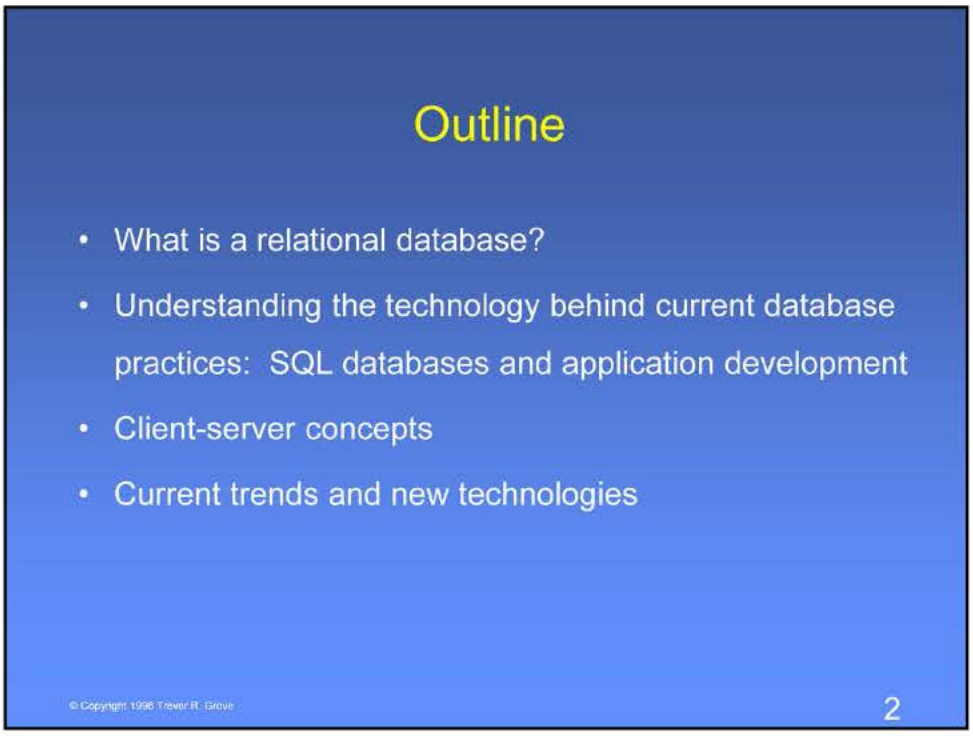
Technology Briefing: SQL and Relational Databases

Presented to

[redacted]

May 23, 1996

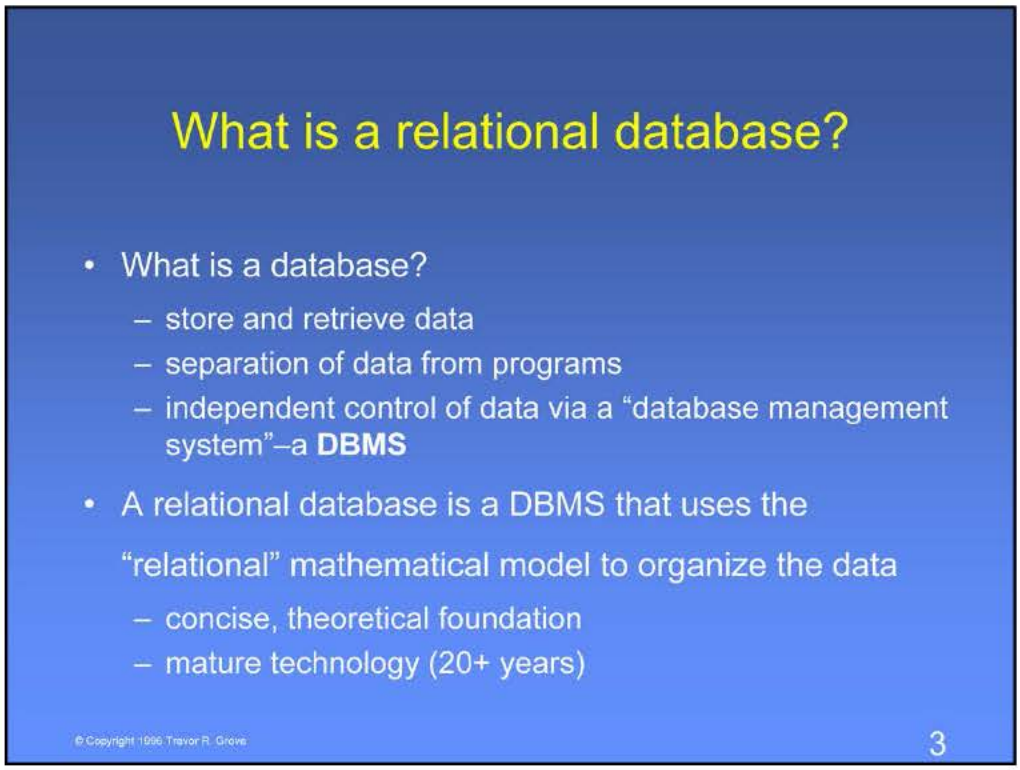
Trevor Grove
Research Associate
Computer Systems Group
University of Waterloo



This lecture: three parts.

Why: too much data, too little organization. Knowledge, information, data (data = raw facts, information = organized data, knowledge = applied information). In “information age”, where information & knowledge is currency of wealth, want to maximize effectiveness of data, infor and knowledge to get most we can.

SQL SQL is Structured Query Language (pronounced “sequel” by some...). Once existing data is organized, need to be able to add to it, retrieve it, work with it. SQL is a standard language used to interact with relational DBs.



Before we can say what a relational database is, need to understand what any kind of database is.

All computer programs are sequences of instructions that manipulate data: eg a program that adds 2 + 3 is manipulating data, so is it a database?

No. We call a database a program that manages data for other programs. Eg our adding program would ask the database for the first number, then the second, and then would do the arithmetic.

In this way we achive a separation of the data from the programs that manipulate the data.

Another example

Concepts: Think of manual filing system. Design a filing system to store data and information, facilitate retrieval. searching not always so obvious, so organize system to find stuff with only one or two probes. But consider example: bought a toaster, paid with credit-card. need to file paperwork. Have: file for house, files for credit-card. also file for warranty cards (but proof-of-purchase must accompany waranty card). How to file? Everything in CC file implies must remember how paid. Everything in house, gets too big, becomes shoe-box. Warranty file? maybe, but it has warranty for evertyhing, half of which are expired Three choices, how to choose? Choosing any single place makes retrieval harder (have to remember where things are) Could make copies and file everywhere: increases likelihood of retrieval, but very redundant. Could store one place, leave little reminders everyrwhere else. Ok, but a pain.

Computer-based systems have some of the same problems, but have extra abilities esp wrt searching and creating reminders. So, design of computer-based systems probably not be same as manual systems. Need to look at data storage and retrieval with a view to computer-based systems. We will look at a form of data organization called relational model (hence relational DB).

We'll comback to this in a minute

So a relational database is just a database that ...

We need to understand the principles behind databases in general before we worry about relational ones.

Buzzwords, TLAs and FLAs

- DBMS
- relational
- schema
- DDL
- DML
- SQL
- constraints
- referential integrity
- data dictionary
- 3NF, BCNF
- client–server
- data independence
- views
- relations
- attributes
- domains
- atomic
- ESQL
- ODBC
- ER modelling
- data warehousing
- OODB

Applications of database technology

"Classic":

- inventory control; payroll
- electronic funds transfer; reservations systems

Recent:

- computer aided design (CAD); computer aided software engineering (CASE), development environments
- geographic/environmental information systems (GIS, EIS); telecommunications systems (AIN)
- document storage and retreival

Computing and DB have gone together since almost "day 1" . They maybe haven't been obvious or apparent, but they've

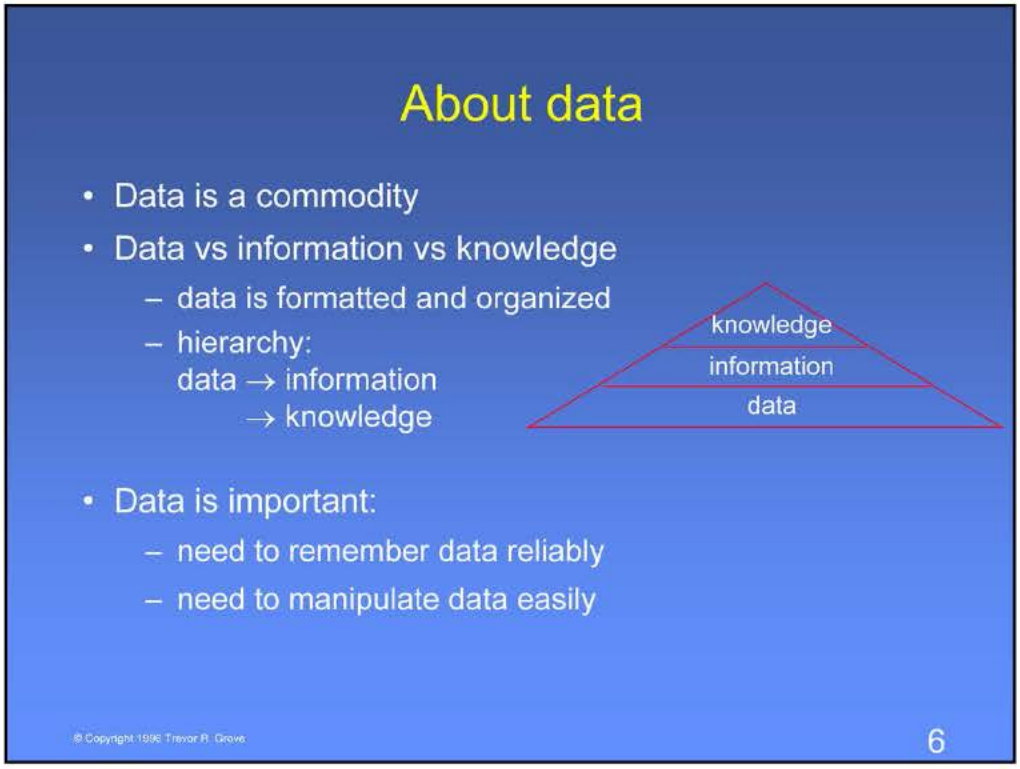
Classical applications like inventory and payroll. Not online, but still a database. Earliest online systems included reservations systems like Sabre and others developed for air-travel industry, funds transfer systems at insitutuional level and retail level in ATMs

Newer applications bring complete database systems to desktop for single-use, even embedded into other single-use applications.

Examples include sw-eng systems that use databases to store information about programs and systems. some even store source-code in database.

CAD and GIS systems represent drawings as points, line, shapes; use DB to store and retrieve these. Organize data into layers, can present drawings with different detail levels etc.

Telecommunications: very fast real-time DB. (AIN = Advanced Intelligent Network). Put DB apps in phone switch. Eg "310" numbers. Process transactions between time finish dialing and time of first ring.



If we were discussing a business that dealt with widjets, it would be appripriate to understand widjets: size, shape, cost, lifespan

View data in the same light -- it's a commodity that has properties like any other commodity. Now, some corporate data fulfills a record-keeping role, keeping track of corporate revenue and expenses. However, the terms "information age. info society, knowledge-based business" etc treat information as a corporate resource that must be treated like iron-ore to a steel manufacturer. Need to understand data bedore we can understand databases.

So, what can we say about data? First of all, terminology: data, information & knowledge are banterd about interchangeable, but actually have very precise meanings. Data is raw unstructured facts, numbers. Information is data that has been organized. Knowledge is information that has been applied. Eg. if I say that sales data is 87, 24, 36, 17, 55, mostly useless as is, but necessary to produce information. Eg information 24, 17, 36, 55, 87 is data that has been organized in chronological order, it is more useful. The knowlege is that sales decreased but then rebounded and have been increasing.

So data is:

formatted: text (characters), numbers; also tables, charts, images: i.e. information. Must be able to store format and organizational information with the data

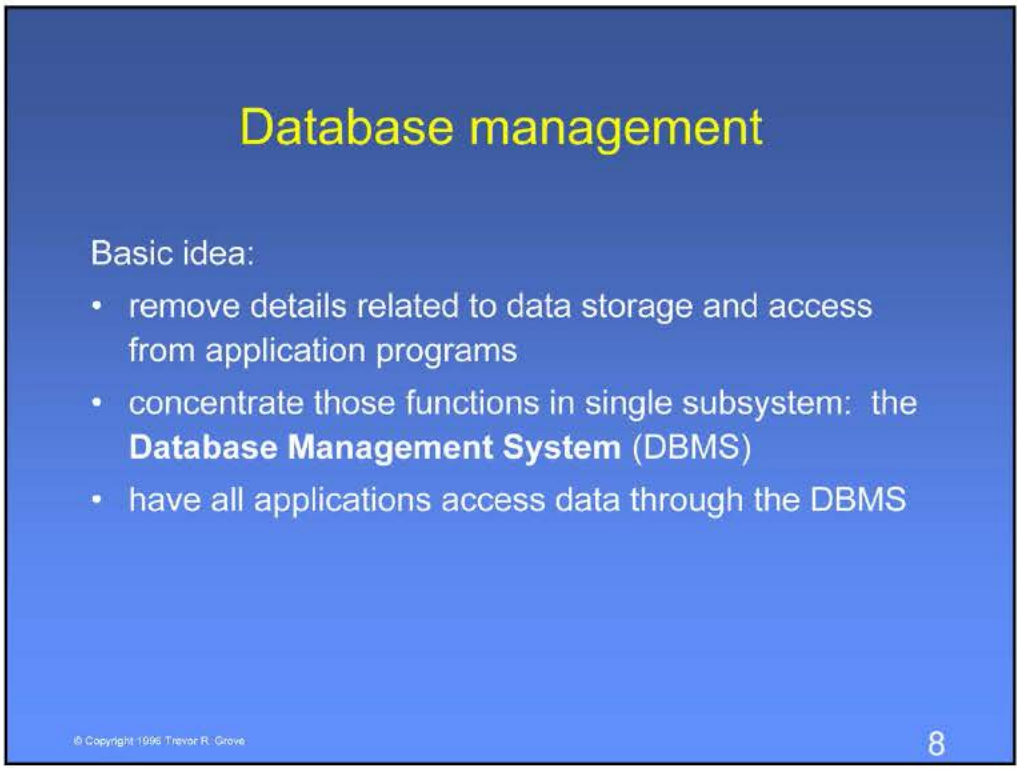
important: must be accurate (pointless if not); must be reliable (stored data must be guaranteed). must be able to get at it when we need it (i.e. online, not offline: fiche is a reliable storage medium, but not very useful for interactive use. writeable CDs are becoming interesting here)

large amounts: gigabytes and terabytes and ?. Even thought computers are good at searching, vast amoust of data means need strategies for getting at data efficiently[CocaCola ancedotal story))

simultaneous: online systems must handle simultaneous access. classic examples from banking, reservations result in overbooking, overdrawing etc.

diverse: different apps want access to the same data or portions. must be able to restrict access

Any individual application could provide these features,. but multiple applications means duplication means increased costs, decreased reliability, inconsistency. Woulf like to gather together the solutions and implementation of these principles into a single place, so individual apps don't need it. Call this idea data independence (apps are independent of data manipulation)



So, this idea of data independence is what motivates the development of independent database systems; brings us back to our discussion of DBMSs

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 DMBS. Treat DBMS as a black-box to the rest of the world.

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)

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 retreival 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 DMBS: can handle concurrency by serializing (only allowing one app access). performance: DBMS can guarantee certain levels of response to apps, facilitte priorities.

But: 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 in data descriptions)

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, thereare 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.)

- 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.

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 doesnt care where DMBS stores data, or how DBMS stores or accesses, or what other data/pgms DBMS is managing

...continued

Examples of changes in storage structures:

- data encoding
- record structure
- file structure

Data dependence is expensive because changes in the way data is stored or described requires changes in application programs.

Appropriate data independence can provide some degree of vendor independence for DBMS software and application development software, avoiding problems associated with proprietary solutions.

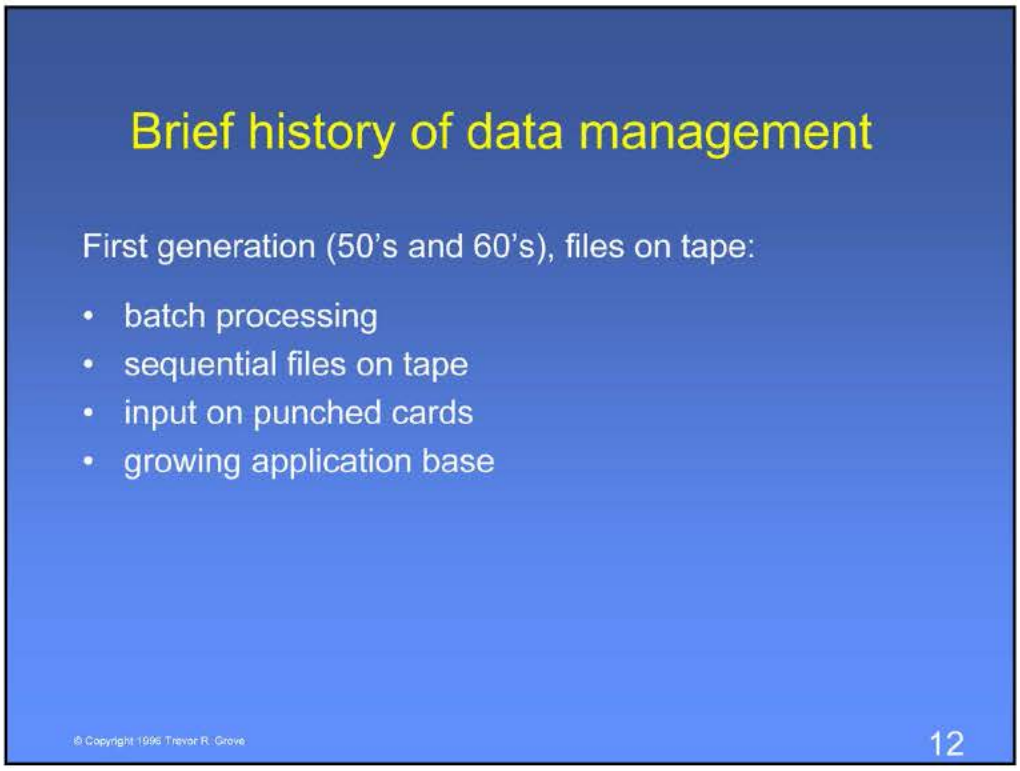
Some egs of things that we don't have to worry about if we have independence:

- data encoding eg character sets, numeric representation (eg big/little endian, fp)

- record structure: how data is organized into records and fields. we just say the name of the firld we want, don't care about anything else.

