PURDUE
UNIVERSITY

CS34800
Information Systems

Course Review

Prof. Chris Clifton

7 December 2016

Indiana Center for Database Systems ™

1

Course Outline

- Relational Databases
  1. Relational model overview
  2. Formal definitions, relational operations
  3. Query: SQL
- Database Design
  4. Entity-Relationship Model
  5. Relational Design
  6. Database Normalization
  7. Object Databases
  8. XML databases
- Integrity and Consistency
  9. Transactions
  10. Concurrency
  11. Constraints
- Advanced Topics
  12. Big Data: MapReduce, Hadoop, Spark
  13. Data Analysis / Data Mining
  14. Information Retrieval

What is a Database?

- Collection of data, used to represent the information of interest to one or more applications in a given organization
  - Usually large
  - Organized for rapid search and retrieval
- Database Management System (DBMS):
  Tool to ease construction of databases
  - (Vendor) definition of database: Collection of data managed by a DBMS
- Desirable Properties:
  - Persistent Storage
    A File System does this
  - Query Interface
    Information retrieval system
  - Transaction Management

Goals of a DBMS

Data Integration	→	Enhances the accessibility of data, reduces redundancies and inconsistencies
Data Independency	→	Simplifies the development of new applications, and the maintainance of existing applications
Centralized Data Control	→	Assures data quality, confidentiality, and integrity

Services provided by a DBMS

Service	Description
Data definition	To specify the data to be stored
Data manipulation	To access data, to insert new data, to modify and delete existing data
Semantic integrity	To prevent the storage of incorrect data
Storage structures	To represent in secondary storage the data model constructs, to store efficienty store and retrieve data
Query optimization	To determine the most efficient strategy for data access
Access control	To protect data from unauthorized accesses
Recovery	To prevent data inconsistency in case of errors and failures
Data dictionary	To determine the data stored in a database and access their definitions

Data Models

- A data model is a "conceptual tool", or *formalism*, that includes three fundamental components:
  - One or more data structures.
  - A notation to specify the data through the data structures of the model.
  - A set of operations for managing data; these operations are defined in terms of the data structures of the model.

Data Models

- Each data model must answer two fundamental questions:
  - (a) how to represent the entities and their attributes
  - (b) how to represent the relationships

(a) Almost all models use structures such as the record; each component in a record represents an attribute

(b) Data models widely in this respect; relationships can be represented as:

- – specific structures, values, pointers (logical or physical)

Levels of Abstraction

- **Physical level:** describes how a record (e.g., instructor) is stored.
- **Logical level:** describes data stored in database, and the relationships among the data.

type *instructor* = **record**

ID : string;
name : string;
dept_name : string;
salary : integer;

end;

- **View level:** application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.

Instances and Schemas

- Similar to types and variables in programming languages
- **Logical Schema** – the overall logical structure of the database
  - Example: The database consists of information about a set of customers and accounts in a bank and the relationship between them
    - Analogous to type information of a variable in a program
- **Physical schema**– the overall physical structure of the database
- **Instance** – the actual content of the database at a particular point in time
  - Analogous to the value of a variable
- **Physical Data Independence** – the ability to modify the physical schema without changing the logical schema
  - Applications depend on the logical schema
  - In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

SQL

- The most widely used commercial language
- SQL is NOT a Turing machine equivalent language
- To be able to compute complex functions SQL is usually embedded in some higher-level language
- Application programs generally access databases through one of
  - Language extensions to allow embedded SQL
  - Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database

Data Definition Language (DDL)

- Specification notation for defining the database schema

  Example: **create table** *instructor* (
  ID **char**(5),
  name **varchar**(20)**,**
  dept_name **varchar**(20),
  salary **numeric**(8,2))

- DDL compiler generates a set of table templates stored in a ***data dictionary***
- Data dictionary contains metadata (i.e., data about data)
  - Database schema
  - Integrity constraints
    - Primary key (ID uniquely identifies instructors)
  - Authorization
    - Who can access what

Data Manipulation Language (DML)

- Language for accessing and manipulating the data organized by the appropriate data model
  - DML also known as query language
- Two classes of languages
  - **Pure** – used for proving properties about computational power and for optimization
    - Relational Algebra
    - Tuple relational calculus
    - Domain relational calculus
  - **Commercial** – used in commercial systems
    - SQL is the most widely used commercial language

Relational Algebra

Mathematical view of SQL queries

- Formal view of what we've seen in SQL
  - Selection (where clause)
  - Projection (select clause)
- Cartesian Product
- Join
- Set operations
- Aggregation

Relational Algebra
A good way to think about queries

σ SELECT
π PROJECT
} UNARY

X CARTESIAN PRODUCT
U UNION
– SET-DIFFERENCE
$\cap$ SET-INTERSECTION
} BINARY
} FUNDAMENTAL

$\bowtie_{\theta}$ THETA-JOIN
$\bowtie$ NATURAL JOIN
÷ DIVISION or QUOTIENT
} CAN BE DEFINED IN TERMS OF FUNDAMENTAL OPS

