CZ2007 Lecture Notes Part 2

# Structured Query Language

* SQL is a **declarative** language – it specifies what data to retrieve, but not how to do it

## SQL Rules

1. There are reserved words
2. SQL queries are case-insensitive, except for string constants
3. SQL is free-format, i.e. whitespace is ignored
4. Semi-colons are often used as statement terminators, but not required

# Data Manipulation Language (DML)

* Used to perform **queries** and **updates**

## General Format

SELECT [columns]

FROM [tables]

WHERE [condition]

Example:

SELECT name

FROM Products

WHERE manufacturer = ‘Apple’

### SELECT \* – Select All Attributes

SELECT \*

FROM Products

### AS – Rename Attributes

SELECT name AS device

FROM Products

### SELECT [mathematical expressions]

SELECT price\*1.10 AS priceWithTax

FROM Products

### SELECT INTO [table name] – Store Result In Temporary Table

SELECT name

INTO ProductNames

FROM Products

### WHERE Conditions

* **[condition1] AND/OR [condition2]**

SELECT \*

FROM Products

WHERE manufacturer = ‘Apple’ OR manufacturer = ‘Samsung’

* **NOT [condition]**

SELECT \*

FROM Products

WHERE NOT manufacturer = ‘Apple’

* **BETWEEN [value1] AND [value2]**

SELECT \*

FROM Products

WHERE NOT price BETWEEN 700 AND 900

* **IN ([value1], [value2], …)**

SELECT \*

FROM Products

WHERE price IN (750, 850, 950)

* **IS/IS NOT NULL**
  + Note: cannot test NULL using = or <>
  + The logic of SQL conditions uses **3-way logic**: TRUE, FALSE, UNKNOWN
    - When any value is compared to NULL, the result is UNKNOWN
    - A query will only produce a tuple if the result of the WHERE condition is TRUE

SELECT \*

FROM Products

WHERE manufacturer IS NOT NULL

### String Matching Patterns

* [attribute] LIKE/NOT LIKE [pattern]
* Patterns: quoted, case-sensitive string
  + % = any string
  + \_ = any character
  + \ = escape character
* Examples:
  + ‘John%’ – any string beginning with ‘John’
  + ‘%John%’ – any string containing the substring ‘John’
  + ‘\_ \_ \_’ – any three-letter string
  + ‘\_ \_ \_%’ – any string with at least three letters
  + ‘ab\%cd%’ – any string beginning with ‘ab%cd’

SELECT \*

FROM Customers

WHERE name LIKE ‘John%’

### ORDER BY – Order Results as ASC or DESC

* Default order: ASC
* Can order by multiple attributes, in order
* NULL is treated as less than all non-null values

SELECT \*

FROM Customers

ORDER BY age ASC, name DESC

# Multi-Relation Queries

* More than one relation/table can be listed in the FROM clause of a query
* Refer to the attributes as **[relation].[attribute]**
  + Example: Using Preferences(custName, prodName) and FrequentCust(custName, supplName), find the products preferred by at least one customer who is a frequent customer of Apple

SELECT \*

FROM Preferences P, FrequentCust F

WHERE P.custName = F.custName AND F.supplName = ‘Apple’

* Queries can also reference the same relation **more than once**
  + Example: From Products(name, manufacturer), find all pairs of products by the same manufacturer, producing pairs in alphabetical order

SELECT \*

FROM Products P1, Products P2

WHERE P1.manufacturer = P2.manufacturer AND P1.name < P2.name

## Subqueries

* **Scalar**: returns single value
* **Row**: returns one or more columns, single row
* **Table**: returns one or more columns, multiple rows

Example: From Sells(supplName, prodName, price), find the supplier that sells iPhone for the same price Xiaomi charges for the Mi Phone.

SELECT supplName

FROM Sells

WHERE prodName = ‘iPhone’ AND price = (

SELECT price

FROM Sells

WHERE supplName = ‘Xiaomi’ AND prodName = ‘Mi Phone’

)

### IN – Check if Tuple Exists in a Relation

Example: From Products(name, manufacturer) and Preferences(custName, prodName), find the name of each product and manufacturer that Fred likes

SELECT \*

FROM Products

WHERE prodName IN (

SELECT name

FROM Preferences

WHERE custName = ‘Fred’

)

### ALL – Check if Condition is True for all Subquery Tuples

SELECT \*

FROM Sells

WHERE price >= ALL(

SELECT price

FROM Sells

)

### SOME – Check if Condition is True for all Tuples

SELECT \*

FROM Sells

WHERE price >= SOME(

SELECT price

FROM Sells

)

## Correlated vs. Uncorrelated Subqueries

* **Correlated subquery**: references attribute in outer query
  + This means that the subquery has to be computed for every tuple in the outer query
* **Uncorrelated subquery**: subquery is only computed once for entire outer query
  + Most queries can be written so that a subquery is not needed, using JOINs

### EXISTS – Check that Subquery isn’t Empty

Example: From Products(name, manufacturer), find products that are unique to their manufacturer.