- file structure: how files are stored. sequential, random, keyed, indexed... up to DMBS to decide, maintain

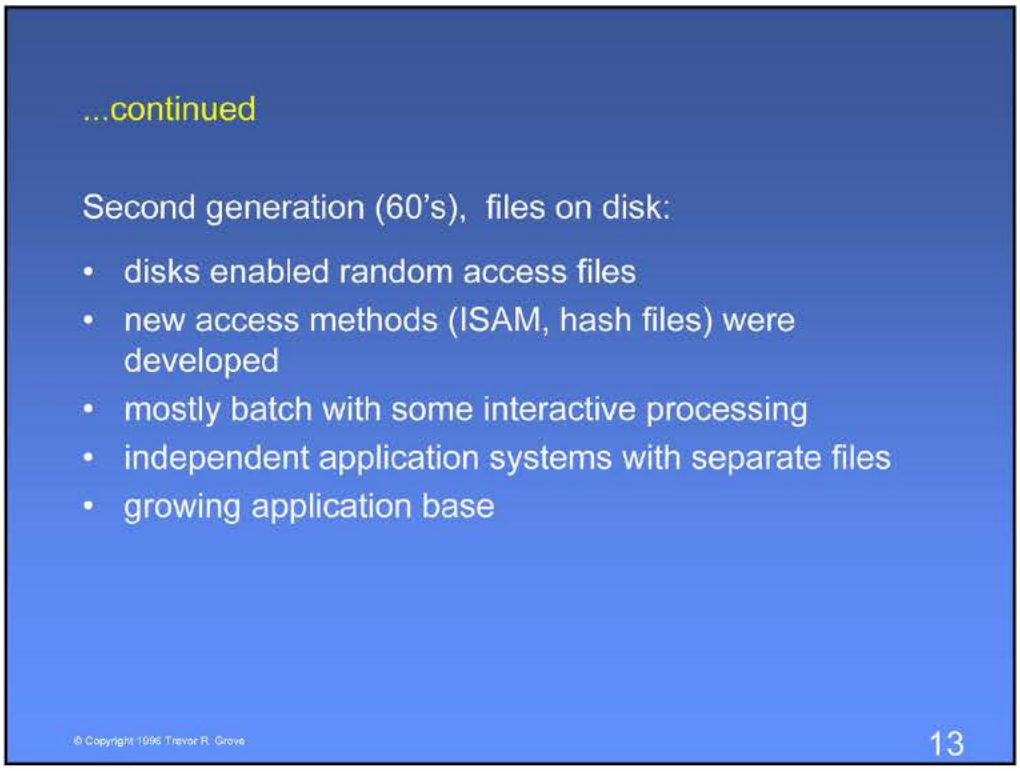
Last word (for now): without data independence, every time one of these items changes, every application program must change. in complex DP environments where data is shared among many apps, cost of pgm maintenance goes up.



As long as there has been data, it has been managed, sometime informally (or subconsciously)

In early systems, punch-card and tape based -- not often considered a DB, but certainly fits the general definition. Issues like security, concurrency non-existent. application base relatively small; maintenance due to reorganization not typically a problem

As computing capacity grows, application complexity and sophictication increases. Deployment of ramdon-access disks, permanently-mounted file-systems. Beginnings of recognition of need for separation of applications from underlying data. development of access methods to improve performance and increase capabilities (searching via index and hash files eg). with establishment of permament file systems and structures, application base grows faster. beginnigs of interactive systems requiring interactive response-times



As long as there has been data, it has been managed, sometime informally (or subconsciously)

In early systems, punch-card and tape based -- not often considered a DB, but certainly fits the general definition. Issues like security, concurrency non-existent. application base relatively small; maintenance due to reorganization not typically a problem

As computing capacity grows, application complexity and sophictication increases. Deployment of ramdon-access disks, permanently-mounted file-systems. Beginnings of recognition of need for separation of applications from underlying data. development of access methods to improve performance and increase capabilities (searching via index and hash files eg). with establishment of permament file systems and structures, application base grows faster. beginnigs of interactive systems requiring interactive response-times

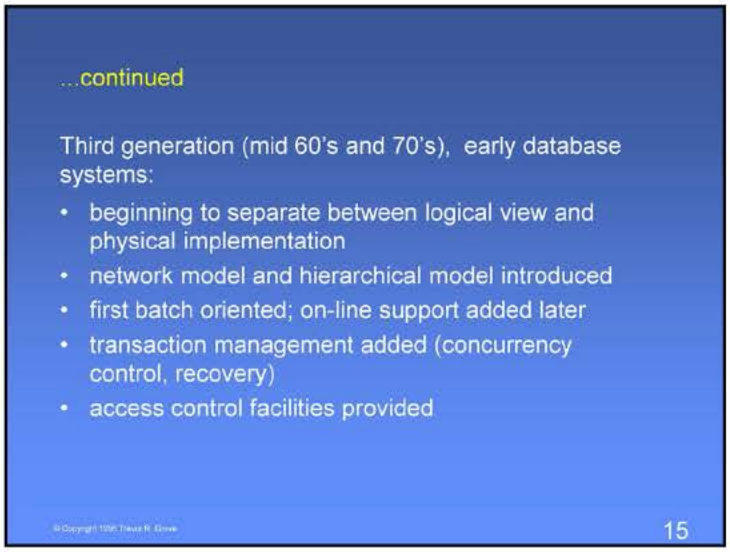
expliosive industry growth creates problems. applications reinvent the wheel over and over again, need to share of data (eg financial reports, statistical analyses of same data) , but hard to do.

Lots of problems: all the classics. each app would access and update data, cause consistency and accuracy problems. attempts to resolve these often used multiple copies of data, but then had to figure out how to reconcile these copies.

Data access techinques varied, complex to implement, harder to change.

Duplication makes it hard to keep track of data, leads to security concerns

Overall, expensiveand precarious



Problems with ad-hoc approaches recognized, formal DBMS developed to improve situation -- first attempts at data-independence

1968 ibm's IMS (Info mgmt sys): hierarchical data model, where data model is a hierarchy. information is viewed as being composed of smaller pieces. Eg: auto mfg composed of corporate, mfg and dealers; corporate composed of finance, engineering, marketing; dealers composed of regions; regions composed of dealers; dealers composed of sales dept, service dept; sales composed of individual salespeople.

To get from top to information about sales staff, traverse levels in hierarchy ==> need to know how to navigate around the hierarchy.

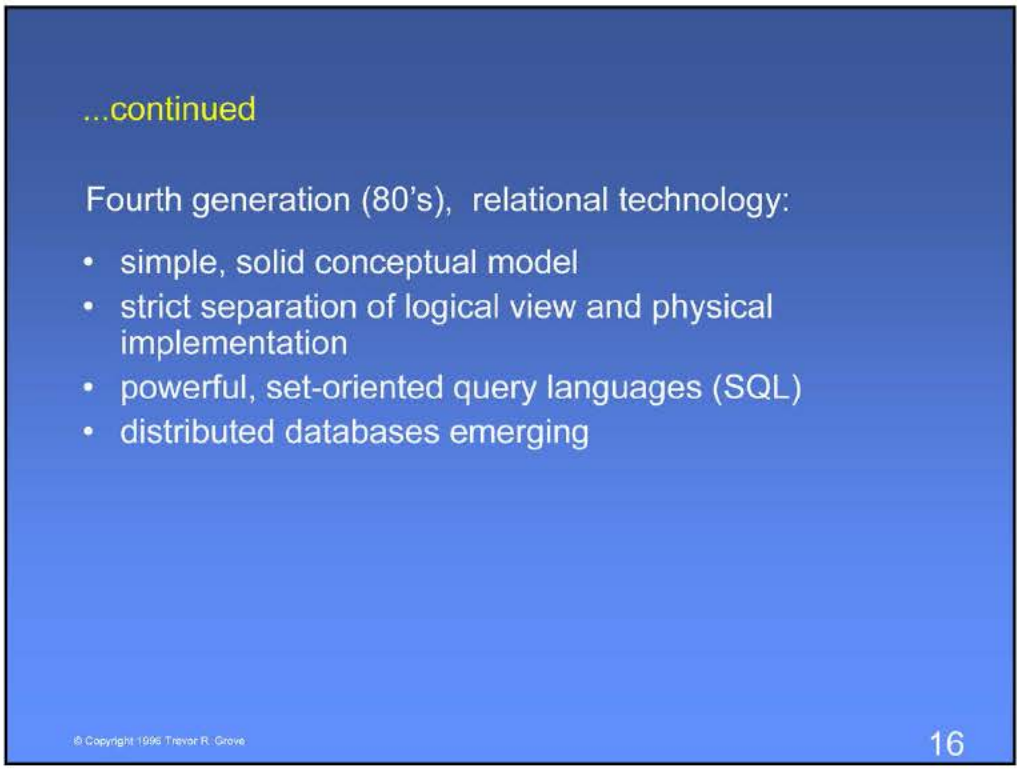
~1971 cullinet's IDMS: network data model, based on industry "codasyl" report. generalized version of hierarchical model. same top and bottom (leaf nodes), but more that one set of connections. a little better at some kinds of data structures.

[hierarchical: directed tree; network: acyclic directed graph]

In both cases, retrieving data (aka sending a query to the DMBS) queries really hard to create, structures hard to change, queries change if structure changes.

But: still in use [IMS especially].

Because these systems provided single interface point, addition of concurrency, integrity features possible. with transaction (sending a message and receiving a reply), DMBS could do transaction management to resolve these problems. Also provided ability to add security/access control.



4th: experience with hierarchical DBMS, esp with difficulty in forming queries to retrieve data, led to research by ibm into better ways to do queries. research prototype call SEQUEL created in mid-late-70.

SEQUEL based on a organizational model called relational. relational model has very strict mathematical basis (relational algebra [CODD 1970] and set theory)

SEQUEL product evolved into SQL, which is a language for describing and querying databases. SQL statements are completely independent of implementation of data: say what you want, let DMBS figure out how to get it. finally achieve true data independence.

Relational DB organization and SQL are focus of this course.

distributed db (where pieces of one DB are spread out onto different machines) emerge as computing hardware price/performance improves (workstations etc).

...continued

Fifth generation (90's), post-relational systems:

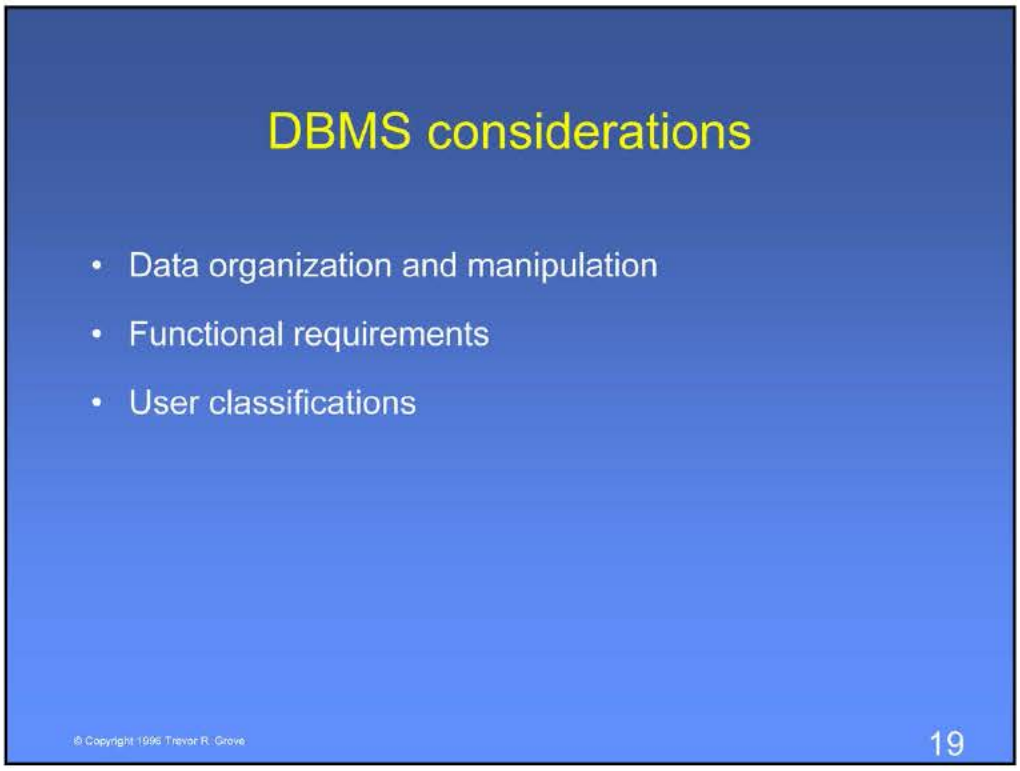
- added functionality, more complex data (temporal, spatial)
- serving a broader class of applications
- object-oriented systems
- multidatabase systems
- nomadic (remote) databases

5th: Current trends: DBMS everywhere, used for "non-traditional" data like time, space (eg adding "when" information to data; storing geographic information and retrieving base on "where").

Buzzword the rest.

Are there any obvious areas where databases are not useful?

No. Everything has data; the better organized and controlled the data, the more useful it will be and the more successful applications will be.



There are many features and operating characteristice that we want a DBMS to have.

1) remember that data independence is important (inprove reliability, consistency, protect investments). Need to organize data & develop applications to ensure independence.

2) Generally, want DBMS to look after data. what are some of the details of functional requirements of a DBMS

3) In the final analysis, a DBMS and its set of applications are tools that people use to get the job done. Varies from administrative/office support to executive to technical. Each of these has different requirements and methods of working

The "three-schema" architecture

schema (or scheme): description of data contents, structure, and possibly other aspects of a database

external schema (view): describes data as seen by an application program or by an end user.

conceptual schema: describes the base logical structure of all data.

internal schema: describes how the database is physically encoded, including selection of files, indexes, etc.

First thing a DBMS must do is let us create a description of data and information (data and its structure).

Remember the data independence goal. Following a standard technique in software, use several well-defined layers of definition to implement independence.

Typical layering consists of three layers. Called "schema".

1: has nothing to do with actual physical organization. says what we want in any particular circumstance.

2: full structure. external schema may be restricted by security, performance considerations etc. still has nothing to do with physical.

3: physical organization. has nothing to do with users or how data is presented, used. Nuts and bolts of file organization

These are defined to facilitate program-data independence.

...continued

- separation of external schema from conceptual schema enables logical data independence
- separation of conceptual schema from internal schema enables physical data independence
- database schema (intention) is different from database instance (extension)

external vs conceptual == logical independence. Eg we can add new items to conceptual schema without affecting existing aplications

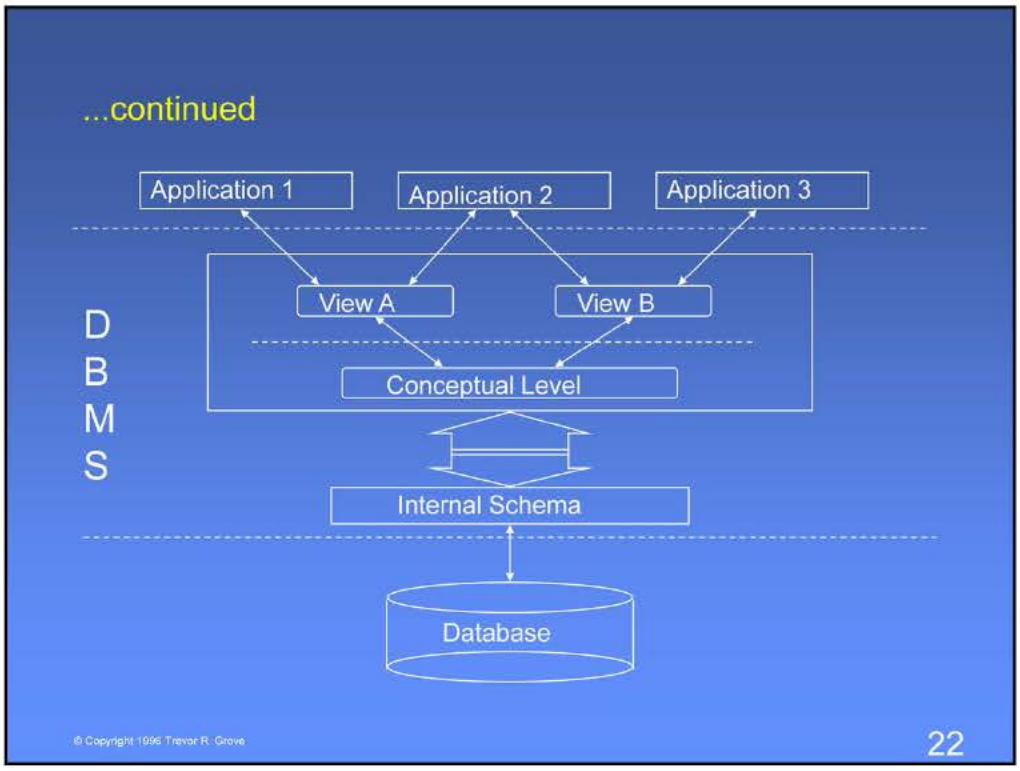
conceptual vs internal == physical independence. Eg create an index for better retrieval speed doesn't affect coceptual schema.

during discussion, may need to distinguish between schema and instance (design versus implementation)

(picture to follow)

Schemas are just descriptions, they can be applied to anything. Once schema can be applied to many different DBMSs. The three schema architecture can be applied to any DBMS, although it fits very nicely with relational model (coming up)

logical
data indep



external
schema

conceptual
schemal

internal
schema

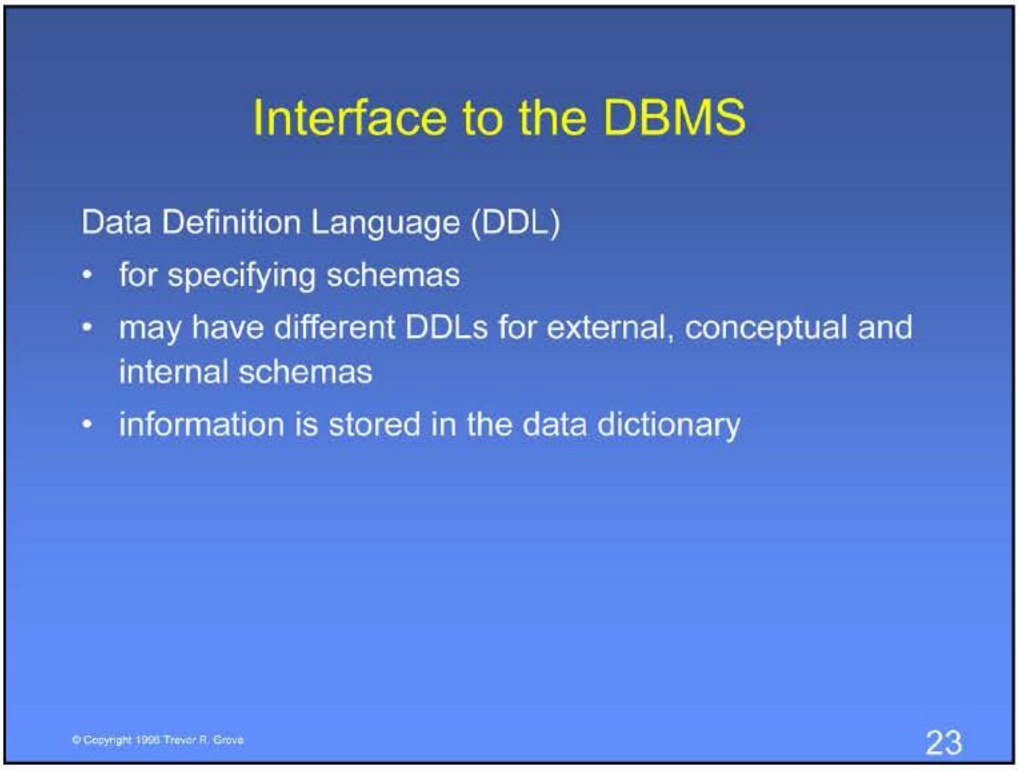
physical
data indep

a picture is worth 1000 words:

applications each get a view (ie an external schema), which is derived from the conceptual schema. internal schema says how to do physical organization on media

consider that the DBMS is everything between dotted lines

The database: not yet well defined, lots more to say. Informally, think of database as collection of data organized into tables (rows and columns) -- think of a spreadsheet. this tabular organization is a fundamental to the ideas of relational databases.



