: a change to one of s1's phone numbers must be made to all rows fo s1. if we don't, we will have a corrupt database, because it is clear that every entry for s1 should have the same phone number. Now, it turns out that it is possible in SQL to give a query that would update all the appropriate rows, but we would prefer a design that prevented the potential for the error [constraints? no, make inhereent in good design].

Insert anomaly: suppose that we want to add information about a vendor. with the current design, we must have information about at least one item, which may or may not be suitable. Similarly, if we add information about an item, we must have a vendor with which to associate that information. This is clearly not acceptable. Could use nulls, but still have all the same redundancy, space problems, really seriously complicates application logic. Besides, p-keys can't be null.

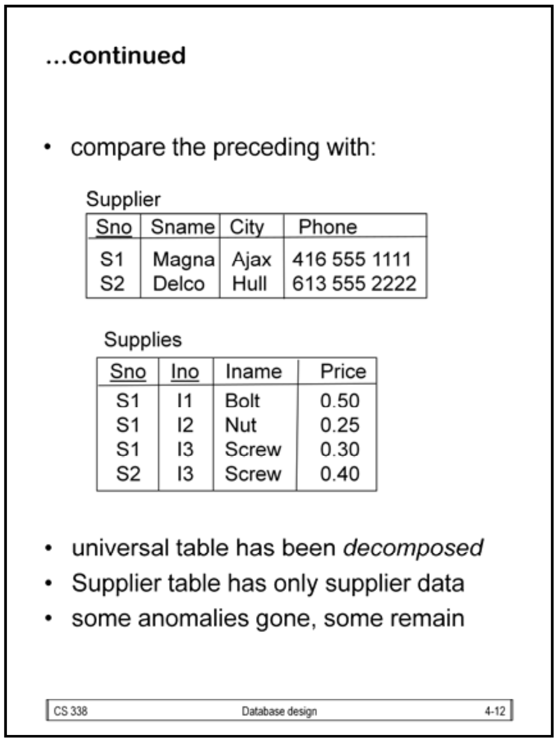
Delete anomaly: the same problem exists for deleting row as for inserting rows. suppose that we decide not to use the screws supplied by Budd. if we remove the row, we lose track of Budd entirely.

Functional dependencies: think about the update anomaly again. it occurs because a change to one of the instances of a phone implies a change to all of the instances of phone numbers forall rows that have the same sno. Eg a change to one of s1 phone numbers means all rows for s1 must be changed. In fact, there are several attributes that go together with sno. By inspection, we could presume that the name, city and phone attributes are related. If we find two rows with the same sno, then we expect to find the same values in name, city, phone.

This relationship is called a functional dependency: We say that attribute sno functionally determines attributes name, city and phone. Conversely, we say that city, name and phone are functionally dependent on sno.

Note also that Ino is **not** functionally dependent on Sno. Intuitively, this seem to be a problem -- the rows in the table seem to consist of two distinct groups: Supplier stuff and item stuff.

Note that in normal circumstances, the functional dependencies are not determined by i nspection, but are specifed as a part of the enterprise rules for the db that we are creating. The specification of functionsl dependencies is similar to specifying ER relationships



...continued

- compare the preceding with:

Supplier

Sno	Sname	City	Phone
S1	Magna	Ajax	416 555 1111
S2	Delco	Hull	613 555 2222

Supplies

Sno	Ino	Iname	Price
S1	I1	Bolt	0.50
S1	I2	Nut	0.25
S1	I3	Screw	0.30
S2	I3	Screw	0.40

- universal table has been *decomposed*
- Supplier table has only supplier data
- some anomalies gone, some remain

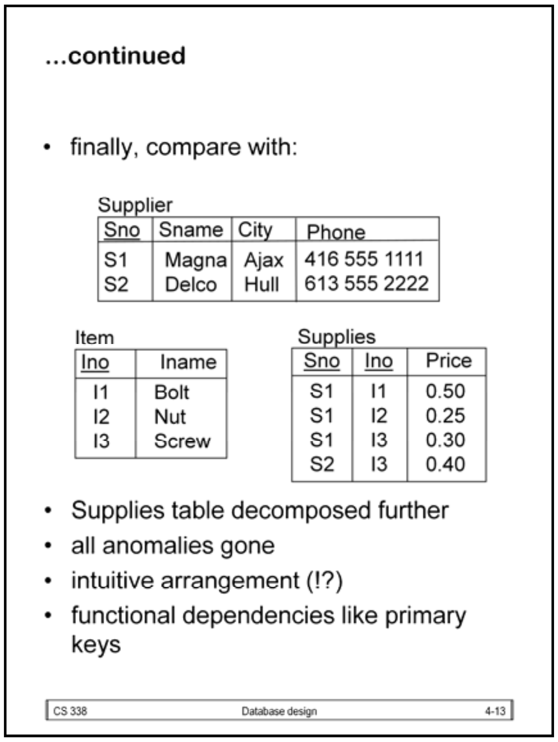
CS 338 Database design 4-12

Now, look at these two tables. They are a **decomposition** of the previous table opposite of join). We've split the information so that supplier informaiton is maintained independently from the item information. The association between a supplied item and its supplier is done with a foreign-key reference from the supplies table to the supplier table.

Splitting tables is actually quite a complicated matter. The textbook treatment on the topic is mathematically-based. The mathematical representation of a table is its heading (ie set of columns) and a set of functional dependencies. splitting a table is a matter of dividing the heading into two or more subsets.

With this set notation, we can do things like for example look at what columns are common in both parts of the decomposi tion -- this would be an intersection between the two subsets. One of the rules about decomposition, for example, is that the union of all of the decomposed table headings must be equal to the original table heading. Another rule is that joining all of the tables back together (with the relational join operator) should give back the original table. This property is referred to as lossless versus lossy decomposition,

In this example, we seem to have a pretty good decomposition: all the columns are there, and if you join these two tables, the original table results. The anomalies for supplier are gone. We still have a problem, though. If we want to change an item-name, there is still some redundancy, and so the update anomaly still exists. Similarly if we want to add a new item, we have to know its supplier and a price. So, ...



...continued

- finally, compare with:

Supplier

Sno	Sname	City	Phone
S1	Magna	Ajax	416 555 1111
S2	Delco	Hull	613 555 2222

Item

Ino	Iname
I1	Bolt
I2	Nut
I3	Screw

Supplies

Sno	Ino	Price
S1	I1	0.50
S1	I2	0.25
S1	I3	0.30
S2	I3	0.40

- Supplies table decomposed further
- all anomalies gone
- intuitive arrangement (!?)
- functional dependencies like primary keys

CS 338 Database design 4-13

Here we have split the suppliers table inteo even smaller parts. Now we have a suppliers table with information about tables, an items table with information about just the items, and a supplies table that records the information about which items come from what supplier, and the pricing information.