- Properties:
  - Limited expressive power (subset of possible queries), but rich enough to be useful
  - Closed (input is relation, output is relation)
  - Finite (result always defined)
  - Easy to automatically determine how to efficiently process

Additional operators

- Grouping: $\gamma_L(R)$
  - L is a list of elements that are either
    - Individual (grouping) attributes, or an
    - Aggregate $\theta(A)$, where θ is an aggregation operator and A the attribute to which it is applied
- Aggregation operators
  - Sum, Average, Count, Min, and Max
  - Return a single row for projection, a row for each group with group-by
- Renaming: $\rho_x(E_1)$, x is the name for E_1

18

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.

    select *name*
    from *instructor*
    where *salary* **is null**

Database System Concepts - 6th Edition 3.20 ©Silberschatz, Korth and Sudarshan

Null Values and Aggregates

- Total all salaries

  select sum (*salary*)
  from *instructor*

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

Accessing SQL From a Programming Language

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- Various tools:
  - JDBC (Java Database Connectivity) works with Java
  - ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic. Other API's such as ADO.NET sit on top of ODBC
  - Embedded SQL

Key concept: *Cursor*

- Query returns a table
  - Could be viewed as a "Set" data type
  - Not all programming languages deal with this
- Instead, idea of a *cursor* to iterate over table
  - Access one row of result at a time
  - Typically used in a loop construct in the language
- Query processor "understands" cursor
  - Can start making results available before query completes

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors

ODBC

- Open DataBase Connectivity (ODBC) standard
  - standard for application program to communicate with a database server.
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC

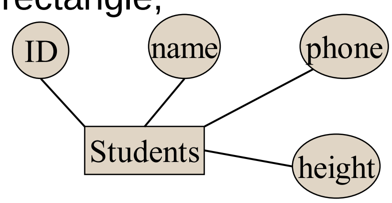
Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language (e.g., C, Java).
  - Functions written in an external languages are particularly useful with specialized data types such as images and geometric objects.
    - Example: functions to check if polygons overlap, or to compare images for similarity.
  - Some database systems support **table-valued functions**, which can return a relation as a result.
    - *Think of this as a (very complex) view*
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

Entity/Relationship Model

Diagrams to represent designs.