So we have the idea of three separate scehmas for a database. How do we write these down and get the DBMS to do something?

DBMS must provide a definition language, called a DDL. DDL says what the structure of the DB is, what the datatypes are etc.

Note that all this is, is a language, allows us to write down definitions and give them to the DBMS. Don't use programming-language-style constructs, because they tend to defeat independents. We'll clarify this in a minute.

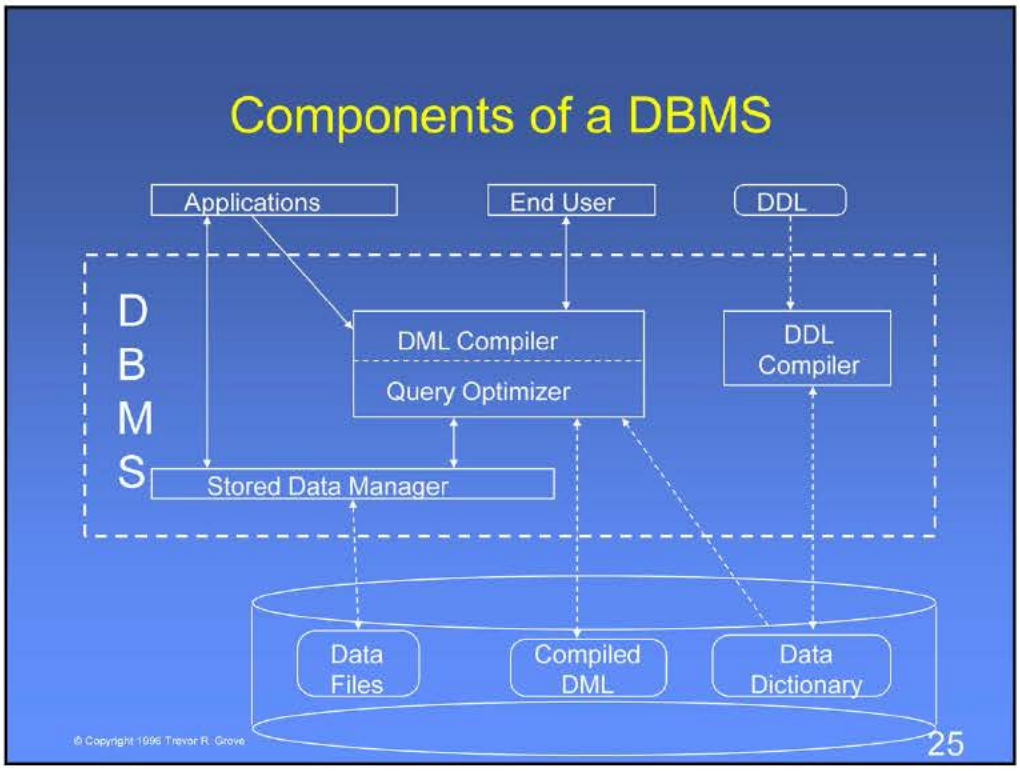
DBMS must provide data dictionary aka catalog to store DDL for a DB

On top of the DDL, have a DML -- data manipulation language -- for "doing stuff" to the database like adding, retreiving, modifying, deleting data; controlling the DMBS, establishing security, performance tuning.

DML used from application programs, or from "command interface" (type commands directly at DBMS

Two kinds of DML: user/programmer must know how to get to data (navigational), or non-navigational (say what you want). Think of hierarchical/network db.

Note: these things don't have to be fancy; a paper description of a DB meets the requirements for a DDL. We haven't said anything about any particular DBMS yet. DDL and DML don't imply relational.



another 1000-word picture.

DDL get created, fed into program that processes it and produces a description of the DB (or, what the DB would be if an instance were creat ed (instantiated)).

Users and pgms then use DML to do things. Will instansiate the DB from its schema, add some data (use a data manager to control disk/backing-store), do queries (might optimize them for speed).

DML can be processed on the fly, or processed into a stored form (typical separation is canned applications vs interactive ad-hoc queries)

Functional requirements for a DBMS

- provide data definition facilities:
  - define a data definition language (DDL)
  - provide a user-accessible catalog (data dictionary) (database should be self-describing)
- provide facilities for storing, retrieving and updating data:
  - define a data manipulation language (DML)
- support multiple views of data (user views):
  - end user or application should see only the data needed, and in form required

We've said how to describe the DB to get independence (three-schema architecture); we've defined the communication mechanisms with the DBMS (ie said that we will have separate DDL and DML). Try to outline some specifics of what the DBMS should (ie functional view).

Obviously, DBMS must allow DB to be defined, but esp want this definition to be accessible to users. Want DBMS structure to be definable with its own DDL.

Obviously, DMBS must store, aretrieve and modify data. But, remebmer external scehma: want to be able to present different views of data to different users. users should get subsets of data per request (eg only the columns fields)

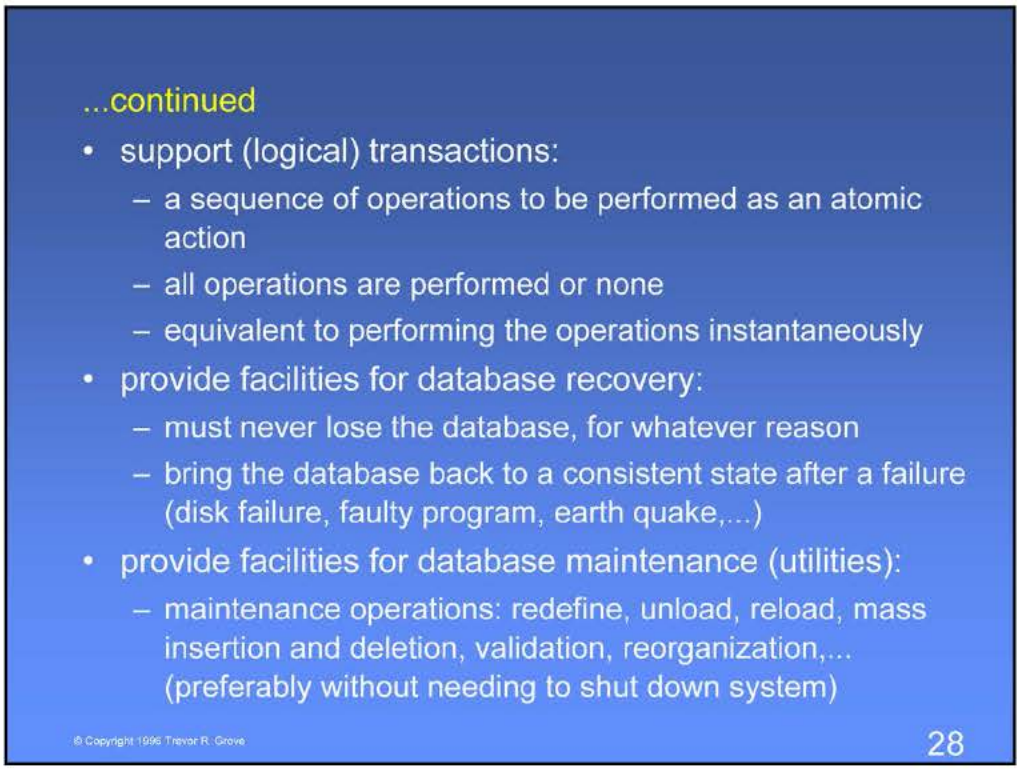
"Integrity constraints", "Referential integrity" -- fancy terms, just mean that we want to be able to put restrictions on the stuff in the rows and columns of the database. Eg might want to say that a certain field is supposed to contain numbers from 1..10: an integrity constraint is a DBMS feature that puts a "guard" on the field and notifies user if someone attempt to put a number outside the range into the field.

Referential integrity: says that a data value uesd in one place is defined elsewhere in another field. Don't worry about the terminology.

Could impose these restrictions in apps, but then every app woould have to do it. Use the DMBS to check data automatically whenever input or update occurs.

Clearly, DBMS is right place to implement security, cnetral (single) point of access. Administrators can impose restrictions on individual users or apps. Otherwise, hard to imagine: put security checking into every app? (too insecure) System service? (too much) Trusted users (login security)? (not always available)

Concurrency is even more compelling: individual apps/users ought not to even know about others.

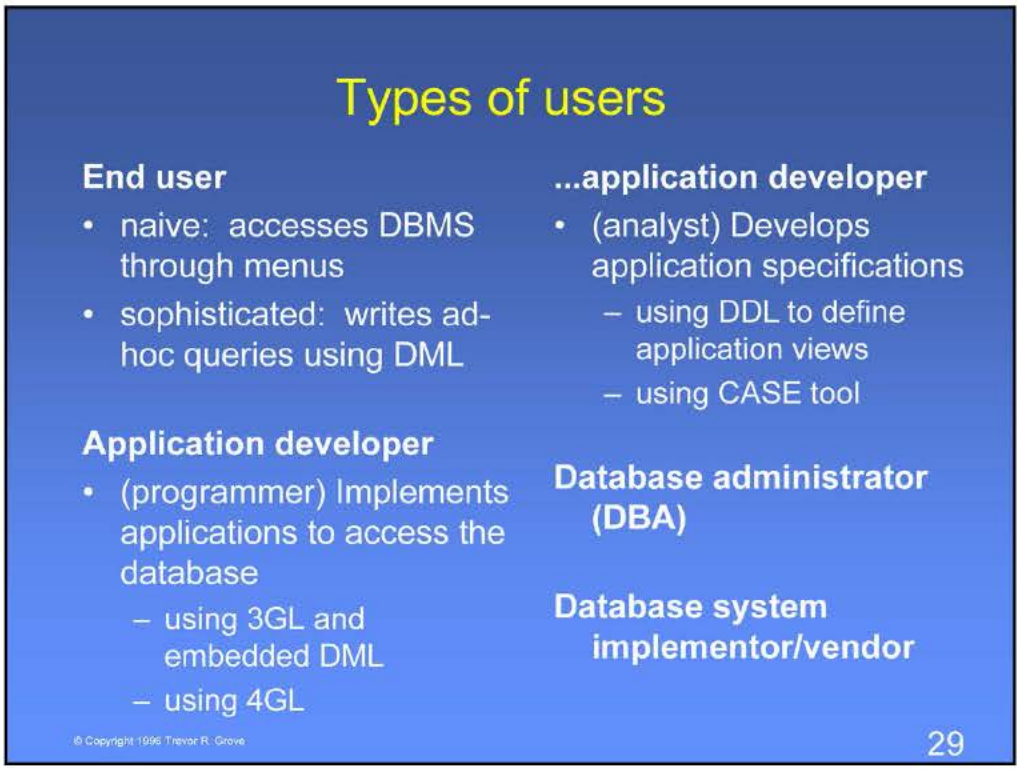


Other function requirements:

transactions: want to be able to gather together a bunch of operations into a single batch. batch can't be subdivided (atomic); either finished completely, or not at all (atomic) (no partially-finished batches)

db recovery: bottom line: must never "lose the big picture" -- might lose little bits, (eg today's transactions), but DBMS must provide checkpointing, logging, whatever, that lets us recover from any problems.

utility functions: all the mundane DB stuff that has to be there: bulk loading, data rollout & rollin, reorganizations. Want to be alble to do as much as possible with "live" systems (if the need arises).



Have been mentioning DBMS users informally. There are several distinct types whos roles should be clearly identified.

End-users: naive: use applications, typically menu-driven or otherwise isolate user from DB structure. requires no knowledge of DML.

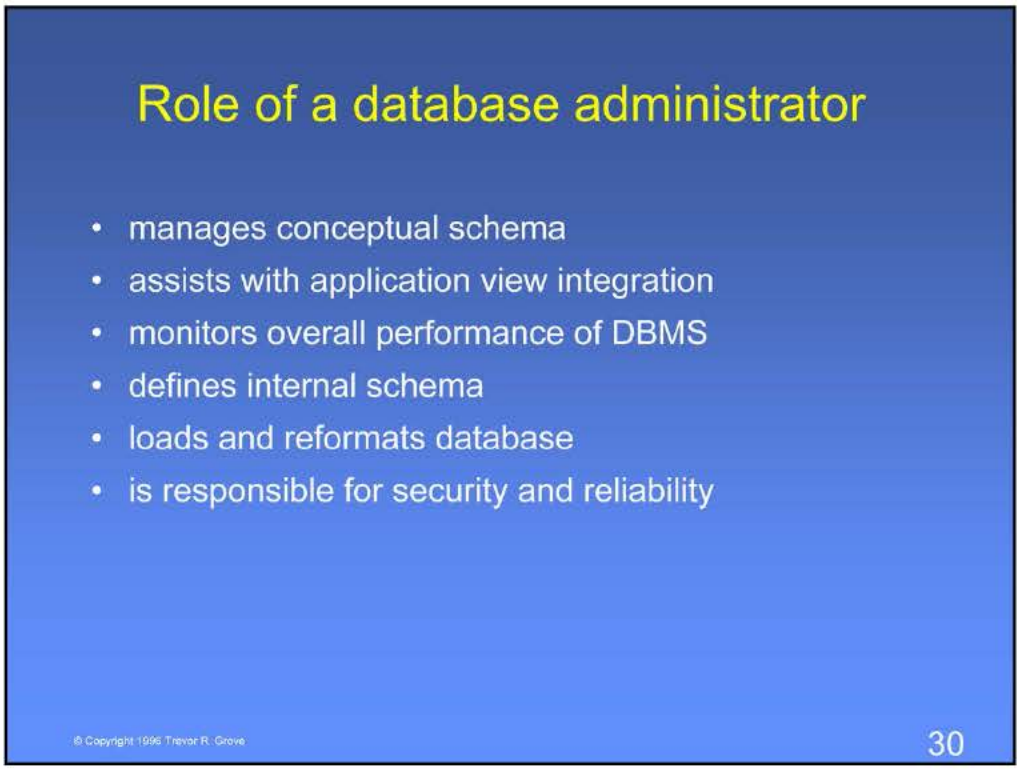
advanced: knowledge of DML, uses a DML-based tool to form queries and do other operations. perhaps book's use of "casual" is better than sophisticated. has knowledge of external and conceptual schemas

Developers: implementation of apps. various technologies; traditional programming lang (3GL) with DML embedded into programs (typically functions calls, more on this later), or 4GL app generator like powerbuilder, SQL-windows (ie the tool generates the DML from specifications)

analysis, use DDL to construct views for use by applications. CASE tools to help generate diagrams, test definitions etc. Creates external schema given conceptual schema

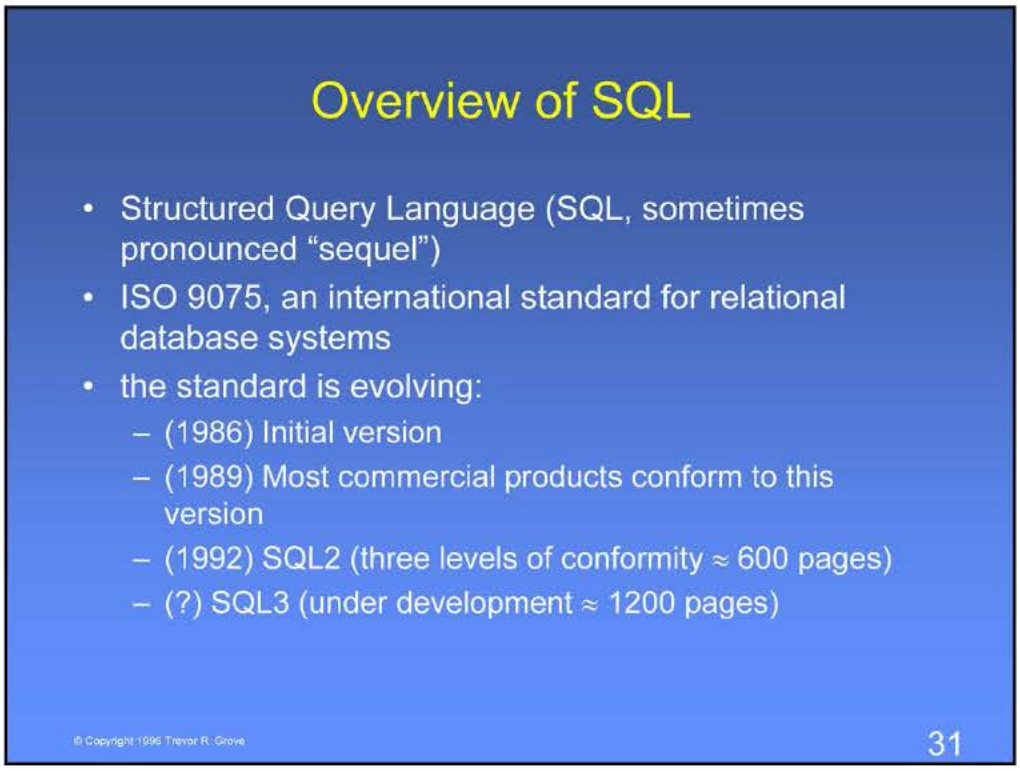
DBA -- next page

implementor - person who creates DBMS system; knows schemas for the DBMS (remember self-defining)



pretty must as stated.