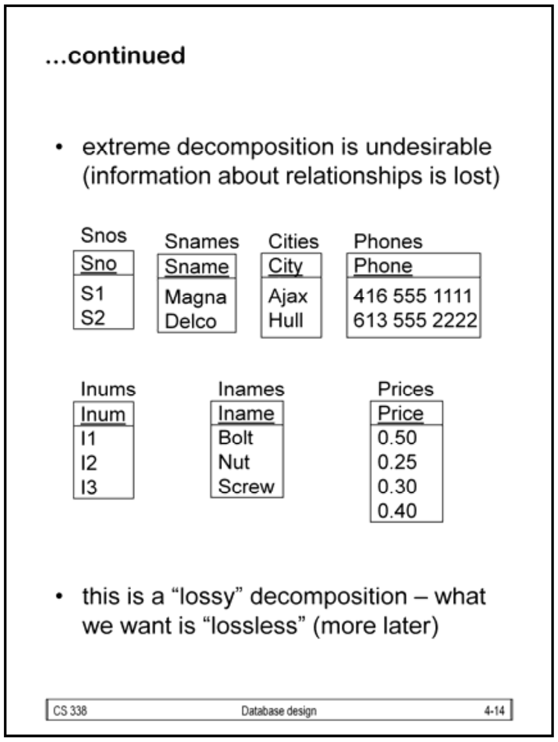
This decomposition looks really good. We have all the columns, and a join between these three will yield the original table (and, of course, a join bwtween item and supplies will yield the preceding version of supplies). Intuitively, this factoring seems good, because it puts related information into its own table, and has a separate table that represents the connection.

Of course, this arrangement should come as no surprise whatsoever, since these three tables are exactly what we would have created it we used an ER model, with supplier and item as the entities and supplies as the relati onship. [clever, eh?]

We mentioned earlier that functional dependencies are normally given as part of the enterprise rules for the database scheme. we didn't actually have any here, we just inferred them in a reasonable fashion. Indeed, the fact that the price in in the supplies table suggests that an item price varies depending upon who supplies it. The other possibility would be to place the price in the item table, in which case it would be more like a sale price. The difference is that we control a sale price, but the supplier controls the supplied price. Either is reasonable.

Using the tables as shown, we infer that Sno functionally determines Sname, city and phone, and that Ino functionally determines Iname, and that the Sno-Ino pair functionally determines price. Once again, these seem intuitive: for a given Sno, the name, city and phone number are fixed. any use or reference to that Sno implies the dependent attributes. If another row had the same Sno, then we woujld expect it to have the same three functionally-dependent attributes (of course, that can't happen in this table because of the uniqueness rules). Now, here's a coincidence: in each case, the attribute that functionally determines the other attribute happens to be the primary key of the table. Of course, this is no conincidence at all. The process of specifying functional dependencies is pretty much the same activity as defining entiy sets and their primary idenifiers in an ER model.

[these tables are in BCNF. in each table, every single attribute is functionally dependent on the key for the table. Every original FD is preserved in one of the tables (preserve: for every FD x->y there is a table where x u y is a subset of the table heading; or, dependencies aren't split across tables)



...continued

- extreme decomposition is undesirable (information about relationships is lost)

Snos

Sno
S1
S2

Snames

Sname
Magna
Delco

Cities

City
Ajax
Hull

Phones

Phone
416 555 1111
613 555 2222

Inums

Inum
I1
I2
I3

Inames

Iname
Bolt
Nut
Screw

Prices

Price
0.50
0.25
0.30
0.40

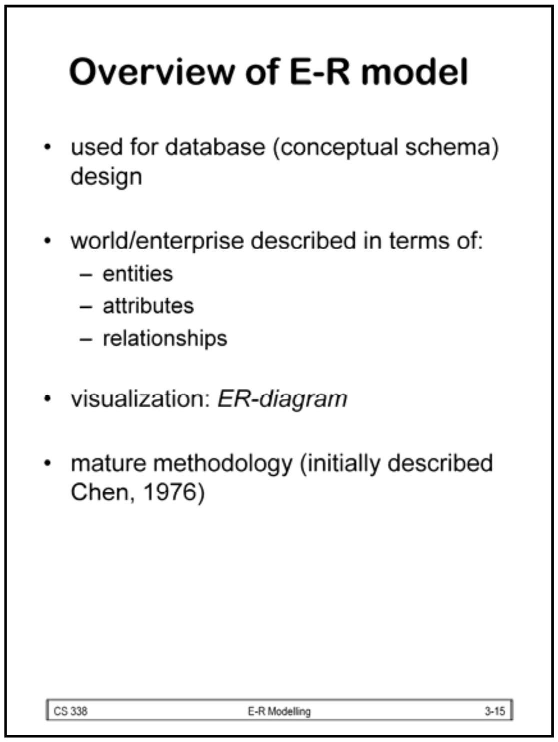
- this is a "lossy" decomposition – what we want is "lossless" (more later)

The decomposition process can be carried too far. In this case we've deposed everything into single-attribute tables. This is useless, for a number of reasons:

1) this decomposition is lossy. Remember that the result of decomposition should be able to be joined back together to yield the original table. The join of all these table is huge and certainly has not much to do with the original table.

2) we've completely lost any representation of the original functional dependencies. Eg. an sname can change independently of an sno. (in this decomposition, its so obvious that its hard to see) This is referred to as a non-dependency preserving decomposition. What we want are dependency preserving decompositions

[the importance of dependency preservation decompositions come up as follows. We can thing of functional dependencies as uniqueness constraints. if a fd lies in a table, then relational uniqueness guarantees no duplicate tuples. a non-dependency-preserving decomposition will split an fd constraint across two or more tables. when this happens, we have to join the tables in order to determine if the fd still holds]



Overview of E-R model

- used for database (conceptual schema) design
- world/enterprise described in terms of:
  - entities
  - attributes
  - relationships
- visualization: *ER-diagram*
- mature methodology (initially described Chen, 1976)

CS 338 E-R Modelling 3-15

So what is ER all about? Must understand that is is just a design methodology that we use to create the conceptual schema for a DB. Represents the overall structure of a DB. So, it qualifies as as DDL, but as we'll see, it is a two-step method that models data graphically first, then produce table definitions, which we can render in SQL.