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

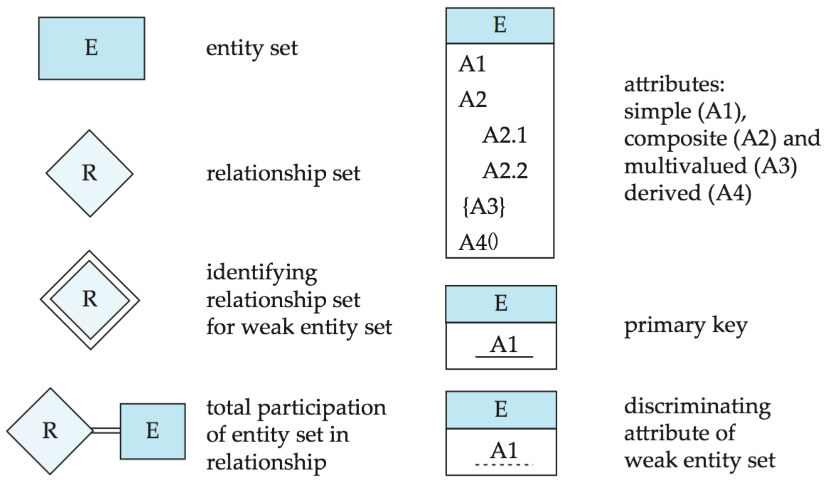


Relationships

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

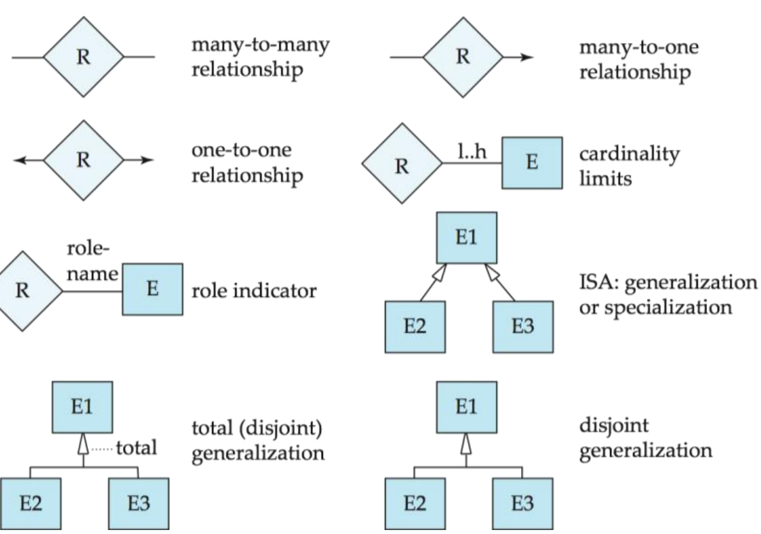
33

Summary of Symbols Used in E-R Notation



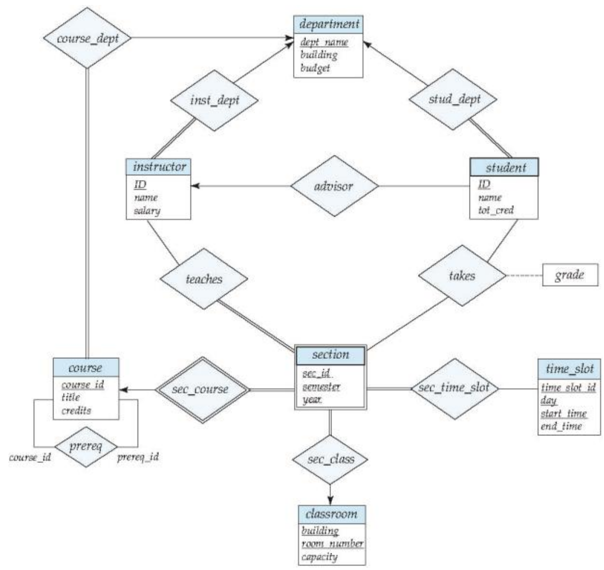
Database System Concepts - 6th Edition 3.34 ©Silberschatz, Korth and Sudarshan

Symbols Used in E-R Notation (Cont.)



Database System Concepts - 6th Edition 3.35 ©Silberschatz, Korth and Sudarshan

E-R Diagram for a University Enterprise

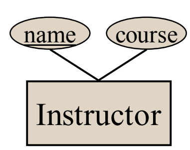


Relational Design

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

Entity Set → Relation

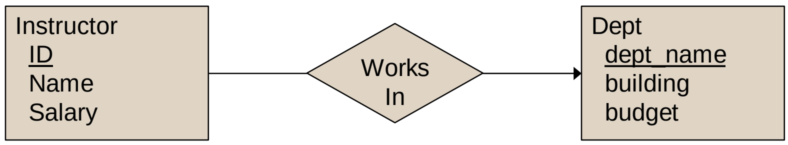
E.S. attributes become relational attributes.



Becomes:

`Instructor(name, course)`

Relational Design



- Instructor(<u>ID</u> number(10) primary key,
  Name varchar(40),
  Salary number(6))
- Dept(<u>dept_name</u> varchar(20) primary key,
  building varchar(30),
  budget number(8))
- Works_in(ID references Instructor(ID),
  dept_name references Dept(dept_name))

Keys of Relations

K is a *key* for relation *R* if:

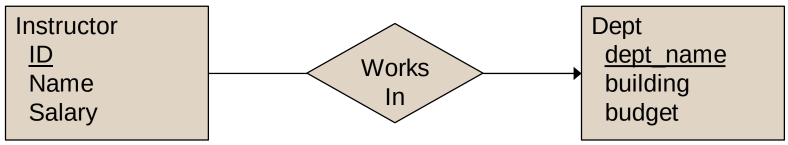
1. $K \rightarrow$ all attributes of *R*. **(Uniqueness)**
2. For no proper subset of *K* is (1) true. **(Minimality)**

- If *K* at least satisfies (1), then *K* is a *superkey*.

Conventions

- Pick one key; underline key attributes in the relation schema.
- *X*, etc., represent sets of attributes; *A* etc., represent single attributes.

Relational Design



- Instructor(<u>ID</u> number(10) primary key,
  Name varchar(40),
  Salary number(6))
- Dept(<u>dept_name</u> varchar(20) primary key,
  building varchar(30),
  budget number(8))
- Works_in(ID references Instructor(ID),
  dept_name references Dept(dept_name))

Key for Works_in?
A. ID
B. dept_name
C. both
D. neither

Lossless Join

- Goal: All legal values can be stored in relations
  - Recover originals through join
- Formally: X, Y is a lossless join decomposition of R w.r.t. F if $\forall r \in R$ satisfying dependencies in F,
  $\pi_X(r) \bowtie \pi_Y(r) = r$

A Lossy Decomposition

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

employee

ID	name
⋮	
57766	Kim
98776	Kim
⋮	

name	street	city	salary
⋮			
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000
⋮			

natural join

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

Functional Dependencies

$X \rightarrow A$ = assertion about a relation *R* that whenever two tuples agree on all the attributes of *X*, then they must also agree on attribute *A*

Why do we care?