analyses data, defines conceptual schema to meet varying (often conflicting) needs: storage efficiency, response time

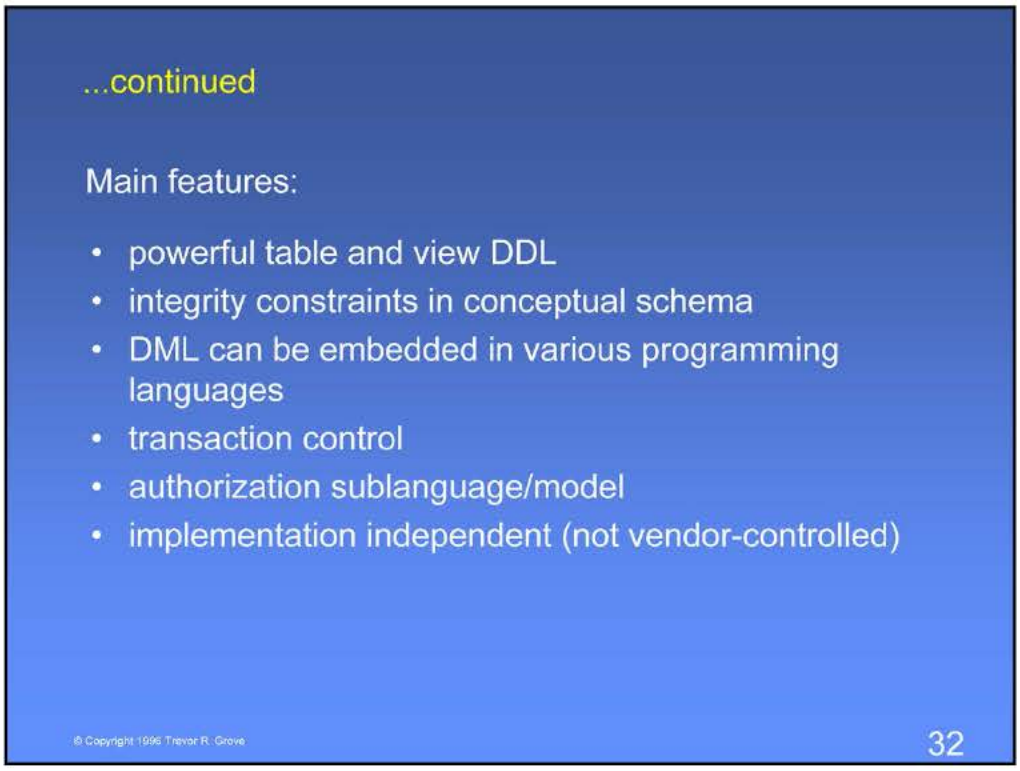


the book claims that SQL is not "Structured Query Language" and that "sequel" pronounciation is not correct ... oh well

Strictly: SQL is a language used to control and query databases. In 99.9% of cases it is used with relational databases, although this is not necesary (eg sql front-end for db3 on PCs ~1990) .

Mentioned earlier: origins mid-70s. based on mathematical system called relational algebra described 1970 or so by E.F.Codd. Purpose was to demostrate feasibility of this kind of language for DB interaction. IBM developed prototype systems using relational model, SQL evolved as the de-facto standard for query languages for relational DBs.

Since then, international standardization community has been busy...



SQL contains features to address most of the requirements we gave earlier:

1) it is a DDL and a DML. it is not a DBMS product

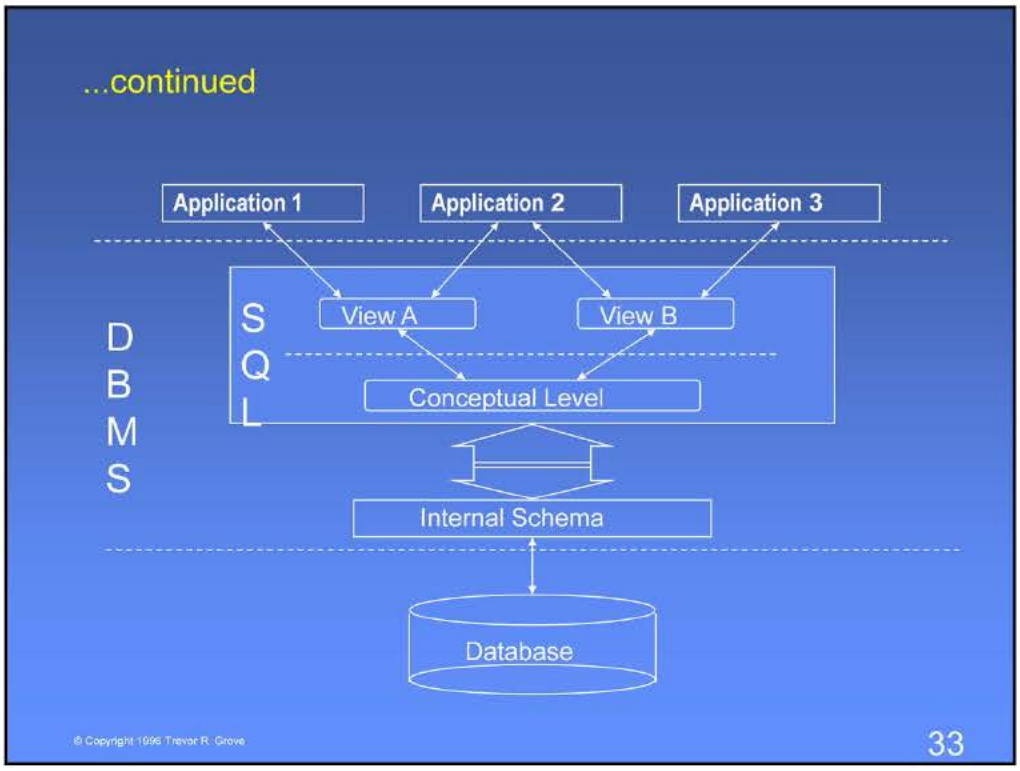
2) DDL part supportd integrity contraints (ability to restrict contents of fields in the DB)

3) DML is defined so that is can be used interactively by casual users, or embedded into programming languages

4) DML supports concept of complex transactions (batching together a sequence of commands into a single unit).

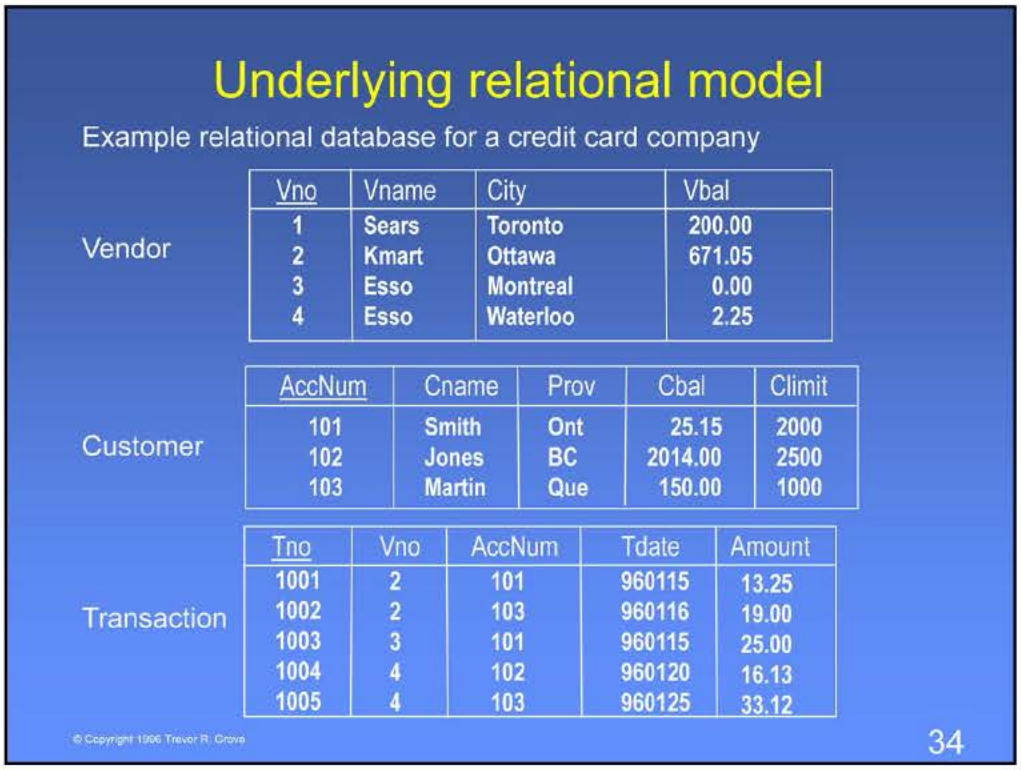
5) DML also provide a security model (userids, passwords, permissions on subsets of database (eg table by table))

6) SQL is independent of any particular vendor's implementation. there may be slight variabtions, but mostly standardized



similar to previous: top are users and apps, bottom is physical storage. SQL is the stuff in the middle, but does not include internal schema.

SQL is a language used to define and interact: it officially has nothing to to with the actual implementation (officially, but obviously relational DBs and SQL evolved together, so there has always been some influence).



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 distinguished 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]

Why is this a relational DB and not a set of spreadsheets? Nothing apparent from this eg, but tables are very carefully composed.

The SQL DDL

- used for defining tables (conceptual schema), views (external schema)
- example:

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


attribute domain

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 distiguish the row

"not null" unless otherwis specified, column values can be omitted (ie null), "not null" means cannot be omitted - common for primary keys

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).

The SQL DML

SQL has a *non-navigational* DML:

E.g. "Find names and provinces of customers who owe more than $1000 to the company."

```
select Cname, Prov
    from Customer
    where Cbal > 1000;
```

E.g. "List the names of the customers who live in Ontario and whose balance is over 80% of their balance limit."

```
select Cname
  from Customer
  where Prov = 'Ont' and
        Cbal > 0.8 * Climit;
```


Have looked at the DDL for defining database schema; turn our attention to the DML that we use to manipulate data in the DB (insert, retrieve, update).

First and formost, the SQL DML is not navigational -- to use the DML you don't have to know anything about the conceptual schema, deninitely not the internal schema. how tables are arranged and stored is irrelevant to DML users.

So eg, want to find ...; use the given DML statement "select". saywhat attributes we want, what relation contains the information, and any constraints on the attributes.

Note once again very programming-language-ish.

The result of this query is another table that has two columns (cname and prov) and as many rows as appropriate to meet the condition of the "where" clause

...continued

- Other SQL DML examples:

```
insert into Customer
  values (104, 'Trevor', 'ON',
          0, 4000);
delete from Customer
  where Cname = 'Smith';
delete from Transaction;
update Customer set Cbal = 0
  where AccNum = 102;
update Customer set Climit = Climit + 100;
```


Another example, showing more complex expression.

Thus far...

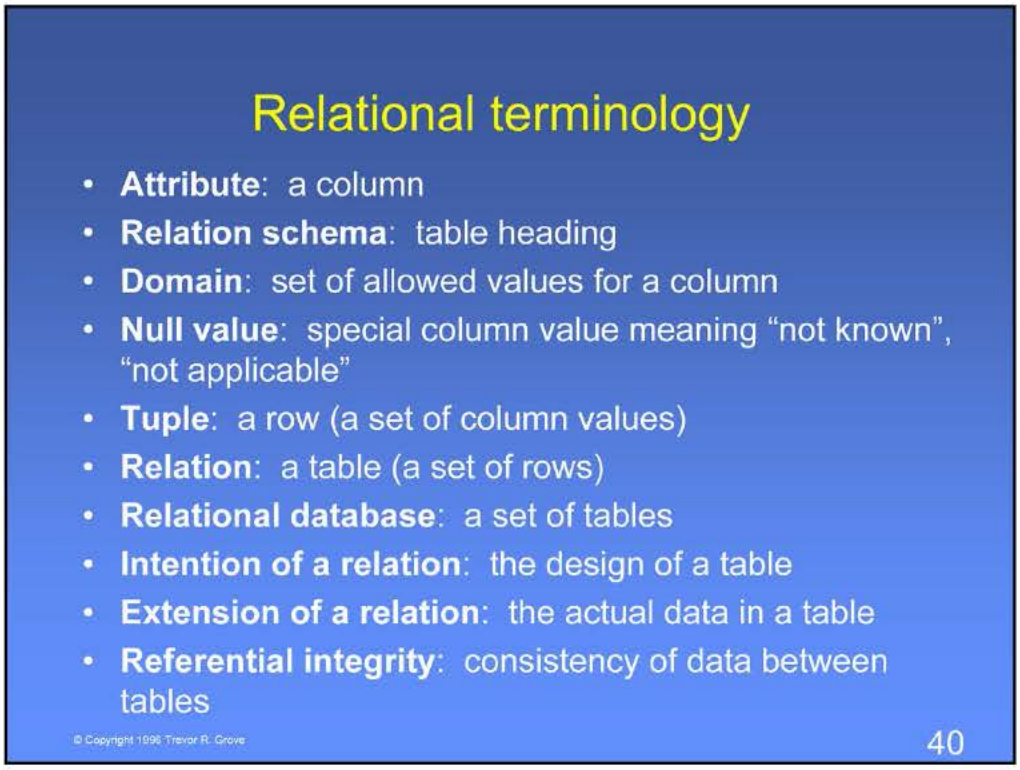
- Data is important
- Data is structured
- Data should be maintained independent of the applications that use it
- DBMSs are independent systems that “own” data and provide and control access
- SQL is an ISO-standard, vendor-independent means of defining (DDL) and manipulating (DML) data in a DBMS
- SQL was developed in conjunction with relational DBMS

Relational databases

- Basic concepts and operations of the relational model
- Using SQL to exploit the relational model

Warning:

Excessive mathematics and formalisms following!



Concepts are clear; formal study unfortunately requires lots of terminology (!, mathrmatical basis) So, formally:

- call a table a relation, it relates (connects) rows and columns. A database is therefore a collection of relations (hence relational database). Relation from mathematics -- connection between two entities.

- call a row a tuple, a bunch of column values. all tuples have the same number of values. Eg customer is cust#, name, location, balance, limit

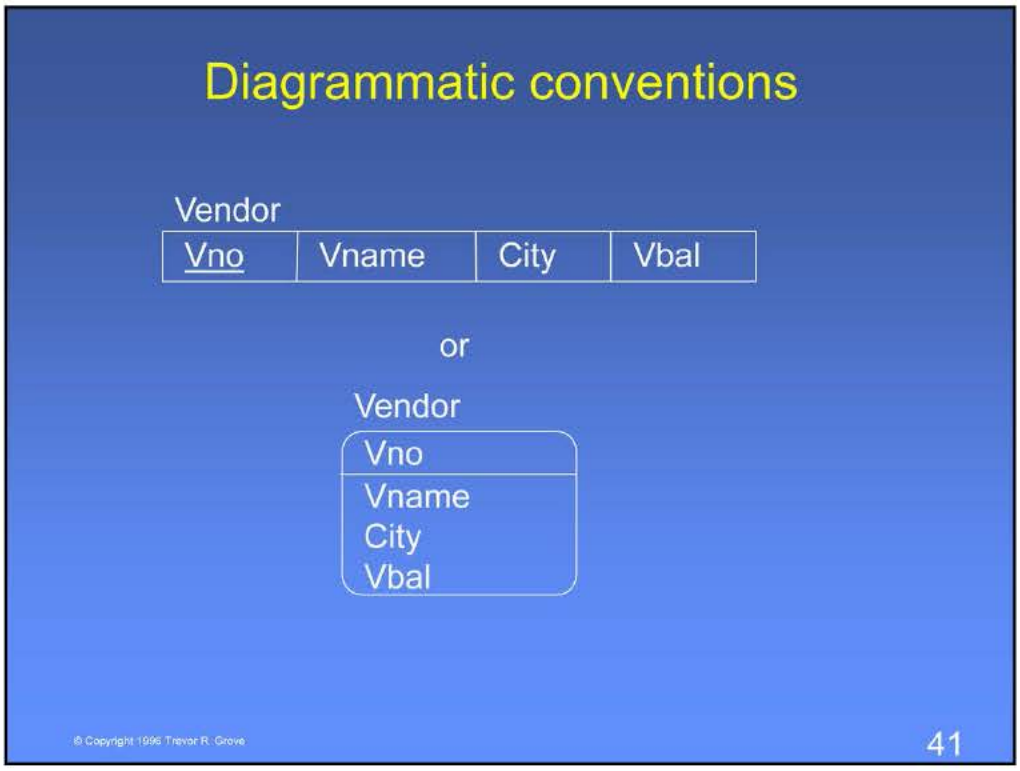
- column (ie each of the things in a row) is an attribute -- ie a property (so a row is a collection of attributes). a row (collection of attributes) is an element of the relation. Eg a row from the vendor table is a vendor; vendors have number, name, city balance. Can restate tuple as "collection of attribute values"

- columns (attributes), which are properties of members of tuples, are from s set of possible values. Eg a balance is a currency amount, which is a number >= 0 with 2 decimal places. A name is any character string up to (say) 25 characters in length. The set of possible values for an attribute is called the domain of the attribute.