The process of constructing an ER model for a DB involves significant understanding of the real-world enterprise that is being modelled. Sometimes hear the term enterprise model or enterprise scheme.

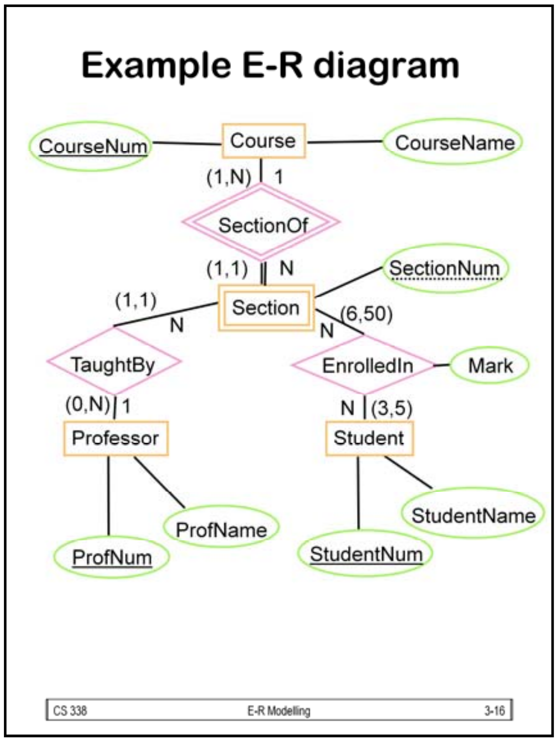
At the core of ER modelling is the concept of viewing the world as entities -- things, objects, identifiable, distinguishable. Eg customer, car, bank account, course, classroom, instructor. In all cases, we can somehow tell entities apart.

The way that we distinguish entities is with their attributes or properties. Eg customer have different names, cars are different models or colours, bank accounts have numbers, classrooms have locations and capacities. So not only do we have entities, the entities have attributes. These attributes can be descriptive, or can describe real-world be limitations on the attributes. Latter case are often called constraints.

Once we have a collection of entities, we define relationships between the entities, ie how the entities interact with each other. Eg a customer own a car, a course has an instructor.

ER modelling is a visual design method. It uses formalized diagrams (ER diagrams) to describe entities, their attributes and their relationships. Once constructed, ER diagram is a visual representation of the compete enterprise scheme. It can them be translated into a relational schema and implemented in SQL. It can also be translated into any number of other DDL. Some people have experimented with query languages that deal directly with ER diagrams.

Note that ER modelling is a mature technology, first described 20 years ago



So, a big example with lots of stuff.

section:course is n:1 there can be many sections of 1 course

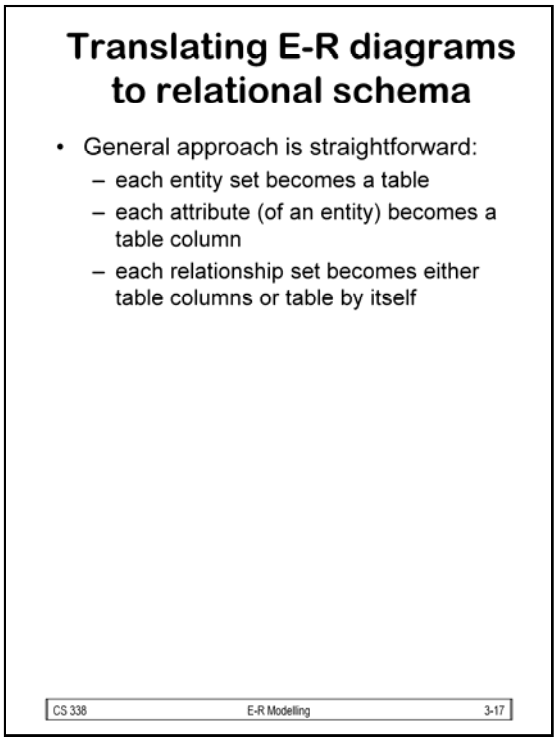
a course can have min 1, max N sections; a given section can be a section of only a single course

section:professor is n:1 there can be many sections taught by one professor

a section is taught by min 1 max 1 professors; a professor can teach from 0 to N sections

section:student is n:n sections have many students, students take many sections

a section has a min of 6 max of 50 enrolled in it; a student enrolls in min 3 max 5 sections



Translating E-R diagrams to relational schema

- General approach is straightforward:
  - each entity set becomes a table
  - each attribute (of an entity) becomes a table column
  - each relationship set becomes either table columns or table by itself

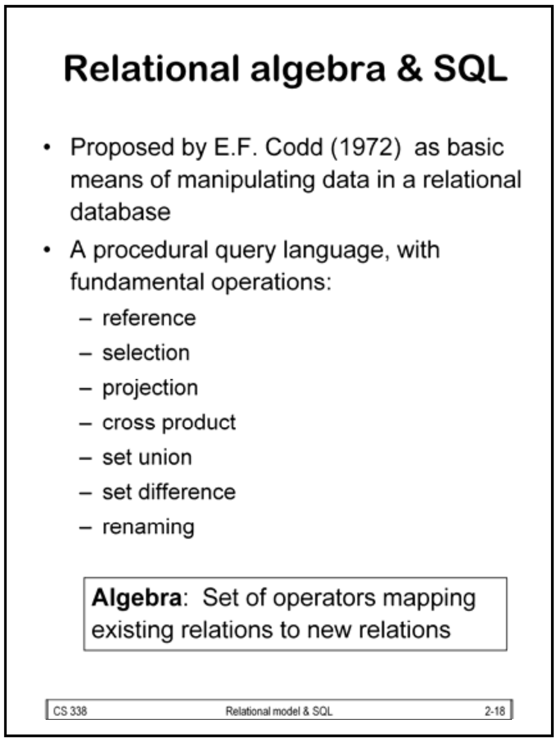
For the most part, converting a diagrams to a set of table definitions is obvious. We’ll look at several examples, covering each of the different bits of ER diagram techniques that we looked at last time. Text references is the “transformation rules” that pop up occassionally in the discussion of modelling.

This stuff really is quite intuitive -- don’t make the mistake of trying to read too much into it.

So, ER diagrams consist of entity sets and relationship setss:

No surprise, entity set becomes a table, attributes of the entity become columns.

As it turns out, relationship sets also turn into tables, or nothing at all.



Relational algebra & SQL

- Proposed by E.F. Codd (1972) as basic means of manipulating data in a relational database
- A procedural query language, with fundamental operations:
  - reference
  - selection
  - projection
  - cross product
  - set union
  - set difference
  - renaming

Algebra: Set of operators mapping existing relations to new relations

CS 338 Relational model & SQL 2-18

We've been skirting around the issue that there is a concise mathematical basis for the relational DB model. Will take an informal look at this.