Knowing functional dependencies provides a formal mechanism to divide up relations *(normalization)*

- Saves space
- Prevents storing data that violates dependencies

Armstrong's Axioms

- Armstrong's Axioms:
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (reflexivity)
  - if $\alpha \rightarrow \beta$, then $\gamma\,\alpha \rightarrow \gamma\,\beta$ (augmentation)
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (transitivity)
- *Owner pet_name age → species*
  species → vaccination
- Applying transitivity gives:
  A. *pet_name age → species*
  B. *Owner → vaccination*
  C. *Vaccination → species*
  D. *Owner pet_name age → vaccination*
  E. Transitivity can't be applied to these rules

Functional Dependencies (FD's) and Many-One Relationships

- Consider *R*(*A*1,…, *An*) and *X* is a key
  then $X \rightarrow Y$ for any attributes *Y* in *A*1,…, *An*
  even if they overlap with *X*. Why?
- Suppose *R* is used to represent a many → one relationship:
  *E*1 entity set → *E*2 entity set
  where *X* key for *E*1, *Y* key for *E*2,
  Then, $X \rightarrow Y$ holds,
  And $Y \rightarrow X$ does not hold unless the relationship is one-one.
- What about many-many relationships?

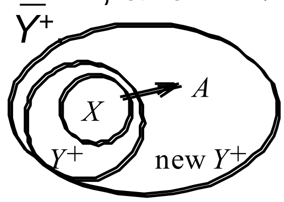
Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F.
  - For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by F is the **closure** of F.
- We denote the *closure* of F by F^+.
- F^+ is a superset of F.

Algorithm

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

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



- End when Y^+ cannot be changed.

Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \; B \rightarrow C, \; A \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - E.g.: on RHS: $\{A \rightarrow B, \; B \rightarrow C, \; A \rightarrow CD\}$ can be simplified to $\{A \rightarrow B, \; B \rightarrow C, \; A \rightarrow D\}$
    - E.g.: on LHS: $\{A \rightarrow B, \; B \rightarrow C, \; AC \rightarrow D\}$ can be simplified to $\{A \rightarrow B, \; B \rightarrow C, \; A \rightarrow D\}$
- Intuitively, a canonical cover of F is a "minimal" set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

Computing a Canonical Cover

- $R = (A, B, C)$
  $F = \{A \rightarrow BC$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
  - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
    - Yes: in fact, $B \rightarrow C$ is already present!
  - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
  - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
    - Yes: using transitivity on $A \rightarrow B$ and $B \rightarrow C$.
      - Can use attribute closure of A in more complex cases
- The canonical cover is: $A \rightarrow B$
  $B \rightarrow C$

CS34800 Information Systems

Course Review

Prof. Chris Clifton

9 December 2016

Normalization

- Let R be a relation scheme with a set F of functional dependencies.
- Decide whether a relation scheme R is in "good" form.
- In the case that a relation scheme R is not in "good" form, decompose it into a set of relation scheme $\{R_1, R_2, ..., R_n\}$ such that
  - each relation scheme is in good form
  - the decomposition is a lossless-join decomposition
  - Preferably, the decomposition should be dependency preserving.

BCNF

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

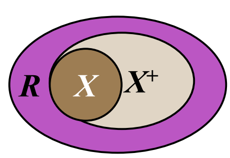
- Formally, *R* is in BCNF if for every nontrivial FD for *R*, say $X \rightarrow A$, then *X* is a superkey.
  - "Nontrivial" = right-side attribute not in left side.

Why?

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

Algorithm for BCNF

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



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

Third Normal Form

- A relation schema R is in **third normal form (3NF)** if for all:

  $$\alpha \rightarrow \beta \text{ in } F^+$$

  at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
  - α is a superkey for R
  - Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

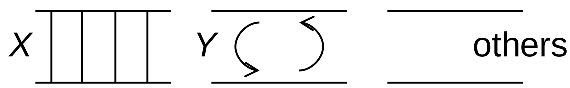
  (**NOTE**: each attribute may be in a different candidate key)
- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).

3NF Synthesis

- Given a canonical cover F_C for F
- Schema $S = \emptyset$
- $\forall\ A \rightarrow B \in Fc$
  - If there is no $R_i \in S$ such that $AB \subseteq R_i$
    - $S = S + AB$
- If there is no $R_i \in S$ containing a candidate key for R
  - $S = S$ + (any candidate key for R)

Multivalued Dependencies

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



Example: ID →→ name

MVD Rules

1. Every FD is an MVD.
   - Because if $X \rightarrow Y$, then swapping Y's between tuples that agree on X doesn't create new tuples.
   - Example: `name →→ addr`.
2. *Complementation*: if $X \rightarrow\rightarrow Y$, then $X \rightarrow\rightarrow Z$, where Z is all attributes not in X or Y.
   - Example: since `ID →→ name addr`
     `name addr →→ ID`

4NF Decomposition

- Let schema $S = R$,
  $D+$ be the closure of the functional and multivalued dependencies
- While $\exists\ R_i \in S$ not in 4NF w.r.t. $D+$
  - Choose a nontrivial multivalued dependency $A \rightarrow\rightarrow B$ that holds on R_i, where $A \rightarrow R_i \notin D+$, and $A \cap B = \emptyset$
  - $S = (S - R_i) \cup (R_i\text{-}B) \cup (A,B)$