Important: attribute values must be single values, can't have more than one. Eg in a relation (table) called "person", might be tempted to have an attribute (column) called "children", which would contain zero or more children's names. No good -- that would be more than one thing. have to do it another way (later)

Intention: the “shape” of a database

Extension: the actual data; an intention that has been “filled up”

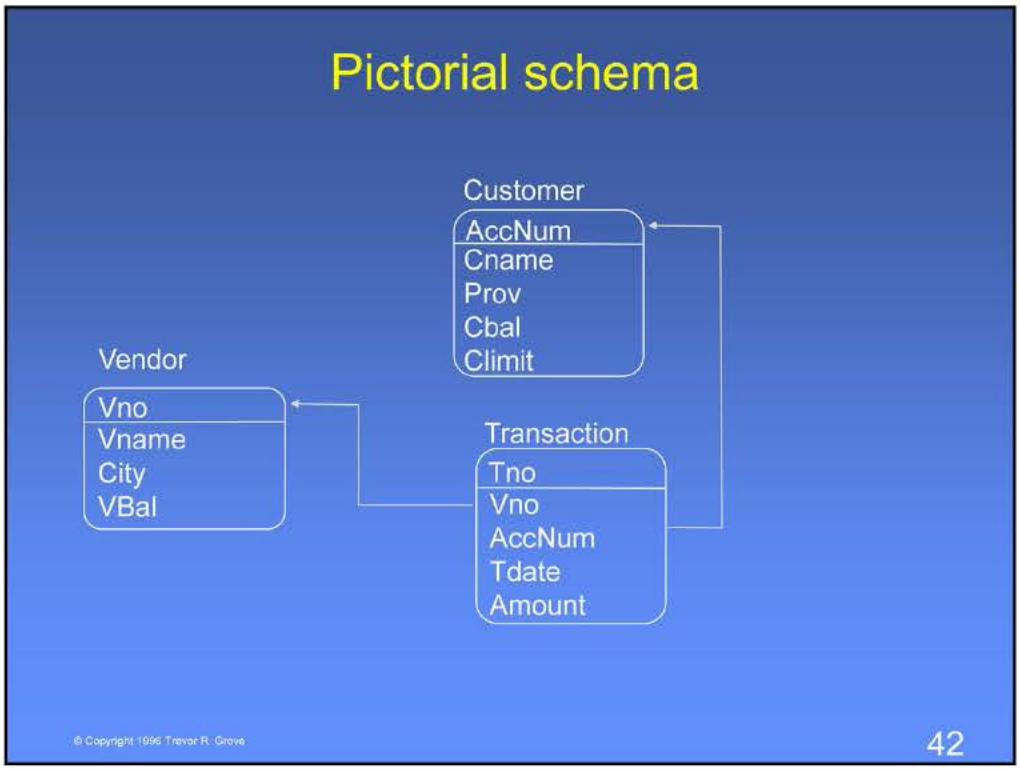


Conventional to use diagrams to described databases. Preceding table diagrams are handy:

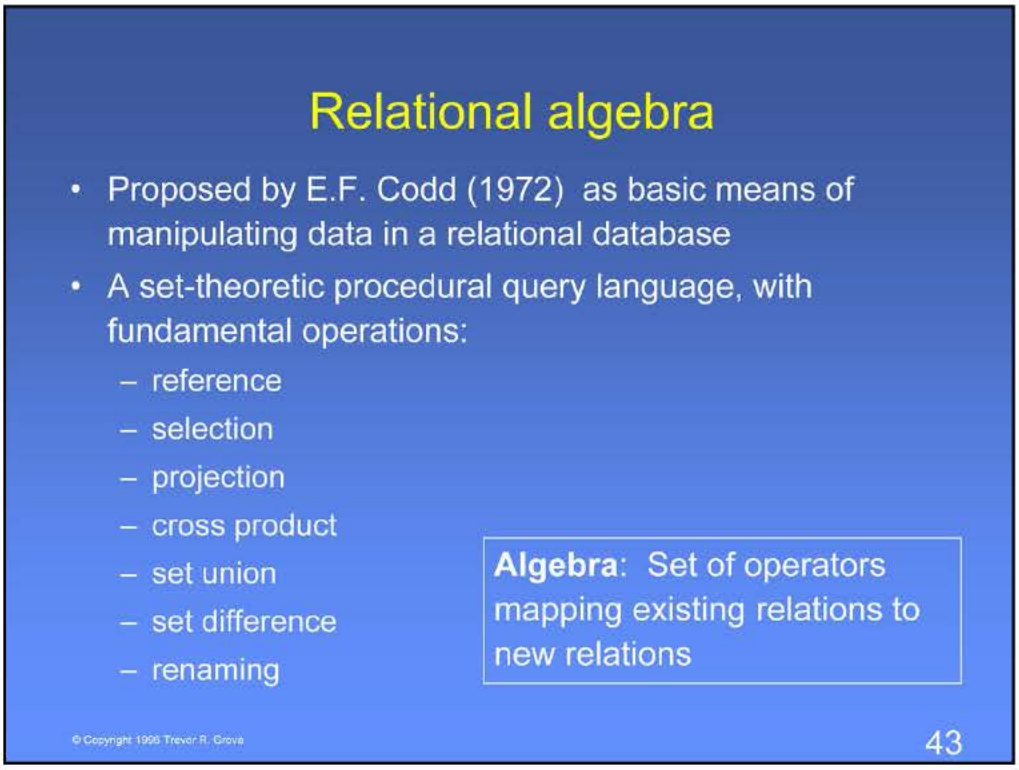
table name, attribute names, underline the attribute(s) (one or more) that is guaranteed to be unique for the tuples.

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.



arrows point towards relation (table) where the attribute domain is (column values are) defined"



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.

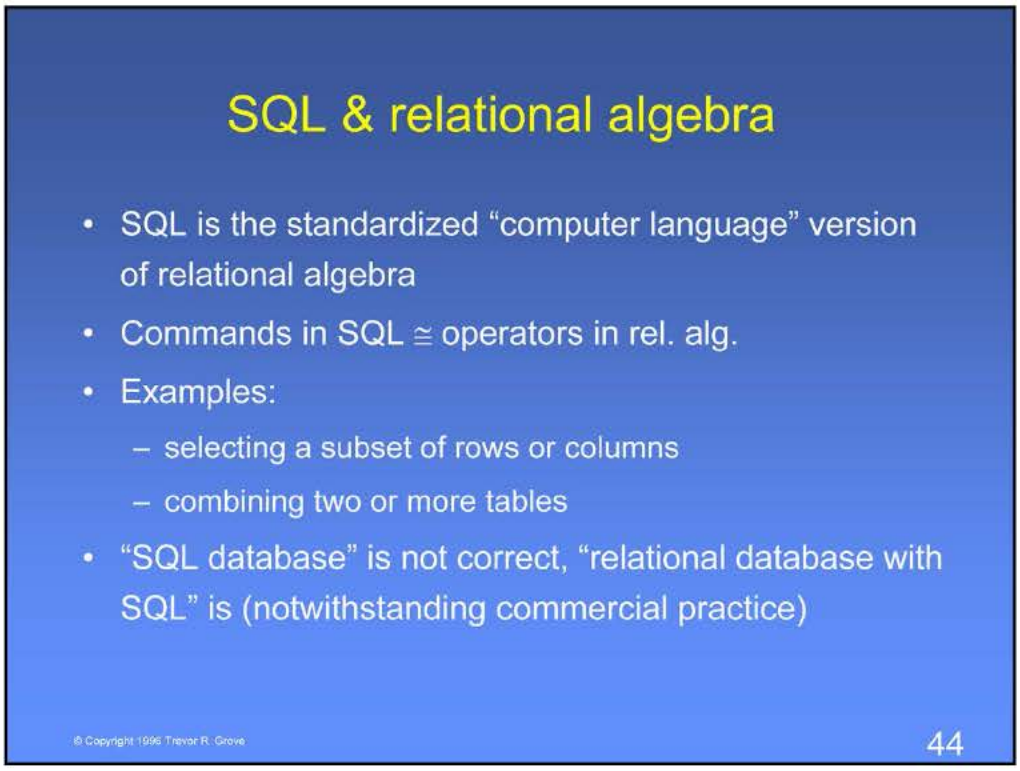
A relation is a connection between attributes and values, viewed as a table.

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.



How does all this stuff fit together?

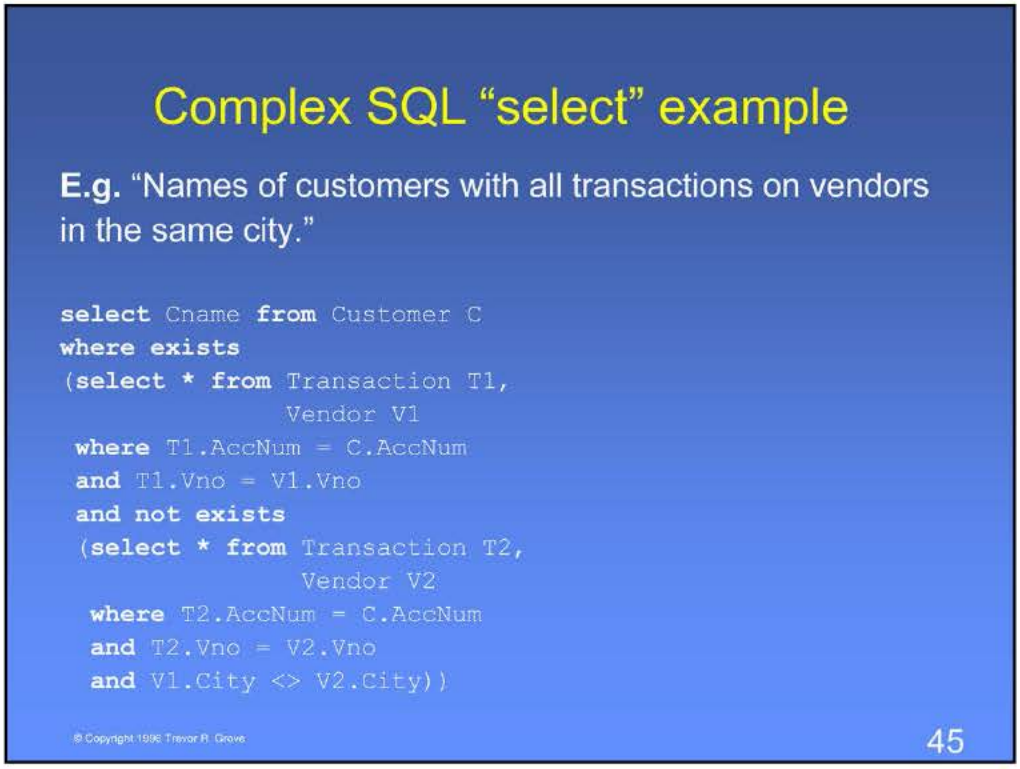
- DBMSs look after data
- relational databases are DBMSs based upon set-theory (relations)
- relational algegra is mathematical system for working on relations
- SQL is a popular commercially-available computer implemetation of relational algebra

Commercial practice is to refer to an "SQL database", although that's not strictly correct. There have been many expreimental relational databases that don't use SQL Eg IBM's QBE; ? Quel

The formal definition of SQL is quite complex. We looked very briefly at stuff last time. There isn't much to it except for the select statement -- the thing that we use for querying. It's as complex as many entire programming languages. It is quite powerful.

It is definitely requires skill and experience to formulate queries with SQL. Is is necessary to know all about relational algebra etc to be effective? Maybe not, but certainly helps to have an understanding of the underlying principles.

Following is a non-trivial example just to give a feeling for the kinds of things that can be done.



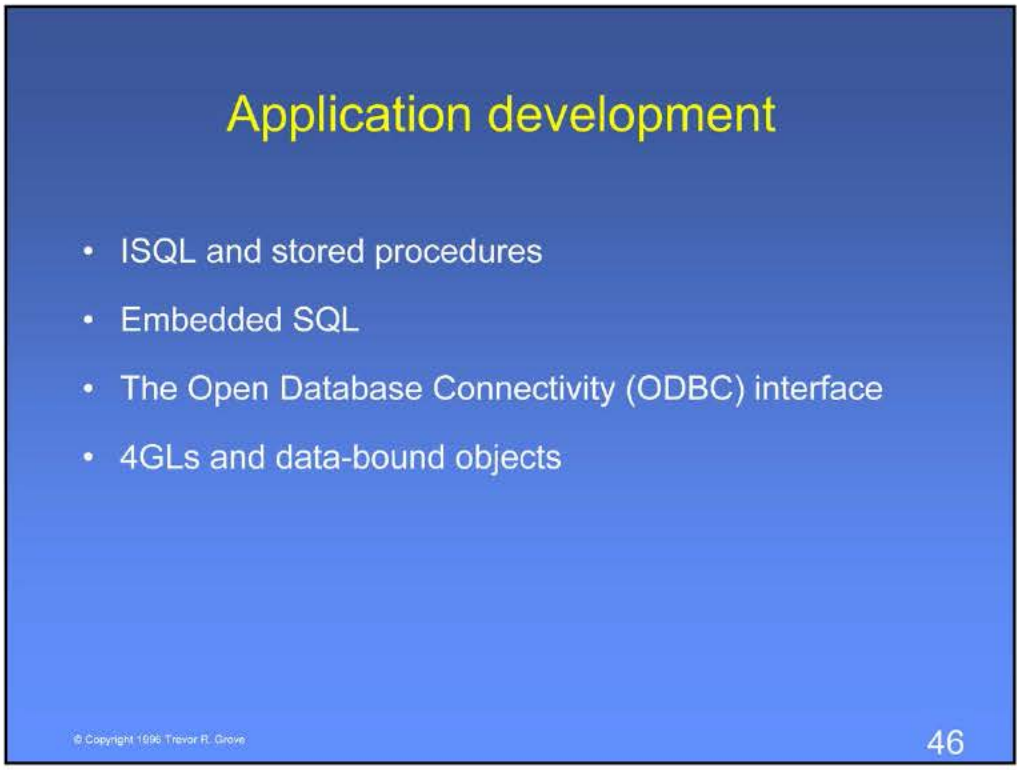
Eg: "names of customers all of whose transactions are from vendors in the same city"

first subq picks all the transactions for a given customer, then traverses that candidate set

second subq picks all the transactions whose city isn't the same as the candidate in the first subq.

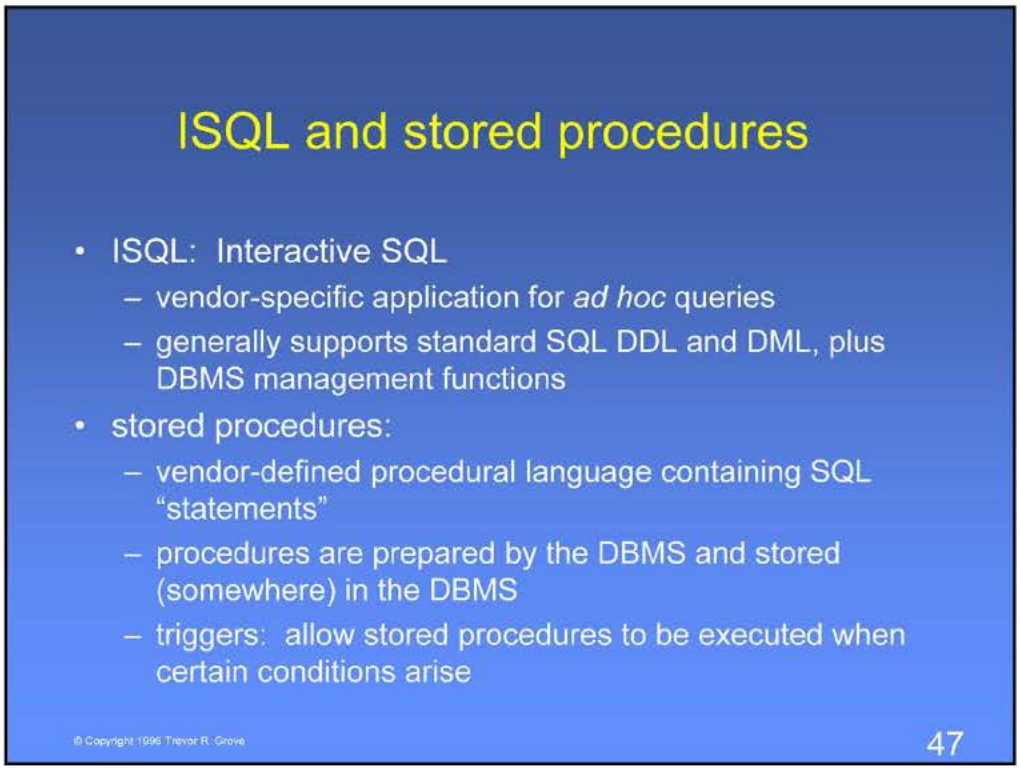
not exists second subq is true only if there are no transactions in a different city (ie city is the same)

exists first subq says that there is at least one customer candidate's transactions are all in the same city.



Saw that SQL seems to be quite powerful, but never good enough for all situations. Ad-hoc querying is useful, but certainly not appropriate for naive users. Need to be able to create “canned” applications.

Take a quick look at several different techniques.