A relation is a connection between attributes and values, viewed as a table.

Mathematically, a relation is a set (a set of tuples that consist of attribute-value pairs). There is a well-established mathematical system for doing things to sets, including combining sets and choosing subsets.

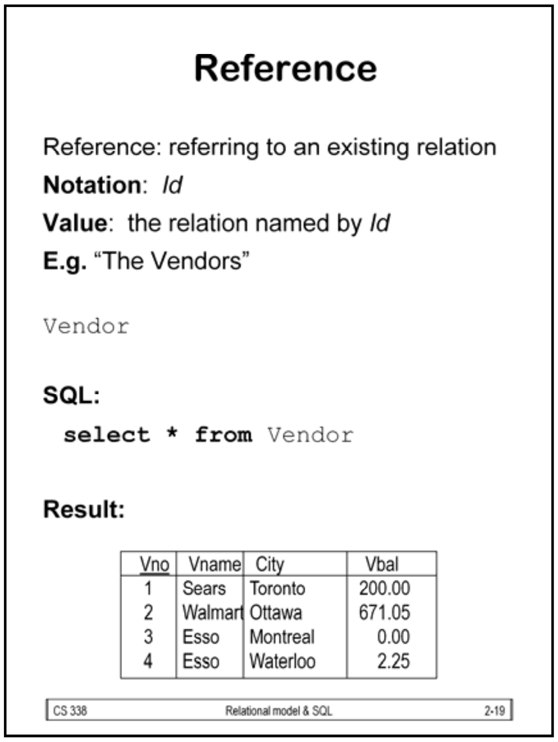
Formally we're talking about an algebraic system, or just an algebra. An algebra consists of operators (actions, verbs) and operands (objects, entities, nouns). Classic example is "algebra" ("ie "the algebra", no indefinite article) which is an algebra of numbers and arithmetic operators like addition and multiplication. We're interested in an algebra of relations: relational algebra..

Mathematically, there are all sorts of assumptions, rules and definitions that go along with an algebraic system. One very important one says that the result of an operation between two operands is the same things as the operands. Eg adding two integers results in another integer. this principle is called closure. Our algebra of relations will be **closed**, so that the result of an operation between relations is another relation.

Why do we care about all this stuff? Because the relational algebra we define is, for the most part, the query language that we will use to do things to relational databases. Some of the operators are ... [refer to list]. As we look at the definition of these operations, we will show examples using a DML-style query.

Note about syntax: we use the text, others exist

Now take a look at several relational algebra operators



Reference

Reference: referring to an existing relation

Notation: *Id*

Value: the relation named by *Id*

E.g. "The Vendors"

```
Vendor
```

SQL:

```
 select * from Vendor
```

Result:

Vno	Vname	City	Vbal
1	Sears	Toronto	200.00
2	Walmart	Ottawa	671.05
3	Esso	Montreal	0.00
4	Esso	Waterloo	2.25

CS 338 Relational model & SQL 2-19

First operation is simple: reference. The result relation is the set of tuples contained in the relation whose name is given.

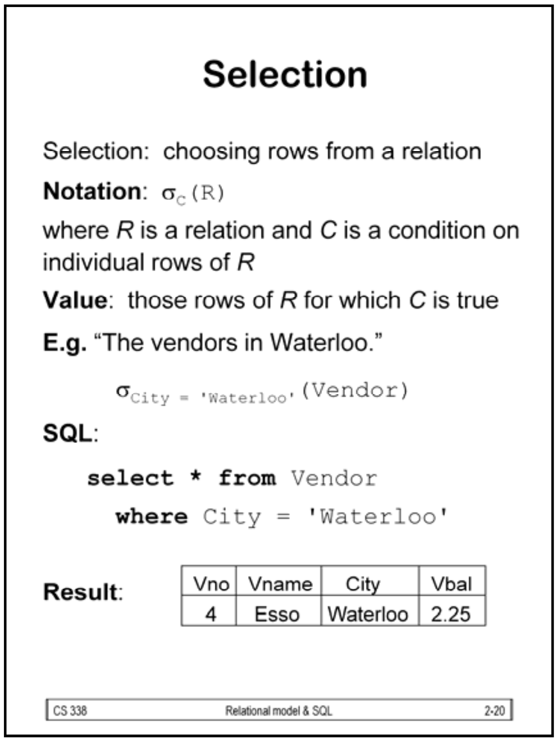
Examples: in our examples, give an English phrase describing the relation that we want (natural-language phrase), then give DML syntax.

From operator-operand viewpoint, "reference" is a unary operator: it results in the named relation. Idempotent

—>Mention notation

[example]

- syntax is the name of the relation
- result is the named relation



Selection

Selection: choosing rows from a relation

Notation: σ_C(R)

where *R* is a relation and *C* is a condition on individual rows of *R*

Value: those rows of *R* for which *C* is true

E.g. "The vendors in Waterloo."

$\sigma_{\text{City = 'Waterloo'}}$(Vendor)

SQL:

```
select * from Vendor
  where City = 'Waterloo'
```

Result:

Vno	Vname	City	Vbal
4	Esso	Waterloo	2.25

CS 338 Relational model & SQL 2-20

Selection: a little more complex. used to choose a subset of the tuples in a relation.

Will give the relation from which we want to choose, and some kind of selection criteria (a condition).

Result is the rows in the relations for which the condition is true.

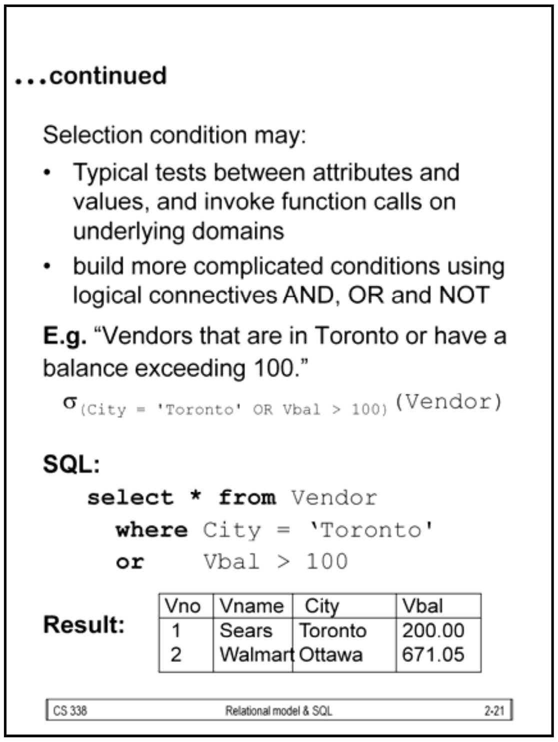
[example]

- syntax

- "reference" followed by "where clause"

condition here is "=" test for equality: must match exactly

Notation: $\sigma_{\text{condition}}$(relation) eg: $\sigma_{\text{city=‘waterloo’}}$(Vendor)



...continued

Selection condition may:

- Typical tests between attributes and values, and invoke function calls on underlying domains
- build more complicated conditions using logical connectives AND, OR and NOT

E.g. "Vendors that are in Toronto or have a balance exceeding 100."