Data Modification

- **insert into** *student*
  values ('3003', 'Green', 'Finance', *null*);
- **delete from** *instructor*
  where *dept_name*= 'Finance';
- **update** *instructor*
  set *salary* = *salary* * 1.03
  where *salary* > 100000;

Integrity Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate

XML

- XML: Extensible Markup Language
  - Developed by WWW Consortium as more flexible version of HTML
  - Derived (as with HTML) from SGML (Standard Generalized Markup Language)
- Goal: Add structure to document
  - Describe content, not presentation
- Key idea: tags
  - <title>Introduction to XML</title>
  - <list><item>XML: Exten… </item> <item>…</list>

Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constraints structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
  - All values represented as strings in XML
- DTD syntax
  - <!ELEMENT element (subelements-specification) >
  - <!ATTLIST element (attributes) >

University DTD with Attributes

- University DTD with ID and IDREF attribute types.

```
<!DOCTYPE university-3 [
   <!ELEMENT university ( (department|course|instructor)+)>
   <!ELEMENT department ( building, budget )>
   <!ATTLIST department
        dept_name ID #REQUIRED >
   <!ELEMENT course (title, credits )>
   <!ATTLIST course
        course_id ID #REQUIRED
        dept_name IDREF #REQUIRED
        instructors IDREFS #IMPLIED >
   <!ELEMENT instructor ( name, salary )>
   <!ATTLIST instructor
        IID ID #REQUIRED
        dept_name IDREF #REQUIRED >
   · · · declarations for title, credits, building,
        budget, name and salary · · ·
]>
```

XML data with ID and IDREF attributes

```
<university-3>
    <department dept name="Comp. Sci.">
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept name="Biology">
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course id="CS-101" dept name="Comp. Sci"
            instructors="10101 83821">
        <title> Intro. to Computer Science </title>
        <credits> 4 </credits>
    </course>
    ….
    <instructor IID="10101" dept name="Comp. Sci.">
        <name> Srinivasan </name>
        <salary> 65000 </salary>
    </instructor>
    ….
</university-3>
```


XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs. Supports
  - Typing of values
    - E.g. integer, string, etc
    - Also, constraints on min/max values
  - User-defined, comlex types
  - Many more features, including
    - uniqueness and foreign key constraints, inheritance
- XML Schema is itself specified in XML syntax, unlike DTDs
  - More-standard representation, but verbose
- XML Scheme is integrated with namespaces
- BUT: XML Schema is significantly more complicated than DTDs.

XML Schema Version of Univ. DTD

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="building" type="xs:string"/>
            <xs:element name="budget" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
….
<xs:element name="instructor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

… Contd.

Manipulating XML Data

- XQuery
  - Based on *sequences*, not sets
  - Describe path within document (XPath)
  - Variables, wildcards, etc. within path that are matched
- Parser-based access (E.g., Java API to XML)
  - Designed for string representation of document
  - DOM: Tree traversal
  - SAX: Streaming through document

XPath

- XPath is used to address (select) parts of documents using **path expressions**
- A path expression is a sequence of steps separated by "/"
  - Think of file names in a directory hierarchy
- Result of path expression: set of values that along with their containing elements/attributes match the specified path
- E.g. /university-3/instructor/name evaluated on the university-3 data we saw earlier returns

  <name>Srinivasan</name>
  <name>Brandt</name>

- E.g. /university-3/instructor/name/text( )
  returns the same names, but without the enclosing tags

XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
  - The textbook description is based on a January 2005 draft of the standard. The final version may differ, but major features likely to stay unchanged.
- XQuery is derived from the Quilt query language, which itself borrows from SQL, XQL and XML-QL
- XQuery uses a
  for … let … where … order by …result …
  syntax
  for ⇔ SQL **from**
  where ⇔ SQL **where**
  order by ⇔ SQL **order by**
  result ⇔ SQL **select**
  let allows temporary variables, and has no equivalent in SQL

Object-Oriented Databases

- Goal: Provide same benefits as object-oriented programming
  - Abstraction
  - Reuse
  - Natural data modeling
- A number of commercial systems
  - Gemstone (1986)
  - Informix, ObjectDB, O2, …

Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.