SELECT P1.name

FROM Products P1

WHERE NOT EXISTS (

SELECT P2.name

FROM Products P2

WHERE P1.manufacturer = P2.manufacturer AND P1.name <> P2.name

)

## Subquery Rules

1. **Cannot use ORDER BY** in a subquery
2. The number of attributes in the SELECT of the subquery must **match the number of attributes** compared to in the outer query
3. Column names refer to columns of the **relations in the subquery FROM clause**, by default
4. Result of subquery must be used as the **right operand**

# Set Operations

* (subquery) **UNION** (subquery)
* (subquery) **INTERSECT** (subquery)
* (subquery) **EXCEPT** (subquery)
* Set operations remove duplicates by default
* The two subqueries must have the same attributes in the SELECT clause

# Bag Semantics

* **Set semantics** remove duplicates
  + UNION/INTERSECT/EXCEPT follow set semantics by default
  + To force bag semantics, UNION/INTERSECT/EXCEPT **ALL**
  + Example:

(SELECT \*

FROM Preferences)  
UNION ALL

(SELECT \*

FROM Sells, FrequentCust

WHERE FrequentCust.supplName = Sells.supplName)

* **Bag semantics** allow duplicates
  + Default SELECT-FROM-WHERE follow bag semantics
  + To force set semantics, SELECT **DISTINCT**
  + Example:

SELECT DISTINCT name

FROM Products

# Joins

* By default, joins use bag semantics (allow duplicates)
* Types:
  + **NATURAL JOIN** – joins on same column name, removes duplicate column
  + **CROSS JOIN** – all possible combinations

### Theta Join

* [Table1] **JOIN** [Table2] **ON** [condition]
  + Example: Preferences JOIN Products ON productName = name
* [Table1] **INNER** **JOIN** [Table2] **USING** (list of attributes)
  + Example: Preferences INNER JOIN FrequentCust USING (name)

### Outer Join

* [Table1] **LEFT/RIGHT/FULL (NATURAL) OUTER JOIN** [Table2] **ON** [conditions]
  + LEFT: Table1’s empty fields are padded with NULL values
  + RIGHT: Table2’s empty fields are padded with NULL values
  + FULL: Both relations’ empty fields are padded with NULL values
* Example: Preferences LEFT OUTER JOIN FrequentCust ON name = custName

# Aggregation

* There are five aggregate functions:
  1. **COUNT** – number of values in a column
     + **COUNT(\*)** returns the number of rows in the table
  2. **SUM** – sum of values in a column
  3. **AVG** – average of values in a column
  4. **MIN** – smallest value in a column
  5. **MAX** – largest value in a column
* All functions ignore NULL values, except COUNT(\*)
* Use DISTINCT to remove duplicates

Examples:

SELECT AVG(price)

FROM Sells

WHERE supplName = ‘Apple’

SELECT COUNT(DISTINCT price)

FROM Sells

WHERE prodName = ‘iPhone’

SELECT COUNT(\*)

FROM Products

# Grouping

* **GROUP BY [attributes]**
* When using grouping, the only attributes that can be in the SELECT clause are the GROUP BY attributes or aggregate functions

SELECT prodName, AVG(price)

FROM Sells

GROUP BY prodName

* **HAVING [condition]** can follow a GROUP BY clause, to eliminate groups that do not meet the condition
  + Conditions can refer to any relation in the FROM clause
  + Must refer to grouping attributes or aggregated values

Example: Using Sells(prodName, supplName, price), find the average price of products that are sold by at least two suppliers

SELECT prodName, AVG(price) AS avgPrice

FROM Sells

GROUP BY prodName

HAVING COUNT(supplName) >= 2

* SQL99 introduced two new functions:
  + **EVERY [condition]** – every row in the group must satisfy the condition
  + **ANY [condition]** – any row in the group must satisfy the condition

# Creation of Tables

## Data Definition Language (DDL)

* Add/modify/drop tables
* Create views
* Define and enforce integrity constraints
* Enforce security restrictions

## Creating/Dropping Tables

CREATE TABLE [TableName]

(

[attribute declarations]

);

DROP TABLE [TableName];

### Data Types

* **INT/INTEGER**
* **REAL/FLOAT**
* **CHAR(n)** – fixed-length string of length n
* **VARCHAR(n)** – variable-length string up to length n
* **DATE** – ‘yyyy-mm-dd’ format
* **TIME** – ‘hh:mm:ss’ format

### Adding/Deleting Attributes

* **ALTER TABLE** [TableName] **ADD** [attribute declarations]
* **ALTER TABLE** [TableName] **DROP** [attribute name]