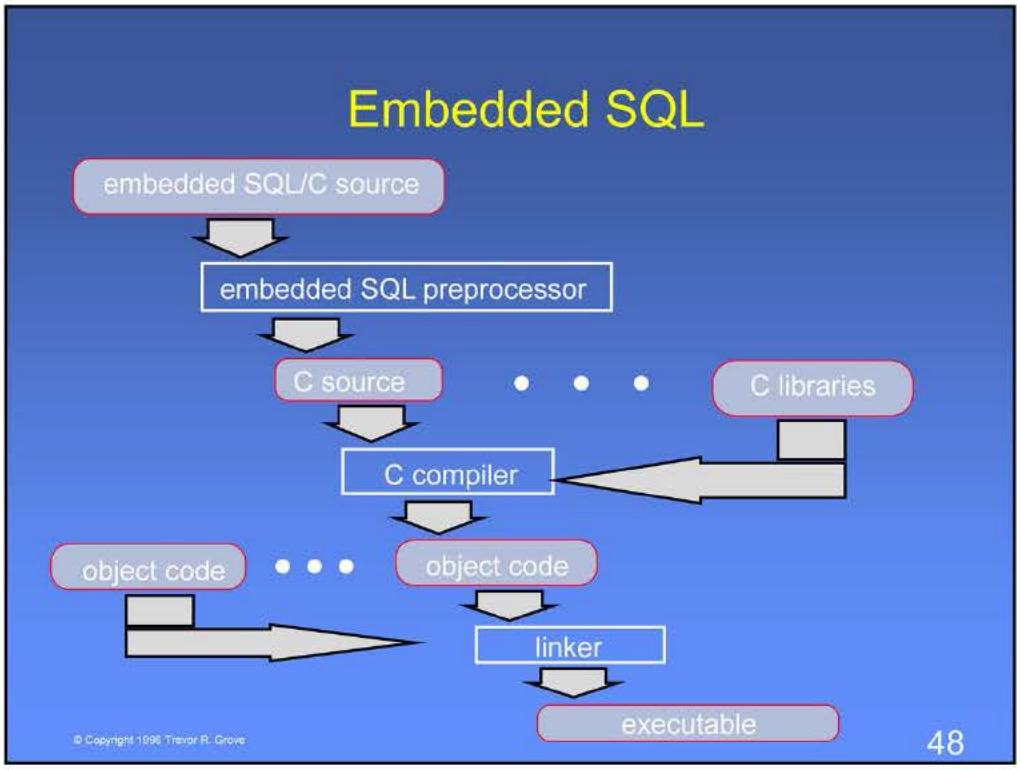
Every DBMS vendor provides some kind of ISQL tool to do DBMS management functions and allow ad hoc querying. Not really an application development tool, but certainly valuable for prototyping, testing qqueries. Can be used by advanced users as a reporting tool

Each ISQL tool defines its own dialect of SQL. Core standard is defined, plus typically lots of extensions. Definitely not portable -- very vendor-specific.

Stored procedures: some vendors define a programming language that can be used to write DB programs for queries, updates etc. Utterly vendor-specific (not all vendors have such a feature); programming language a mish-mash of current 3rd-generation programming languages (C pascal basic etc).

Procedures are stored in the DB (secret place), where they can be invoked by ISQL (or other application-development technologies).

Interesting spin on stored procedures is triggers. Can place instructions into the DBMS that will cause a procedure to be executed under certain conditions. Eg if a credit balance exceeds max, take some action.



This is the real stuff.

General idea is to merge SQL with a traditional procedural programming language (host language). Slide references C, although equally possible to use PL/1, Pascal, COBOL, Ada, REXX.

Can view this merge as an extension to SQL or an extension to C. It's really both and neither.

- Design a language extension closely resembling the SQL we have seen. Use this for database operations; "embed" the SQL into the host language program. Called ESQL

- Code the embedded SQL language along-side existing language constructs. use the host language to express procedural constructs like loops, decision paths

Take the hybrid program, process by an SQL preprocessor which converts the embedded SQL into ordinary host language (taking care of many details, probably uses function calls). Then proceed as usual.

- start with C augmented with embedded SQL

- process with SQL preprocessor, produce pure C code. C preprocessor is generally provided with database product, not C product. It's important that the preprocessor generate C code that is compilable by the desired C compiler.

The pp generally creates a distinct source-file that contains the C code. Can look at it if we want. Usually it gets discarded after use.

- C compiler compiles C code. Other C code (.h files for libraries) will be required. Compiler produces object code.

- Linker combines all object (our SQL, other application code, run-time libraries etc) to produce an executable module.

Why do all this?

1) Want application programmers to be able to use common DML regardless of development language. Same DML as interactive usage, so really convenient to do interactive testing, then immediate translation into ESQL

2) Typically really gorey code. Lots of obscure functions with lots of parameters; detailed data-structures. Preprocessor is responsible for subtleties of each app-dev language.

3) Provides independence of changes to interface definitions. the DML isn't likely to change (in a non-upwards-compatible way), but interfaces to given languges may change.

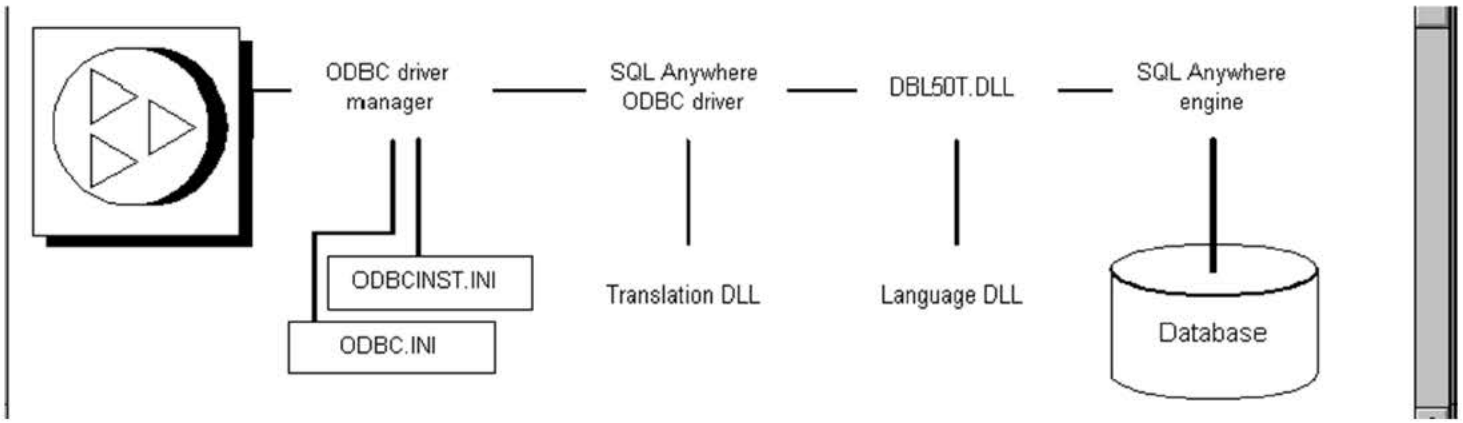
The Open Database Connectivity (ODBC) interface

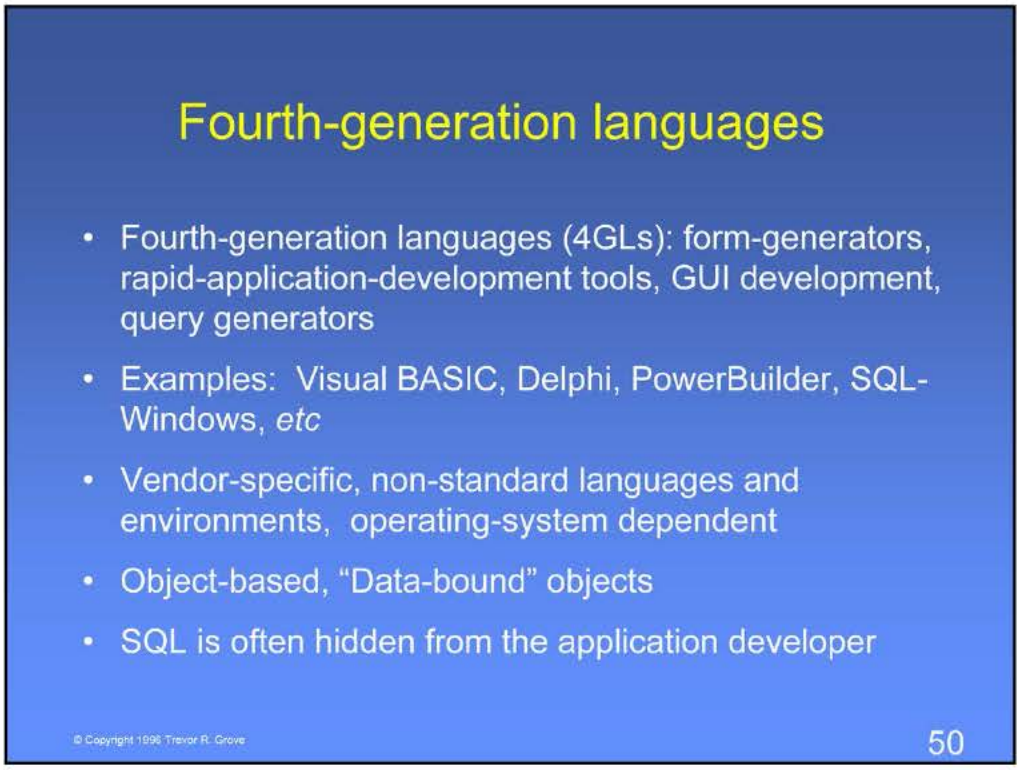
- Defined by Microsoft Corporation as a standard interface to database management systems
- Defined with a set of function calls, called ODBC Application Programming Interface (API)
- The API can be rendered in many languages; variations by vendor, language

Embedded SQL in traditional programming languages is one methodology for developing DB applications.

Another methodology is based on a standard called ODBC. It is similar to ESQL in the sense that it is a modification/enhancement to an existing programming-language. It's implementation philosophy is quite different, though. Instead of creating a hybrid language with its own syntax etc, the ODBC standard defined a set of function/subroutine calls that can be rendered directly in a host language.

Like any industry standard, there are variations and levels. Since the API is packaged as a programming-language library, it must be developed to conform to a specific language implementation. As well, there are potential variations in DBMSs. Generally, must ensure compatibility between language, API and DBMS. Most implementation do this with a multi-layer structure that contains separate layers for ,op-sys dependencies and DBMS dependencies





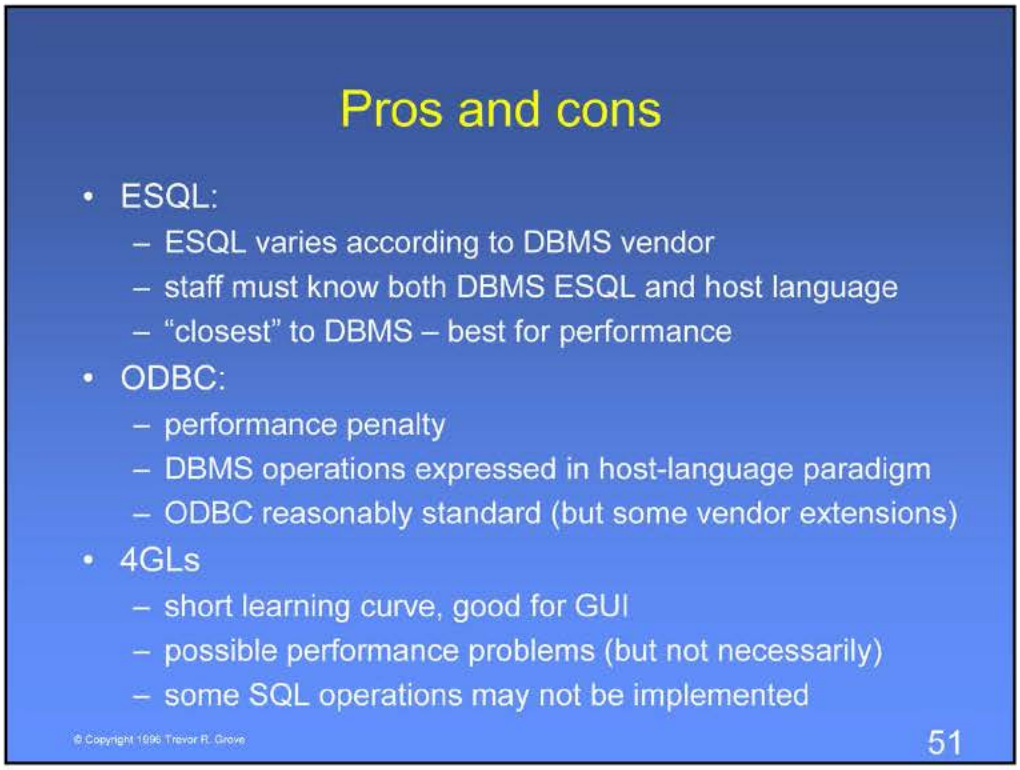
Have seen two development methodologies, both involved C programming. To a greater or lesser degree, both required knowledge of special function libraries, datatypes etc.

Another, considerably different, methodology involves what have become known as 4GL etc. At first glance, tools appear to be radically different, but they all have similar goals: improve productivity of applicaton developers.

Most of these tools have common features, in particular a graphical interface that supports interactive form-painting integrated with a database browser that lets the developer navigate a DB schema.

Many of them use an object-based programming model. In particular, they have a high-level "database object" which hides most of the details of how to do queries. Often referred to as "data-bound" objects, because they provide a connection between ordinary GUI elements such as text-displays, and database fields. The key idea behind a data-bound object is that once the connection is defined, updates, refreshes, commits, are automated.

This approach simplifies life for the app. dev, but invariables hides many details. Not sure if this is a good thing or a bad thing.



How do these methods compare?

ESQL is not portable - apps written to one vendor will almost certainly need rewrites to move. However, this does mean that can wring the most out of the DB by using native features. Training is more expensive, because staff have to know the host language and the ESQL language as defined by the particular vendor.

ODBC is more portable, although some vendors have implemented extensions to get around performance problems. ODBC itself defines conformance levels. There is guaranteed to be some kind of performance penalty over ESQL, since ODBC is implemented as a function layer. Smaller learning curve, since ODBC is presented in same syntax as host-lang (have to understand basic DB operation, but noting as detailed as an ESQL definition).

4GL hides significant details about underlying DBMS (buries details in objects). Should be able to swap DBMSs transparently. Most portable, but potential to prevent some operations (overly abstracted, least common denominator). Performance penalties not absolutely necessary, but probably. Learning curve is typically shorter, most tools have built-in wizards and tools to expedite development. Tools typically oriented towards GUI development.

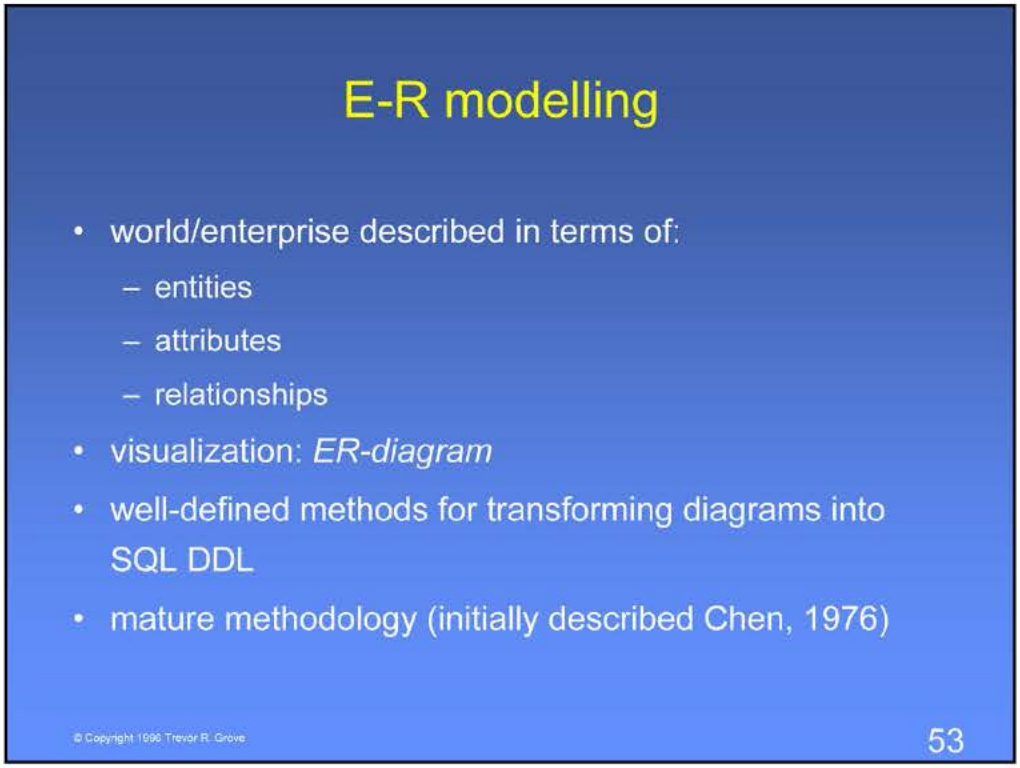
Data modelling

- Creating the database conceptual schema
- Entity-Relation modelling
- Normalization

We have spent considerable time looking at DBs, manipulating, implementing using DDL. We have always had the DB design (table definitions) given.

Now turn attention to designing a DB schema. Given a bunch of data, how do we decide what tables to have? What attributes?

We will touch on two different methods: the Entity-Relationship modelling technique, and normalization technique. At first glance, these methods seem quite different, but not surprisingly, they create similar DB schemas for a given situation.



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.

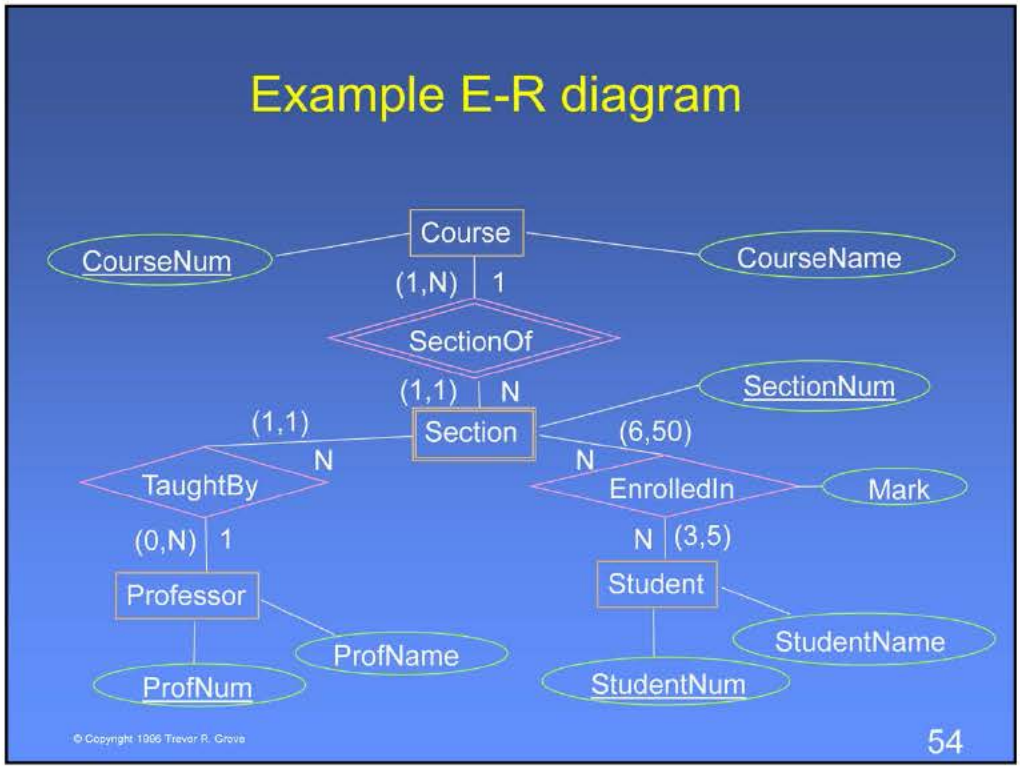
At the core of ER modelling is the concept of viewing the world as entities -- things, objects, identifiable, distinguishable. Eg customer, car, bank account, course, classroom, instructor. In all cases, we can somehow tell entities apart.