Complex Types and SQL

- Extensions introduced in SQL:1999 to support complex types:
  - Collection and large object types
    - Nested relations are an example of collection types
  - Structured types
    - Nested record structures like composite attributes
  - Inheritance
  - Object orientation
    - Including object identifiers and references
- Commercial databases may vary from the standard
  - But some features are present in each of the major commercial database systems
    - Read the manual of your database system to see what it supports

Transaction

- Sequence of operations treated as a "single unit"
  - Either all happen, or none do
- Various syntaxes
  - SQL:1999 : **begin atomic** … **end**
  - Oracle: **set transaction** … **commit**
- Default in most DBMSs: each statement is a transaction
- Also "Rollback" (undo before commit)

Second goal of transactions: *Sequence* of Operations

- Update should complete entirely
  - update stipend set stipend = stipend*1.03;
  - What if it gets halfway and the machine crashes?
- What about multiple operations?
  - Withdraw x from Account1
  - ~~Deposit x into Account2~~
- Simultaneous operations?
  - Print paychecks while stipend being updated

ACID properties

Transactions have:

- Atomicity
  - All or nothing
- Consistency
  - Changes to values maintain integrity
- Isolation
  - Transaction occurs as if nothing else happening
- Durability
  - Once completed, changes are permanent

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

(If each transaction preserves consistency, every serializable schedule preserves consistency.)

Chris Clifton - CS34800

Example

- Consider two transactions:

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.01*A,	B=1.01*B	END

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.
- Assume A=100, B=100 at start. Result:
  A. A = 202, B = 0
  B. A = 201, B = 1
  C. A = 202, B = 1
  D. A = 201, B = 0

Example (Contd.)

- Consider a possible interleaving:

T1:	A=A+100, B=B-100	
T2:		A=1.01*A, B=1.01*B

- Assume A=100, B=100 at start. Result:
  A. A = 202, B = 0
  B. A = 201, B = 1
  C. A = 202, B = 1
  D. A = 201, B = 0

Example (Contd.)

- Consider a possible interleaving:

T1:		A=A+100, B=B-100
T2:	A=1.01*A, B=1.01*B	

- Assume A=100, B=100 at start. Result:
  A. A = 202, B = 0
  B. A = 201, B = 1
  C. A = 202, B = 1
  D. A = 201, B = 0

Example (Contd.)

- Consider a possible interleaving:

T1:	A=A+100,		B=B-100	
T2:		A=1.01*A,		B=1.01*B

- Assume A=100, B=100 at start. Result:
  A. A = 202, B = 0
  B. A = 201, B = 1
  C. A = 202, B = 1
  D. A = 201, B = 0

Chris Clifton - CS34800

Example (Contd.)

- Consider a possible interleaving:

T1:	A=A+100,		B=B-100
T2:		A=1.01*A, B=1.01*B	

- Assume A=100, B=100 at start. Result:
  A. A = 202, B = 0
  B. A = 201, B = 1
  C. A = 202, B = 1
  D. A = 201, B = 0

Chris Clifton - CS34800

Views

- Start with tables

Career	Last	First	Address
clifton	Clifton	Chris	LWSN 2142F

Career	Course
clifton	CS34800
clifton	CS54100

- Create "view" for convenience
  - create view courseList as
    select i.Career, Last, First, Address, Course
    from instructors I, courses c
    where i.Career = c.Career

Career	Last	First	Address	Course
clifton	Clifton	Chris	LWSN 2142F	CS34800
clifton	Clifton	Chris	LWSN 2142F	CS54100

View: Semantics

- Contents of view are current *at the time it is used*
  - If base tables are updated, view is updated
- Equivalent to replacing the view with a subquery

```
select * from courseList where course='CS34800' ≡
select * from
  (select i.Career, Last, First, Address, Course
  from instructors I, courses c
  where i.Career = c.Career)
where course='CS34800'
```

Update issue

Career	*Last*	*First*	*Address*	*Course*
clifton	Clifton	Chris	LWSN 2142F	CS34800
clifton	Clifton	Chris	LWSN 2142F	CS54100

- Insert into courseList values (‘clifton’, ‘Clifton’, ‘Chris’, ‘LWSN 2142F’, ‘CS54701’);

Career	*Last*	*First*	*Address*
clifton	Clifton	Chris	LWSN 2142F
clifton	Clifton	Chris	LWSN 2142F

Career	*Course*
clifton	CS34800
clifton	CS54100
clifton	CS54701

SQL Access Control

- grant select on <table> to <user>;
  - grant insert, delete, update
  - with grant option
    - *Allows “passing on” privileges*
- <table> can also be a view
  - But some caveats on updating/insert/delete