$\sigma_{\text{(City = 'Toronto' OR Vbal > 100)}}\text{(Vendor)}$

SQL:

```
select * from Vendor
  where City = 'Toronto'
  or    Vbal > 100
```

Result:

Vno	Vname	City	Vbal
1	Sears	Toronto	200.00
2	Walmart	Ottawa	671.05

CS 338 Relational model & SQL 2-21

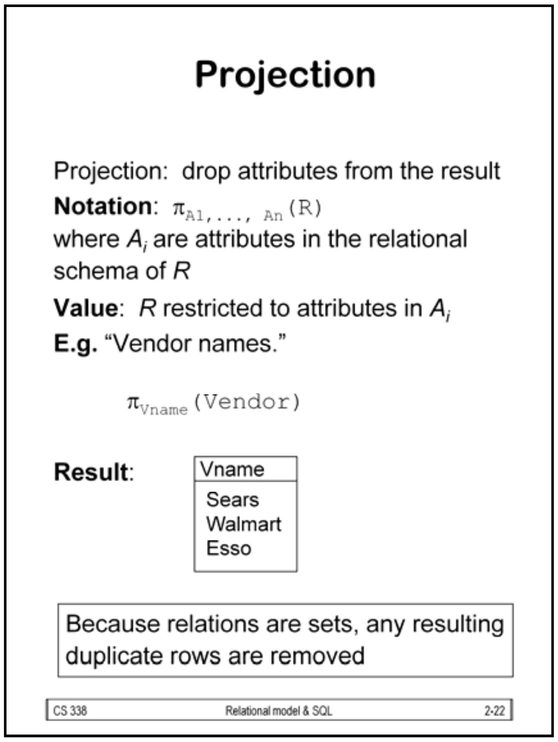
Conditions can be more sophisticated. Not only equality, but other comparisons, too. Eg ordering (greater, less) for numbers (maybe strings). In general, and operation that is defined on the domain of the attribute can be used. Eg for strings, could have operations that convert to upper case, return length of string, trim spaces. For numbers, usual arithmetic ops.

Can create complex conditions by connecting simple ones together with logical operators. [explain and or not??]

[example]

- two parts

- parentheses



Projection

Projection: drop attributes from the result

Notation: $\pi_{A1,\ldots,\ An}(R)$
where A_i are attributes in the relational schema of R

Value: R restricted to attributes in A_i

E.g. "Vendor names."

$\pi_{Vname}(Vendor)$

Result:

Vname
Sears
Walmart
Esso

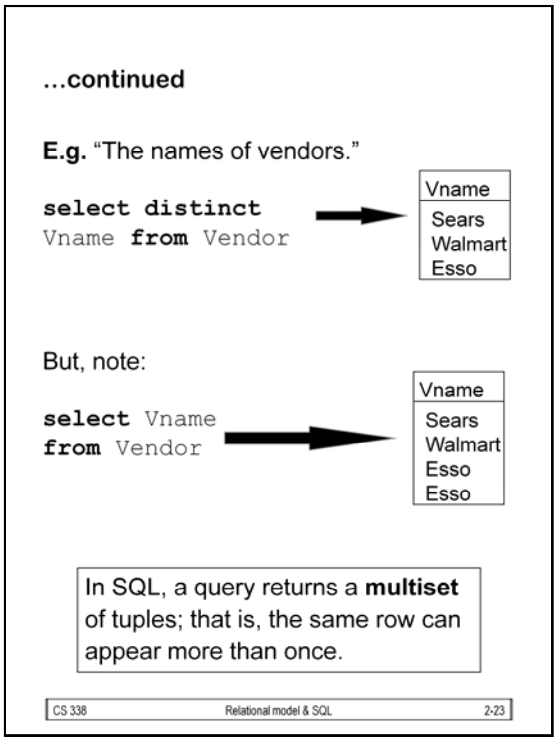
Because relations are sets, any resulting duplicate rows are removed

projection: select one or more columns (ie from each tuple in the relation, form a new relation using only the attributes indicated)

[example]
-syntax

Recall that relations are sets of tuples. Basic premise about sets is that there can be no duplicates, so if there are any dups in the result, discard them. Another way of looking at this is to say that the result must be a relation, and each tuple in a relation must be distinguished from each other.

Notation: $\Pi_{attr}(Relation)$ eg $\Pi_{vname}(Vendor)$



...continued

E.g. "The names of vendors."

```
select distinct
Vname from Vendor
```

Vname
Sears
Walmart
Esso

But, note:

```
select Vname
from Vendor
```

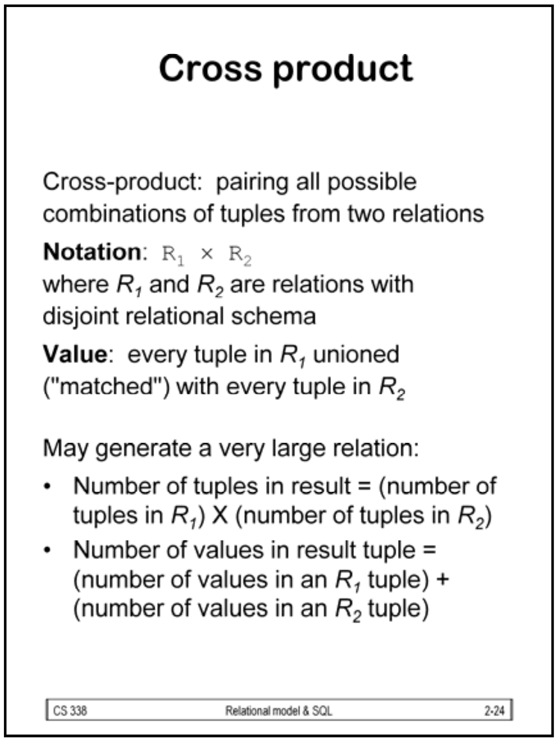
Vname
Sears
Walmart
Esso
Esso

In SQL, a query returns a **multiset** of tuples; that is, the same row can appear more than once.