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 dedefine 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.



A big example with lots of stuff.

- orange boxes: entities
- green ovals: attributes/properties of entities
- pink diamonds: relationships between entities
- numbers: constraints on numbers

ER modelling is like programming in the sense that it requires expertise and experience. As a visual method is has advantages over methods like written descriptions.

Beyond the design phase, ER diagrams are useful for documenting systems.

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

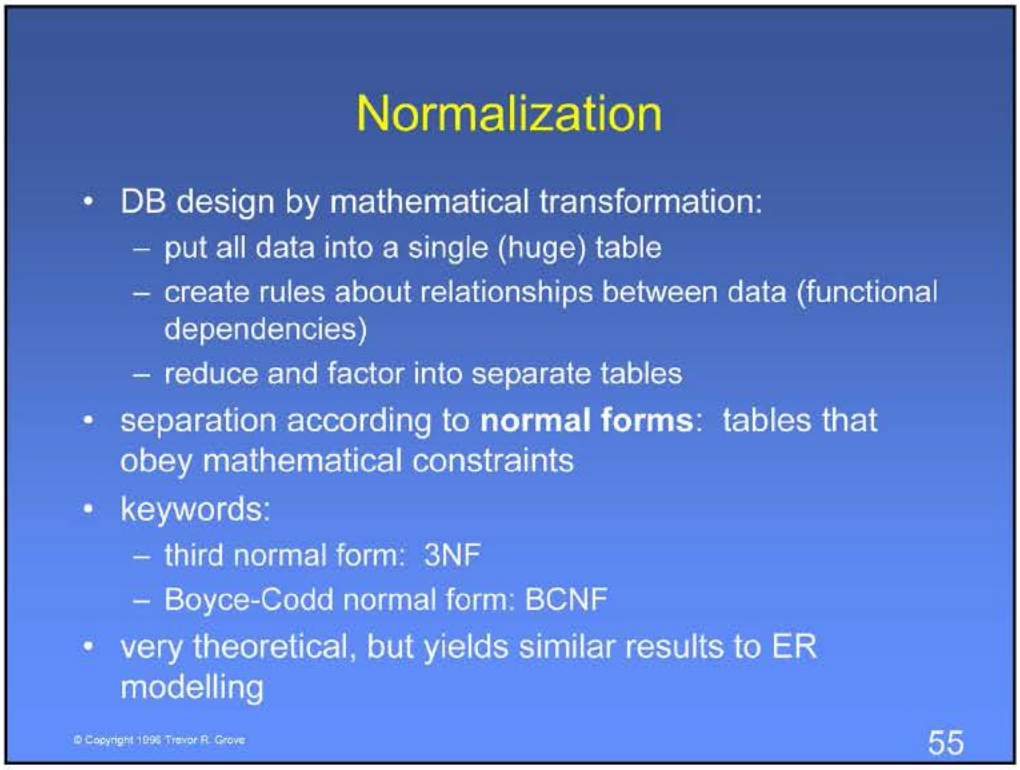
a course can have min 1, max N sections; a given section can be a section of only a single course

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



Normalization is an entirely different approach to DB design.

Here, we put all the data in an enterprise into a single table (universal table). This table will have significant redundancy and other problems.

Along with the tables we write down a set of rules that describes how the data in the table is related (functional dependencies)

Using mathematical transformations (normalization), split the universal table into lots of smaller ones that have fewer problems.

These transformations are done according to rules that preserve the relationships. It turns out that there are standard forms that these decomposed tables follow, called normal forms. 3NF and BCNF are the two most common normal forms.

Normalization is a highly mathematical methodology that is probably most useful to adademics, less practical. However, both methods tend to produce tables definitions that are remarkable similar. Knowledge of each is useful. [eg determining FDs is the similar to defining relationships).

Comparison: ER requires intuition, Normalization is mechanical (once FDs are discovered)

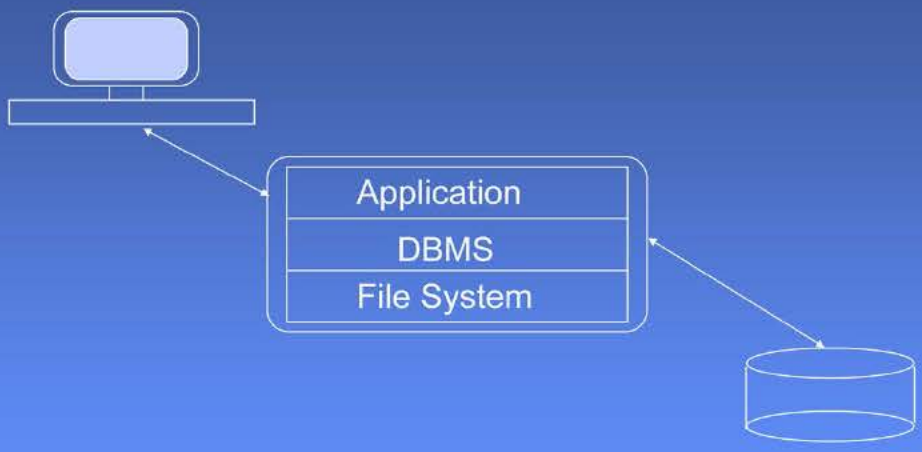
ERh method lends itself to automation. db design tools: ER diagram editors; ER-to-SQL

For normalization, transforming FD to ER can be done. Not much to help creating the FD (intuitive problem like ER).

Database system architectures

- Monolithic systems
- Client–server systems
- Parallel database servers
- Multidatabase systems

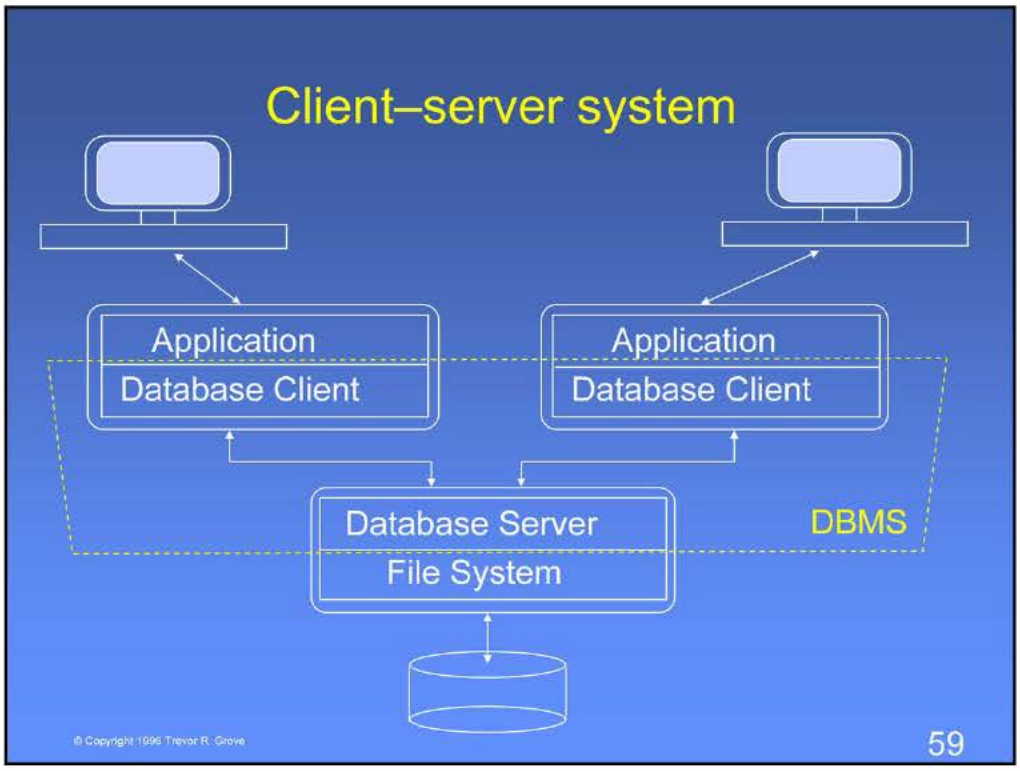
Monolithic system



- Each component presents a well-defined interface to the component above

Component functions

- applications:
  - user interaction: input of queries and data, display of results
  - application-specific tasks
- DBMS:
  - query processing and optimization: select and execute one of many possible procedures for doing a query
  - buffer management: allocation and control of memory
  - transaction management: concurrency control, rollback, and failure recovery
  - security and integrity management: access control and consistency checking
- file system:
  - storage and retrieval of unstructured data on disks



notice that the DBMS has been split in half, into server and client

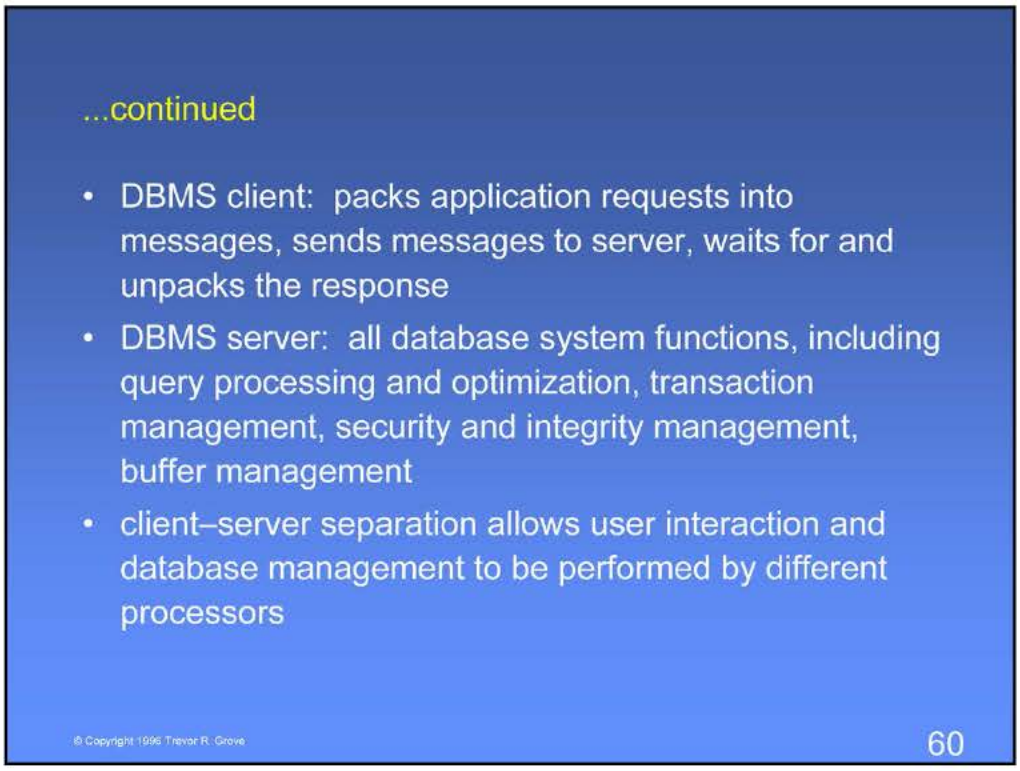
application and DBclient go together, DBserver and FS go together

applications are still presented with a single DBMS and are not aware of the separation between the client and server

middleware: software layers that enable communications. generally part of the op. sys networking protocols. better called “client-server protocol”

A client-server protocol dictates the manner in which clients request information and services from a server and also how the server replies to that request (some examples of client-server protocols are NetBIOS, RPC, Advanced Program-to-Program Communication (APPC), Named Pipes, Sockets, Transport Level Interface (TLI) and Sequenced Packet Exchange (SPX)). [from comp.client-server FAQ]

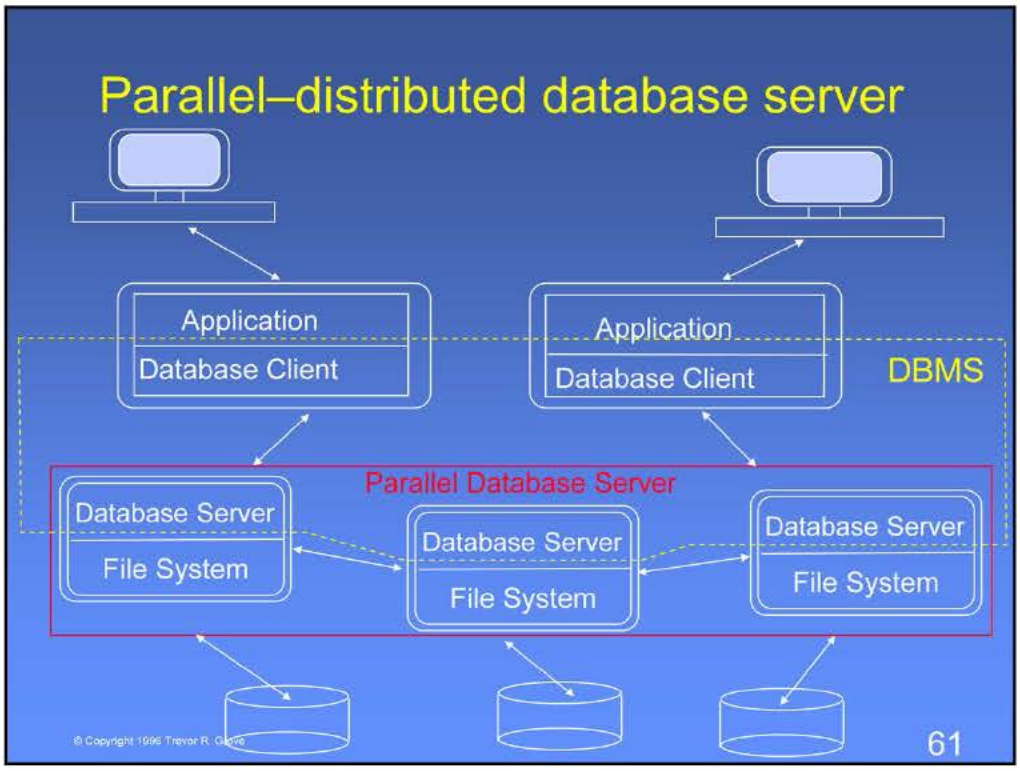
OSF-DCE can also be considered as middleware



IE. client does application-related things; server does gory system-related stuff

this is a two-tier arch.; sometimes in large systems a three-tier system is necessary (agent: handles resource metering, translations)

This structure lends itself to putting the client stuff and the server stuff on different computers. but certainly doesn't have to be that way. Could quite happily argue that this is a better way to organize software in general. Lends itself to object-oriented design and implementation.



the applications still talk to a single server (trsansparently, via a client)

however, the server is actually a collection of separate servers, each with its own underlying system

the organization of the collection is “hidden” in the sense that nothing outside the collection is aware. In typical implementations, one server would act as a coordinator for the activitites of the collection (however the coordinator might change dynamically)

data can be spread out between the ser vers. if this occurs, it is transparent to the applications

...continued

- data is distributed across the **sites**
- relations may be fragmented
- relations (or fragments of relations) may be replicated at several sites
- clients perceive a single database with a single, common schema

Transparency:

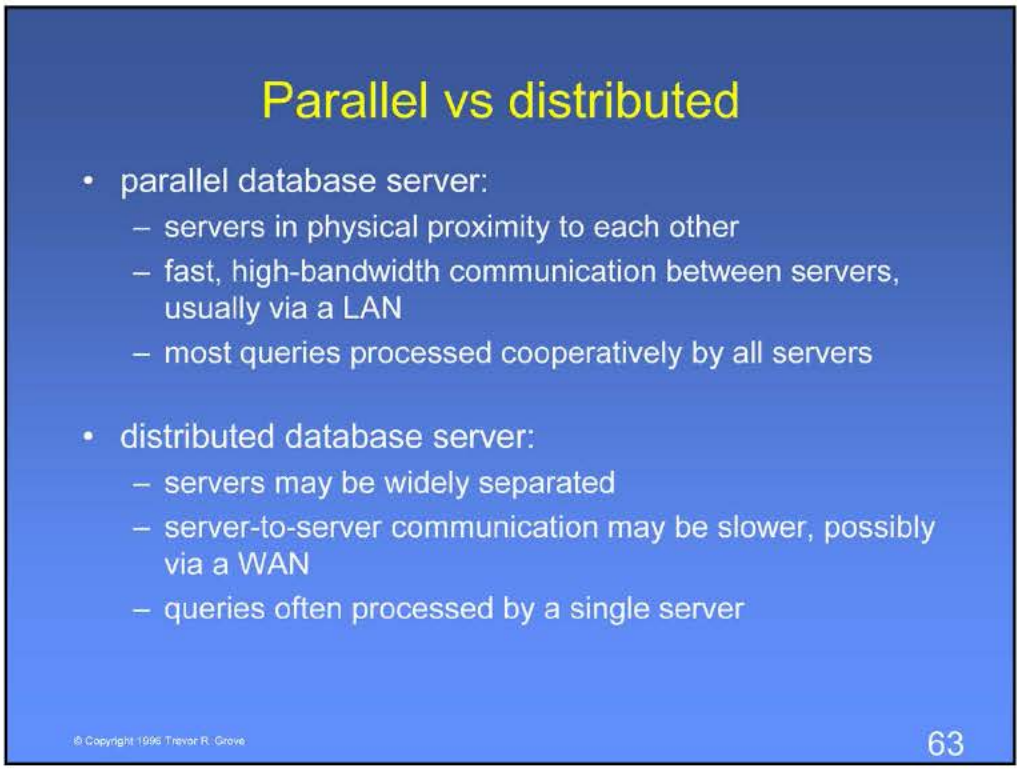
- distribution of data is transparent
- distribution of computation is transparent
- replication is transparent
- fragmentation is transparent

spreading data out is the primary reason for parallel/distributed systems.