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as :** for deletes and updates
  - **referencing new row as :** for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
begin atomic
        set nrow.grade = null;
end;
```

Trigger to Maintain credits_earned value

- **create trigger** *credits_earned* **after update of** *takes* **on** (*grade*)
  referencing new row as *nrow*
  referencing old row as *orow*
  for each row
  when *nrow.grade* <> 'F' **and** *nrow.grade* **is not null**
  and (*orow.grade* = 'F' **or** *orow.grade* **is null**)
  begin atomic
  update *student*
  set *tot_cred*= *tot_cred* +
  (**select** *credits*
  from *course*
  where *course.course_id*= *nrow.course_id*)
  where *student.id* = *nrow.id*;
  end;

Indexing and Hashing

- Goal: Faster access to data
  - Faster than scanning the whole table
- Search Key: attribute/column for which faster search enabled
  - Not the same as keys for database design
- Index: Tree structure allowing faster search
  - Logarithmic time
- Hashing: Group data into "buckets" based on value of search key
  - If all goes well, constant time access

Creating Indexes

- create index <name> on <relation> (<attribute_list>)
  - create index students_i_name on students (lastname);
- Can specify type of index
  - create bitmap index students_i_id on students(StudentID);
- Also delete: drop index student_i_name;
- Oracle: Index automatically created for primary key or unique constraint

Bitmap Index

- Similar in concept to hashing
  - Key values represented as bits in a (long) vector
  - Particularly good when few possible key values
- Supports easy and/or operations in queries
  - lastname = 'Clifton' AND salary > $100k
- More expensive to update

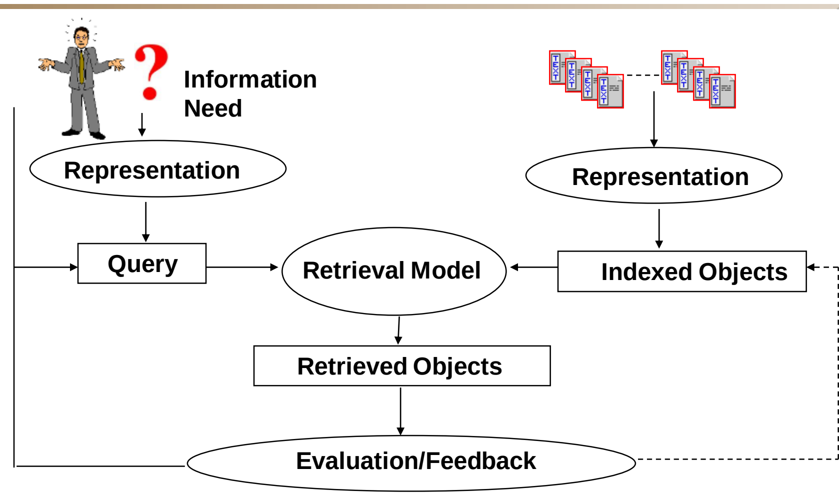
CS34800 100

MapReduce Model

- SQL:
  select word, count(*) from documents group by word
- MapReduce:
  - function map (String name, String document):
    for each word w in document: emit (w, 1)
  - function reduce (String word, Iterator partCounts):
    sum = 0
    for each pc in PartCounts:
      sum += pc
    emit (word, sum)

CS34800 101

AD-hoc IR: Basic Process



Text Representation: Word Stemming

Porter Stemmer

- Based on a pattern of vowel-consonant sequence
  - $[C](VC)^m[V]$, m is an integer
- Rules are divided into steps and examined in sequence
  - Step 1a: ies → i; s →; …..

    cares → care
  - Step 1b: if m>0 eed ee

    agreed → agree
  - ….. Step 5a, Step 5b
- Pretty aggressive:
  - nativity → native

Text Representation: Word Stemming

Examples of Stemming:

- Original Text:

  Information retrieval deals with the representation, storage, organization of, and access to information items

- Porter Stemmer (Stopwords removed):

  Online example:
  http://facweb.cs.depaul.edu/mobasher/classes/csc575/porter.html

  Inform retrieve deal represent storag organ access inform item

Retrieval Models

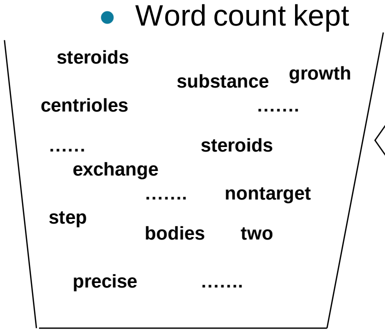
Vector space model vs. Boolean model

- Boolean models