Projection: choosing specific columns

Here we see a noticeable difference between SQL and the underlying relational algebra. In RA, relations are sets, and cannot have duplicates. In SQL, duplicate rows are allowed.
So, use the keyword "distinct" to indicate that we want duplicate rows removes from the result of the selection.

using distinct is expensive and should be avoided unless absolutely necessary. in a properly-designed database, should be able to eliminate most occurrences of distinct. But: get the query correct first, then worry about performance



Cross product

Cross-product: pairing all possible combinations of tuples from two relations

Notation: $R_1 \times R_2$
where R_1 and R_2 are relations with disjoint relational schema

Value: every tuple in R_1 unioned ("matched") with every tuple in R_2

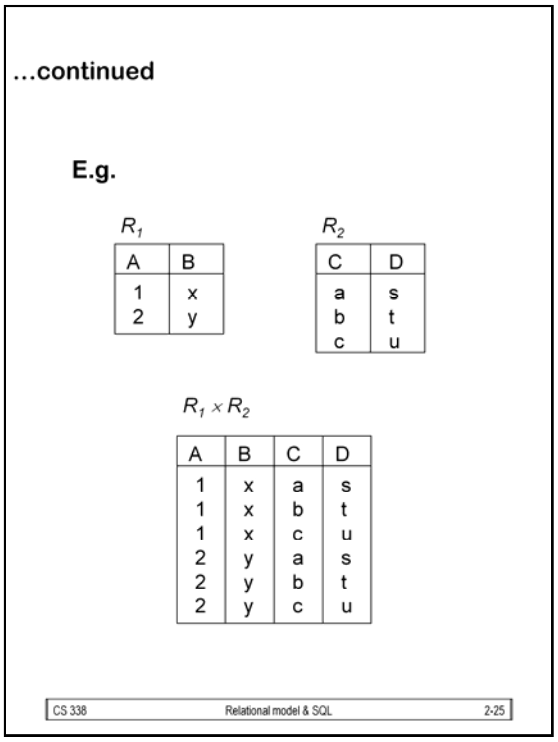
May generate a very large relation:

- Number of tuples in result = (number of tuples in R_1) X (number of tuples in R_2)
- Number of values in result tuple = (number of values in an R_1 tuple) + (number of values in an R_2 tuple)

Cross product: for every row in R1, match it up with every row in R2.

[generally as stated. skip to example then back]

note that schemas must be disjoint: no common attributes. If there were, the crossproduct would have two attributes with the same name, which isn't permitted.



...continued

E.g.

R_1

A	B
1	x
2	y

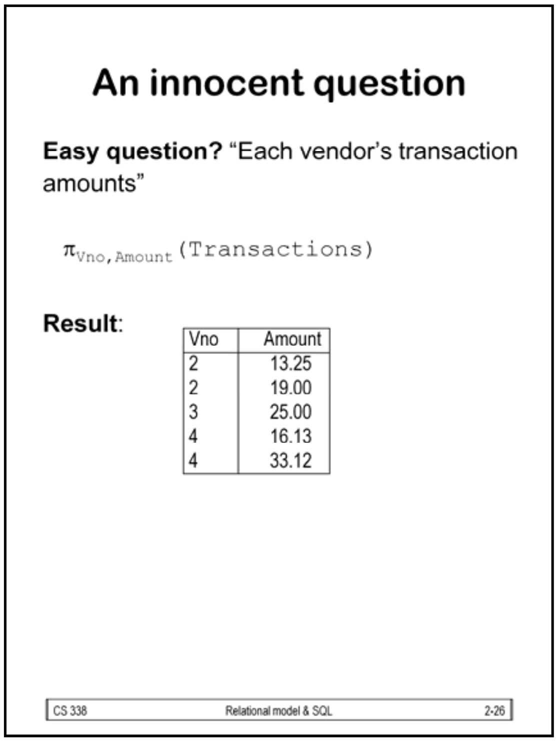
R_2

C	D
a	s
b	t
c	u

$R_1 \times R_2$

A	B	C	D
1	x	a	s
1	x	b	t
1	x	c	u
2	y	a	s
2	y	b	t
2	y	c	u

walk-through:
1, x with a, s
b, t
c, u
2, y with a, s etc



An innocent question

Easy question? "Each vendor's transaction amounts"

$\pi_{\texttt{Vno,Amount}}$ (Transactions)

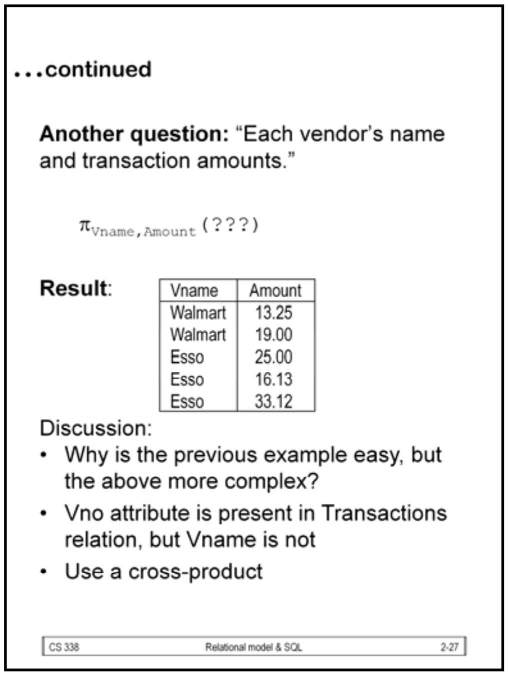
Result:

Vno	Amount
2	13.25
2	19.00
3	25.00
4	16.13
4	33.12

all the rows from one table without the rows from the other table
like union, tables have to have the same shape (relation schemas must be the same)

[eg]
doing a set difference between two projections; vendors[vno] produces a relation with a single attribute (the vno), transactions[vno] does the same. So, we have two (temporary) relations with same schema. Difference is the rows in the left one with the rows from the right one removed.

[work following example on board]
numerically: vendors[vno] has 4 rows: vnos 1, 2, 3, 4
trans[vno] has 3 rows: vnos 2, 3, 4 (trans has 5 rows, but projection removes duplicates)
so, {1, 2, 3, 4 } - {2, 3, 4 } = {1}



...continued

Another question: "Each vendor's name and transaction amounts."