this can involve splitting relations and putting chunks at different places, or making copies of relations at many sitest

all of this is intended to improve the performance and reliability of the DBMS

complicated, fragmentation replication is problematic



parallel well-suited to dynamic load balancing

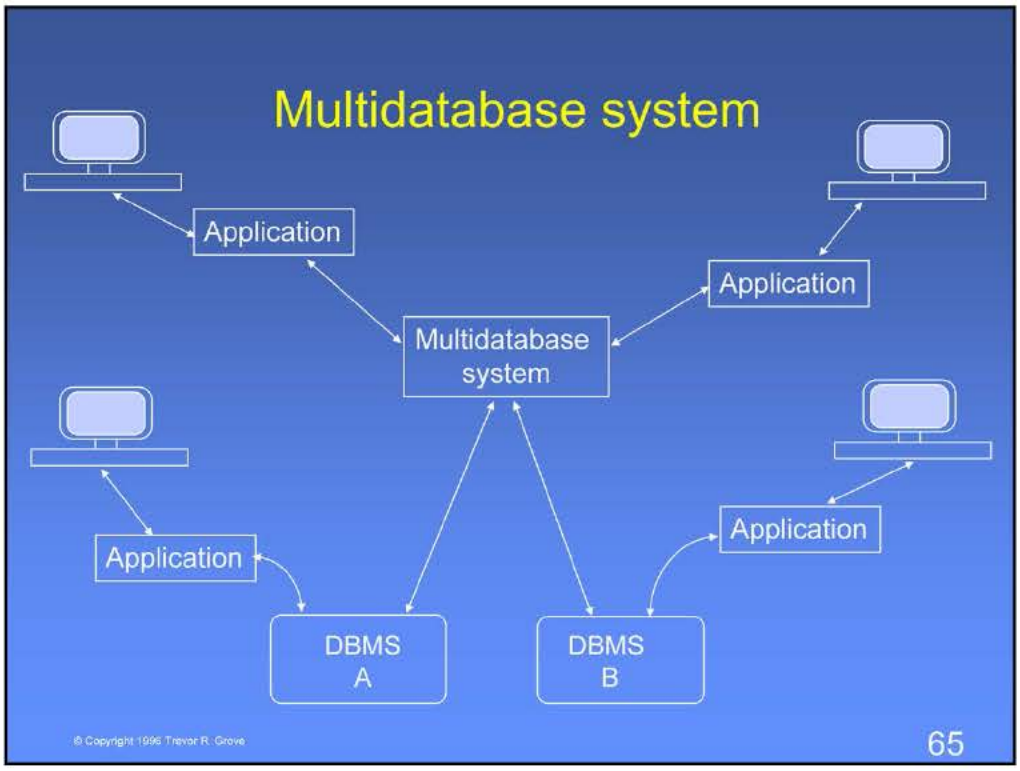
distributed good for isolated sub-tasks

security is less of a concern with an corporate/internal LAN. Using a public carrier for WAN services may cause security exposures

Parallel–distributed: why?

- reliability and availability: if one server fails, another can take its place
- faster query processing: several servers can cooperate to process a query
- data sharing with distributed control: individual sites can share data while retaining some autonomy

Why not? Complicated, lots of places for failure. Networks involved, security, reliability.



a multidatabase system inserts a software layer that acts as a manager or broker between applications and a collection of autonomous (non-integrated databases.

as the picture suggests, it is acceptable for the DBMSs to have their own clients, as well as the ones brokered by the multidatabase

an example: DBMS A and B are member institutions of credit-card co. they each have their own application bases, but also support multidb transactions coming from shared POS system

...continued

- multidatabase system (MDBS) create illusion of integrated single DBMS
- applications perceive a single database system
- servers are autonomous; may use different schemas (external, conceptual, internal) or DBMS models
- some such operations may be difficult or impossible: the MDBS must be able to construct a homogeneous schema and handle distributed–parallel complexities, too

this kind of system can br really complex and has inherent implementation challenges

Emerging technologies

- Object-oriented databases
- Integrating autonomous databases
- Futures

remote DB clients (eg laptops) are an up-and-coming technology area.

Object-oriented database systems

```
class Account  {
   int ANum;
   float Balance;
   Set <Ref<Customer>> AccountHolders;
   Ref<Branch> Branch;
   // Methods
   void Deposit (float amount);
   void Withdraw (float amount);
};
```

- class definition is like a relation schema; attributes are similar to relation attributes
- unlike relations, classes may have **methods** (arbitrary procedures)

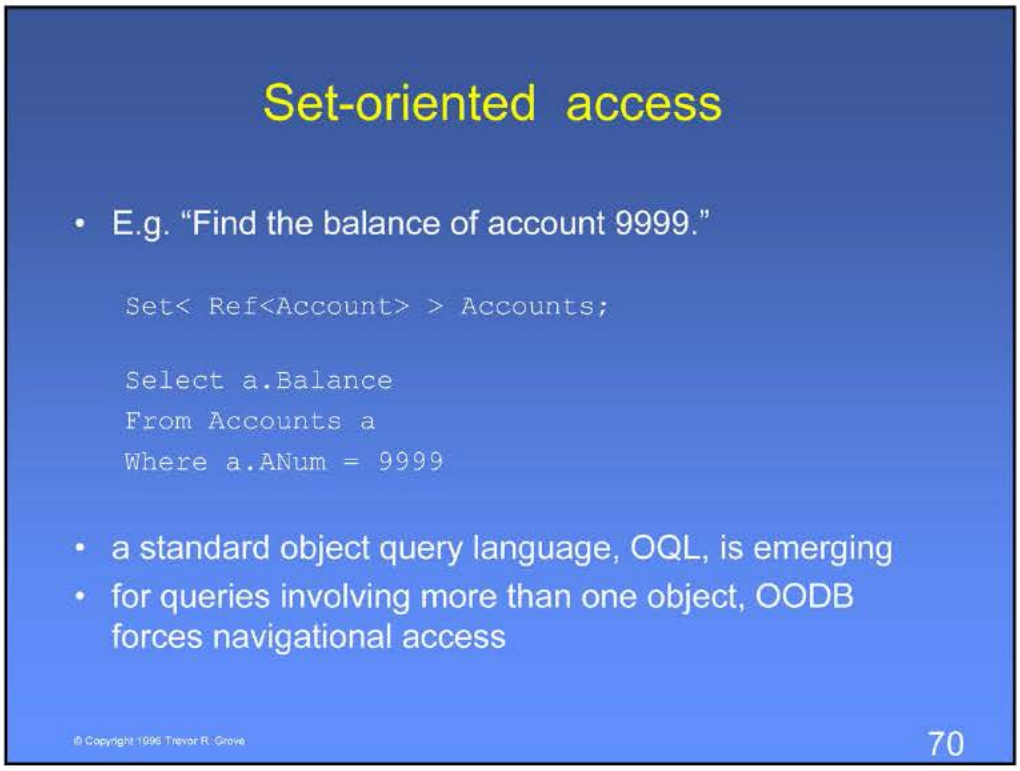
set<ref<customer>> AccountHolder

means AccountHolder is an Attribute. It is a set of values, each of which is found in the Customer object (ie a reference)

Inheritance

```
class CheckingAccount:  Account{
   float PerCheckFee;
   // Methods
   void CheckPayout (float amount);
};
```

- CheckingAccount is a subclass of Account; all methods that apply to Accounts also apply to CheckingAccounts

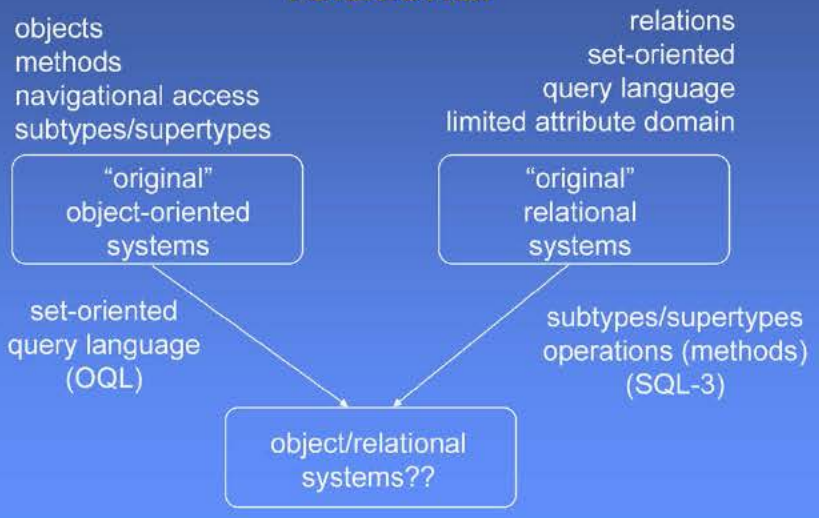


Even though this is object-priented, we really want to process sets of values in response to a query (just like relational).

set ... creates the object instance (like a variable declaration)

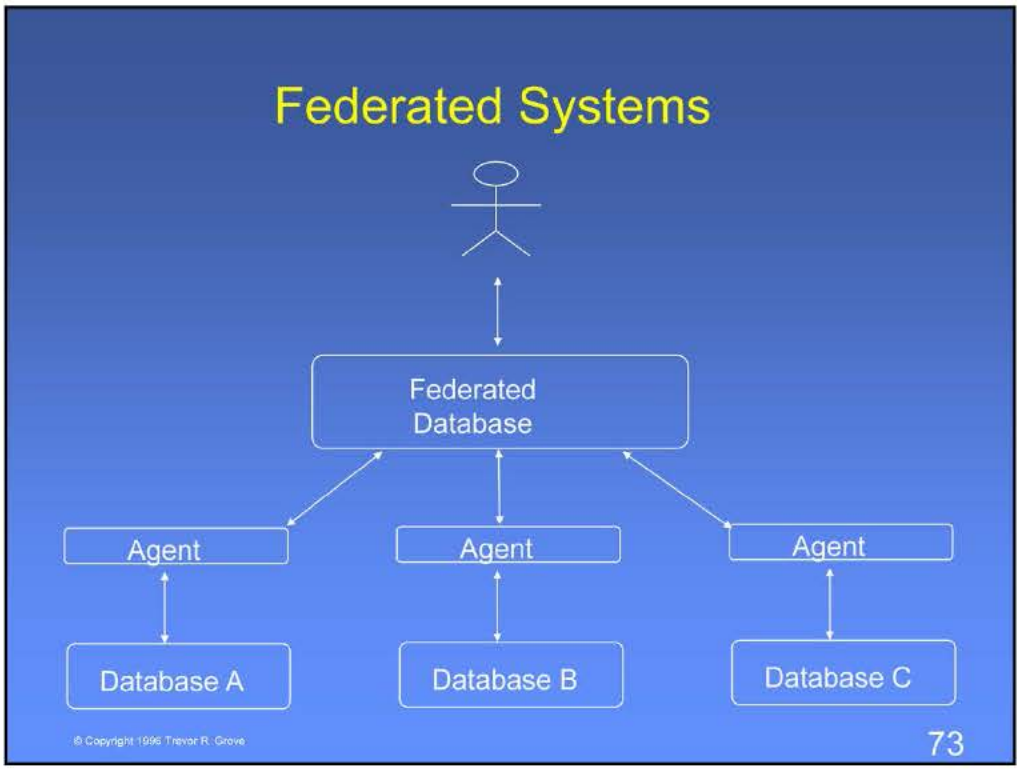
select the balance from an account (correlation name a) where the number is 9999

Convergence of object-oriented and relational



Integrating autonomous systems

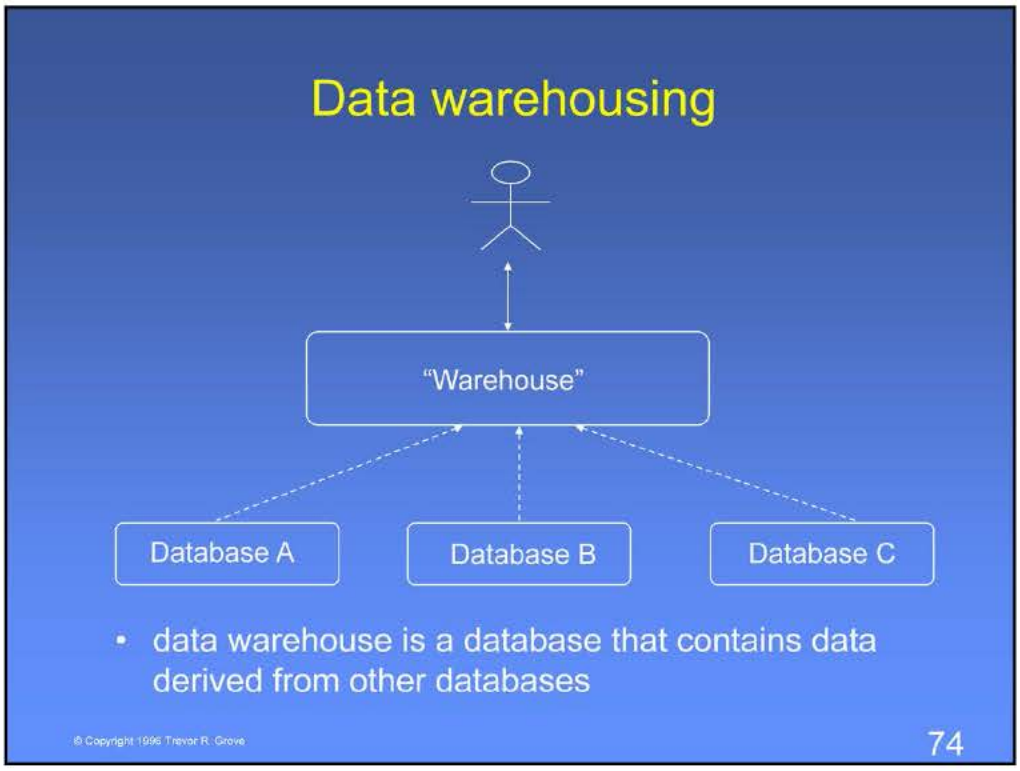
- large organizations (corporations, government agencies) often have multiple autonomous, heterogeneous information resources (databases).
- what kind of "glue" is needed to allow such systems to fully exploit these resources?
- some possible answers:
  - federated database systems, or gateways
  - warehouses (data warehouses)
  - workflow management systems



Similar to MDBS; collection of independent DBMS.

However, all access is through a single point (the federated DB), giving the appearance of a single DB, and preventing independent access to the DBMSs (this is the big difference).

Each DBMS is “fronted” by an agent process that translates/arranges details, so federated DB can have a uniform interface to each constituent.



Data warehouse is a database with an accumulation of data from other databases. Other DB need not necessarily be relational; could be anything. Note arrows are uni-directional; reloads must be done periodically.

Generally used in read-only applications like decision-support systems, exec info systems. subordinate DBs still used for online applications

Just a big DB. the DBMS may need to be optimized for the large size (sum of all other databases).

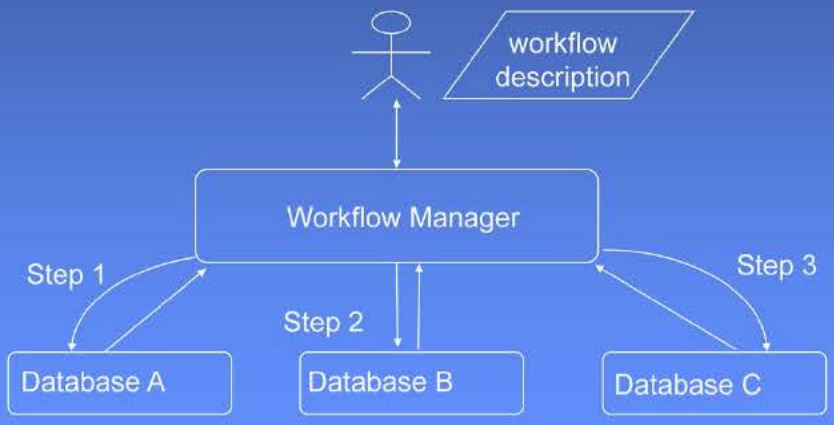
Trick is to create db structure in the warehouse that organizes data from hererogeneous sources and facilitates finding and correlating data. support "drilling down"

Workflow management

- consider the procedure for taking a business trip:
  - obtain travel approval from manager
  - obtain travel advance from financial
  - obtain reservations from travel
  - take the trip
  - obtain expense reimbursement from financial
- each step may require query and update of one or more databases.

...continued

- A **workflow management system** allows such operations to be defined, stored, maintained, and executed



Futures

- Remote databases
  - nomadic database applications (laptops)
  - cache local modifications, update master later
  - refresh other copies
- Text databases
  - storing large amounts of text in a relational DBMS
  - overlap with “information retreival”
  - implications for WWW publishing

Summary

- Relational databases based on formal mathematical systems; mature, stable technology
- SQL is predominant query language for r-DBMS
- Several application development techniques exist, including ad hoc query facilities, ESQL and 4GLs
- Data modelling is the process of designing a DB schema; ER modelling & normalization are methods
- DBMS architectures: monolithic, client–server, distributed–parallel, multidatabase
- Emerging & future technologies: object-oriented databases, autonomous DB integration, remote databases, text databases

Navigational access

- object-oriented databases suffer from navigational access, which relational databases had elminiated!

```
class Branch  {
  string City;
  string Address;
  Ref<Employee> Manager;
};

class Employee  {
 string Name;
 string ENum;
 // etc., etc.,
};
```

...continued

- E.g. "Find the name of the manager at the branch where account 9999 is located."

```
Select a.Branch.Manager.Name
From Accounts a
Where a.ANum = 9999
```

- finds the name of the manager of the branch for account 9999 by traversing three objects: an account object, a branch object, and an employee object.
- In a relational system, a similar query would be answered by joining the Accounts, Branch, and Employee relations.