  - Query: a Boolean expression that a document must satisfy
  - Retrieval: Deductive inference
- Vector space model
  - Query: viewed as a short document in a vector space
  - Retrieval: Find similar vectors/objects

Bag of Words (aka Vector Space Model)

The simplest text representation: “bag of words”

- Query/document: a bag that contains words
- Order among words is ignored
  - Word count kept

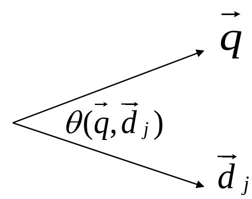


3 steroids	1 cilia-bearing	1 precise	1 two
2 centrioles	1 different	1 receptor	1 unexpected
1 affect	1 exchange	1 regularly	1 vitally
1 already	1 exogenous	1 reveal	1 way
1 Although	1 fluorescent	1 Specific	
1 antibodies	1 growth	1 step	
1 basal	1 identity	1 substance	
1 bodies	1 level	1 suggests	
1 cell	1 localization	1 target	
1 cells	1 nontarget	1 technique	

Retrieval Models: Vector Space Model

Give two vectors of query and document

- query as $\vec{q} = (q_1, q_2, \ldots, q_n)$
- document as $\vec{d}_j = (d_{j1}, d_{j2}, \ldots, d_{jn})$
- calculate the similarity



Cosine similarity: Angle between vectors

$$sim(\vec{q}, \vec{d}_j) = \cos(\theta(\vec{q}, \vec{d}_j))$$

$$\cos(\theta(\vec{q}, \vec{d}_j)) = \frac{\vec{q} \cdot \vec{d}_j}{\|\vec{q}\|\|\vec{d}\|} = \frac{q_1 d_{j,1} + q_2 d_{j,2} + \ldots + q_j d_{j,n}}{\|\vec{q}\|\|\vec{d}\|} = \frac{q_1 d_{j,1} + q_2 d_{j,2} + \ldots + q_j d_{j,n}}{\sqrt{q_1^2 + \ldots + q_n^2}\sqrt{d_{j1}^2 + \ldots + d_{jn}^2}}$$

Evaluation Criteria

- Effectiveness
  - Favor returned document ranked lists with more relevant documents at the top
  - Objective measures
    - Recall and Precision
    - Mean-average precision
    - Rank based precision

For documents in a subset of a ranked lists, if we know the truth

	Retrieved	Not retrieved
Relevant	Relevant docs retrieved	Relevant docs not retrieved
Irrelevant	Irrelevant docs retrieved	Irrelevant docs not retrieved

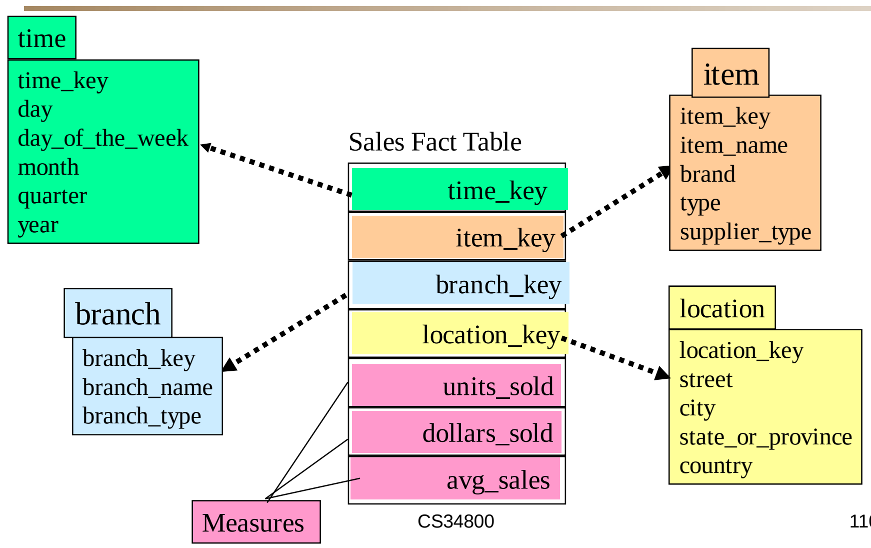
$$\text{Precision} = \frac{\text{Relevant docs retrieved}}{\text{Retrieved docs}}$$

$$\text{Recall} = \frac{\text{Relevant docs retrieved}}{\text{Relevant docs}}$$

Data Warehouse vs. Operational DBMS

- OLTP (on-line transaction processing)
  - Major task of traditional relational DBMS
  - Day-to-day operations: purchasing, inventory, banking, manufacturing, payroll, registration, accounting, etc.
- OLAP (on-line analytical processing)
  - Major task of data warehouse system
  - Data analysis and decision making
- Distinct features (OLTP vs. OLAP):
  - User and system orientation: customer vs. market
  - Data contents: current, detailed vs. historical, consolidated
  - Database design: ER + application vs. star + subject
  - View: current, local vs. evolutionary, integrated
  - Access patterns: update vs. read-only but complex queries

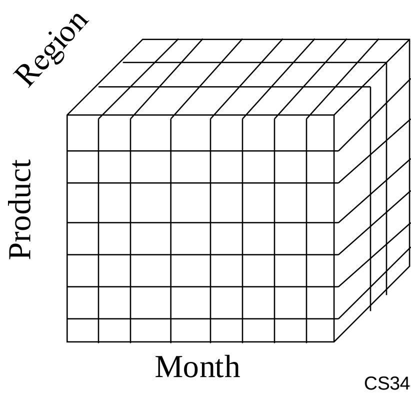
Example of Star Schema

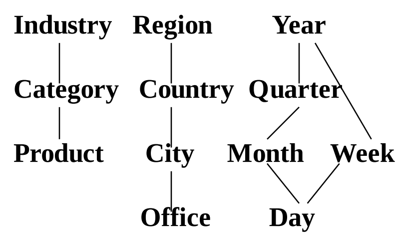


Multidimensional Data

- Sales volume as a function of product, month, and region

Dimensions: Product, Location, Time
Hierarchical summarization paths





CS34800 111