$\pi_{\text{Vname,Amount}}(???)$

Result:

Vname	Amount
Walmart	13.25
Walmart	19.00
Esso	25.00
Esso	16.13
Esso	33.12

Discussion:

- Why is the previous example easy, but the above more complex?
- Vno attribute is present in Transactions relation, but Vname is not
- Use a cross-product

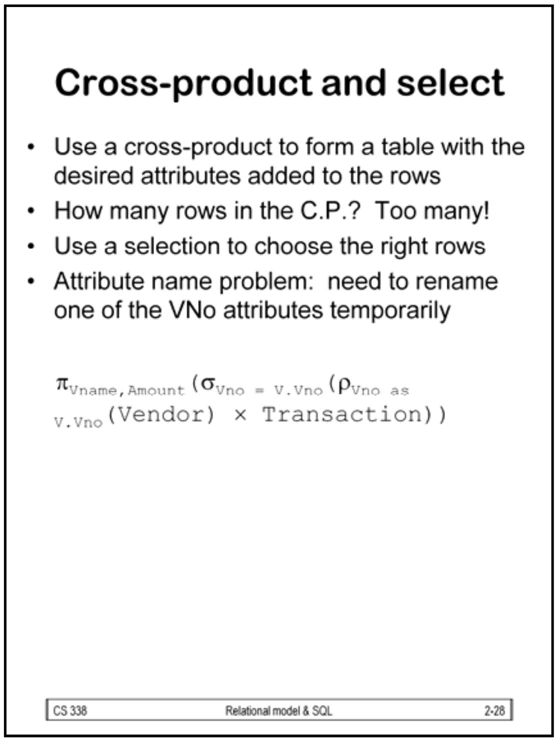
A very similar question.
Can ??? be "transactions" -- no, it doesn't have a Vname attribute
Intuitively, we know the answer is "sears", but how do we get it with a single relational expression?

Whatever ??? is, its schema must have a Vname attribute, and must contain tuples corresponding to transactions. Sort of like transactions with Vname added.

The essential operation here is still to do a subtraction: vendors without transactions. in the previous case it's straightforward, since we can do a subtraction directly on the projections of each table (each table has the desired Vno).

However, in this case, the RHS of the subtraction is more complicated. recall cross-product: gives all combinations of both relations. In our case, this would consist of tuples with both a vno and a vname (plus lots extra).
unfortunately, the CP has some problems. First of all, it will have lots of tuples, more that we want (we want just the tuples that represent the 5 transactions). We start out with a cross product between the two tables, then select the rows that represent the original transactions. Also have a problem with rule about disjoint schemas for CP: both relations have a vno, so can't cross them directly.

==========
also try: vendor[vname] where vno in vendor[vno] - transaction[vno]
vendor where vno in vendor[vno] - transaction[vno]



Cross-product and select

- Use a cross-product to form a table with the desired attributes added to the rows
- How many rows in the C.P.? Too many!
- Use a selection to choose the right rows
- Attribute name problem: need to rename one of the VNo attributes temporarily

$\pi_{\texttt{Vname,Amount}}(\sigma_{\texttt{Vno = V.Vno}}(\rho_{\texttt{Vno as V.Vno}}(\texttt{Vendor}) \times \texttt{Transaction}))$

CS 338 Relational model & SQL 2-28

Cross-product of relations whose schemas have common attribute occurs often Relational algebra has a "rename" operator to handle the problem.

Uses {} and "as" as the rename operator.

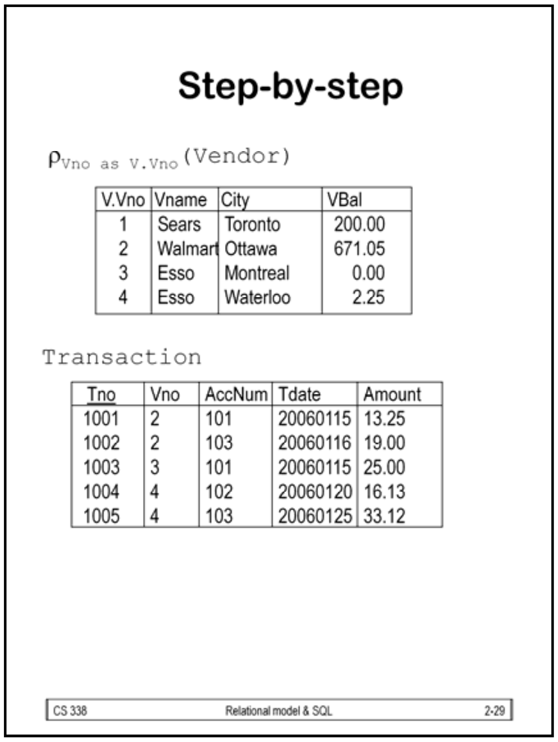
result relation is same as started with, with selected attributes renamed

So, answer to pending question is ...

note rename prior to product,

note where: says we only want the rows from the CP where the vendor vno is the same as the transaction vno (ie the original transactions)

[next notes page: expression tree]



Step-by-step

$\rho_{\texttt{Vno as V.Vno}}$`(Vendor)`

V.Vno	Vname	City	VBal
1	Sears	Toronto	200.00
2	Walmart	Ottawa	671.05
3	Esso	Montreal	0.00
4	Esso	Waterloo	2.25

`Transaction`

Tno	Vno	AccNum	Tdate	Amount
1001	2	101	20060115	13.25
1002	2	103	20060116	19.00
1003	3	101	20060115	25.00
1004	4	102	20060120	16.13
1005	4	103	20060125	33.12

[optional material if suitable for class]
An expression tree for the query:

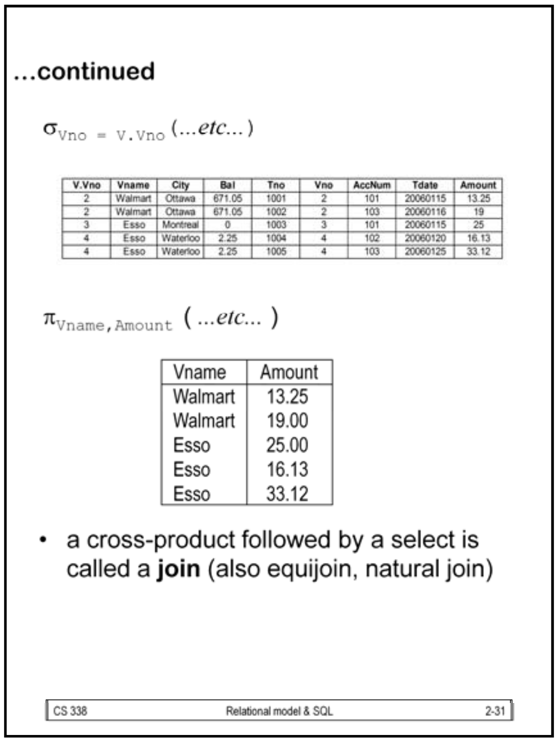
```
           t4: minus
          /        \__________
          |                   |
       [vname]            t3:[vname]
         |                    |
        vendor           where vno=v.vno
                              |
                           t2:times
                          /        \
           t1:{vno as v.vno}       transaction
                     |
                   vendor
```

(where t# references transactions given subsequently)

...continued

$\rho_{\texttt{Vno as V.Vno}}(\texttt{Vendor}) \times \texttt{Transaction}$

V.Vno	Vname	City	Bal	Tno	Vno	AccNum	Tdate	Amount
1	Sears	Toronto	200	1001	2	101	20060115	13.25
1	Sears	Toronto	200	1002	2	103	20060116	19
1	Sears	Toronto	200	1003	3	101	20060115	25
1	Sears	Toronto	200	1004	4	102	20060120	16.13
1	Sears	Toronto	200	1005	4	103	20060125	33.12
2	Walmart	Ottawa	671.05	1001	2	101	20060115	13.25
2	Walmart	Ottawa	671.05	1002	2	103	20060116	19
2	Walmart	Ottawa	671.05	1003	3	101	20060115	25
2	Walmart	Ottawa	671.05	1004	4	102	20060120	16.13
2	Walmart	Ottawa	671.05	1005	4	103	20060125	33.12
3	Esso	Montreal	0	1001	2	101	20060115	13.25
3	Esso	Montreal	0	1002	2	103	20060116	19
3	Esso	Montreal	0	1003	3	101	20060115	25
3	Esso	Montreal	0	1004	4	102	20060120	16.13
3	Esso	Montreal	0	1005	4	103	20060125	33.12
4	Esso	Waterloo	2.25	1001	2	101	20060115	13.25
4	Esso	Waterloo	2.25	1002	2	103	20060116	19
4	Esso	Waterloo	2.25	1003	3	101	20060115	25
4	Esso	Waterloo	2.25	1004	4	102	20060120	16.13
4	Esso	Waterloo	2.25	1005	4	103	20060125	33.12



…continued

$\sigma_{\texttt{Vno = V.Vno}}$ (…*etc*…)

V.Vno	Vname	City	Bal	Tno	Vno	AccNum	Tdate	Amount
2	Walmart	Ottawa	671.05	1001	2	101	20060115	13.25
2	Walmart	Ottawa	671.05	1002	2	103	20060116	19
3	Esso	Montreal	0	1003	3	101	20060115	25
4	Esso	Waterloo	2.25	1004	4	102	20060120	16.13
4	Esso	Waterloo	2.25	1005	4	103	20060125	33.12

$\pi_{\texttt{Vname,Amount}}$ (…*etc*…)

Vname	Amount
Walmart	13.25
Walmart	19.00
Esso	25.00
Esso	16.13
Esso	33.12

- a cross-product followed by a select is called a **join** (also equijoin, natural join)

This kind of connecting or joining of tables is really useful. Comes up whenever we have a query that requires attributes from more that one table
In fact. this "crossproduct and select" operation is so common that it is often defined as a basic operator, although we've see that it isn't really.
Called a join operator (aka natural join, eqijoin, key join); it "joins two tables together using one or more common or shared attributes.

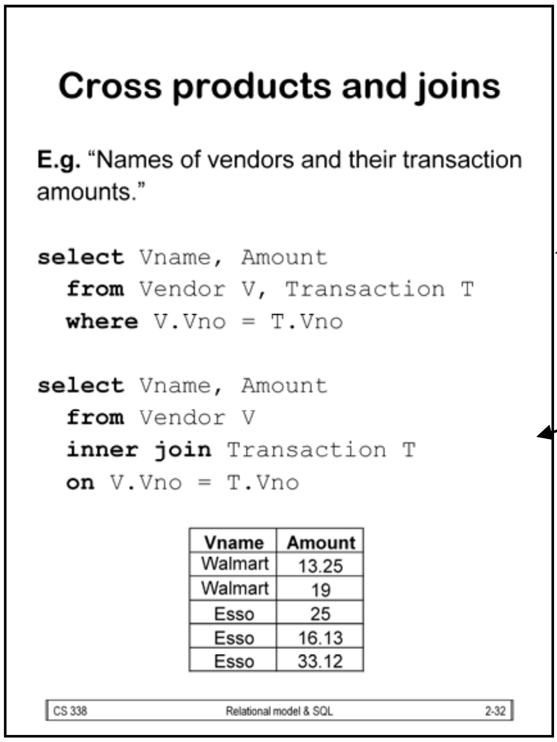
Relational algebra has some other "composite" operators.

Notation: A |><| B

========== additional notes for division, next page ===========================

for relations r1 and r2, assume that all attributes of r2 are in r1, and r1 has extra attribute. The result of r1/r2 has exactly those extra attributes. So in our eg. we have r1=(v,c) and r2=(c); the result r1/r2 has attribute (c).

Algorithm for dividing r1/r2 where r2 consists of single attribute: group r1 into groups of the result attribute. Eg groups r1 into groups over the (c) attribute. Each group of (c) values contributes exactly one value to the result (ie the value of (c) for the group), as long as the values in r2 (attribute (v)) are a subset of the values of (v) in the group.



Cross products and joins

E.g. “Names of vendors and their transaction amounts.”

```
select Vname, Amount
  from Vendor V, Transaction T
  where V.Vno = T.Vno

select Vname, Amount
  from Vendor V
  inner join Transaction T
  on V.Vno = T.Vno
```

Vname	Amount
Walmart	13.25
Walmart	19
Esso	25
Esso	16.13
Esso	33.12

CS 338 Relational model & SQL 2-32

This is a cross-product

This is a join – it selects only the rows where the common key matches

simple cross-product is an implicit operation when a list of tables is given.
So, eg: vendor, transactions does a cross-product and yields the 20-row, 9 column tables we saw previously.

These tables have commonly-named columns. Notice that we don’t necessarily have a problem with duplicate column-names, SQL doesn’t care in this case, since we aren’t referring to any of the duplicated columns

However, in cases where want to refer to a commonly-named column, need to create an alternate name. This typically arises when we want to do a join (since there is supposed to be a common column)
Use: table-name as alternate; can then use alias wherever needed to disambiguate. Renames the whole table, not just one column

Alternate name: correlation name == tuple variables = table alias;

More later