## Database Modifications

### Insertion

**INSERT INTO [TableName] VALUES (list of values);**

Example: INSERT INTO Preferences VALUES (‘Sally’, ‘iPhone’);

**INSERT INTO [TableName] (SQL query)**

Example:

INSERT INTO Sells(prodName) (

SELECT prodName

FROM Products

WHERE prodName NOT IN (SELECT prodName FROM Sells)

);

### Deletion

**DELETE FROM [TableName] WHERE [conditions];**

Example:

DELETE FROM Sells

WHERE EXISTS (

SELECT prodName

FROM Sells

WHERE price > 1000

);

### Update

**UPDATE [TableName]**

**SET [attributes]**

**WHERE [conditions];**

Example:

UPDATE Customers

SET phoneNumber = ‘123456’

WHERE name = ‘Fred’;

# Views

* Views are virtual, dynamic results of queries over base relations
* They are generated at runtime, and queried over the same way as base tables

### Creating Views

**CREATE VIEW [ViewName] AS [SQL query];**

Example:

CREATE VIEW FrequentCustProd AS

SELECT custName, prodName

FROM FrequentCust F, Sells S

WHERE F.supplName = S.supplName;

### Querying Views

* Views can be queried like normal SQL tables
* At compile-time, the DBMS maps queries on views into a query on their corresponding base relation

Example:

SELECT \*

FROM FrequentCustProd

WHERE custName = ‘Sally’

### Updating Views

* Views can only be updated if the data can propagate directly to its base relations without any ambiguity – i.e. the view is derived from a **single relation**
  + Views defined with **JOINs** (on multiple relations) cannot be updated
  + Views defined with **aggregate functions or groups** cannot be updated

### Temporary Views

**WITH [ViewName] AS (SQL query)**

* Introduced in SQL99, makes complex queries easier to read and allows the same view to be used multiple times in a query
* Only supported by some DBMSs

Example:

WITH FrequentCustProd AS

(SELECT custName, prodName

FROM FrequentCust F, Sells S

WHERE F.supplName = S.supplName;)

SELECT \*

FROM FrequentCustProd

WHERE custName = ‘Sally’

# Constraints

## Key Constraints

### PRIMARY KEY/UNIQUE Attributes

* **PRIMARY KEY** – attribute is primary key of table
  + For single-attribute keys, declare after data type: **[attribute] [data type] PRIMARY KEY**
  + For multi-attribute keys, declare separately: **PRIMARY KEY([attr1], [attr2])**
* **UNIQUE** – attribute value must be unique to table
* There can only be one primary key, but multiple unique attributes, per table
* Primary keys can never be NULL, while unique attributes can

### NOT NULL/DEFAULT Values

* **NOT NULL** – attribute value cannot be null
* **DEFAULT [value]** – attribute value, if null, defaults to [value]

### Foreign Keys

* **FOREIGN KEY [keyName] REFERENCES [tableName(attributes)]**
  + Means that values of keyName must exist in referenced table and attributes
  + Referenced attributes must be declared as primary keys
* INSERT: rejected if values don’t exist in referenced table
* ON DELETE/UPDATE: if the value in a referenced table is being deleted or updated, three options:
  + **DEFAULT**: reject delete/update
  + **CASCADE**: changes ripple to foreign key
  + **SET NULL**: corresponding foreign key values are set to NULL

Example:

CREATE TABLE Sells(

prodName CHAR(20) PRIMARY KEY,

prodID CHAR(10) UNIQUE NOT NULL,

supplName CHAR(30) NOT NULL,

price REAL DEFAULT ‘100’,

FOREIGN KEY prodName REFERENCES Products(name)

ON DELETE SET NULL

ON UPDATE CASCADE

);

## Attribute-Based Checks

**[attribute] [data type] CHECK (condition)**

* Can check against other relations/attributes, but only in subqueries
* Condition is only checked during INSERT and UPDATEs

Example:

CREATE TABLE Sells(

prodName CHAR(20) PRIMARY KEY,

supplName CHAR(30) CHECK (supplName IN (SELECT name FROM Suppliers)),

price REAL CHECK (price <= 1800)

);

## Tuple-Based Checks

**CHECK (condition)**

* Can be defined on multiple attributes in the relations
* Condition is only checked during INSERT and UPDATEs

Example:

CREATE TABLE Sells(

prodName CHAR(20) PRIMARY KEY,

supplName CHAR(30),

price REAL,

CHECK (supplName = ‘Apple’ OR price < 1000)

);

## Assertions

**CREATE ASSERTION [name]**

**CHECK (condition)**

* Can be defined on any attributes in any relations in the database schema
* Checked after any modification (INSERT/UPDATE/DELETE) to the relation

Example: In Sells, no supplier should charge an average price over $1000.

CREATE ASSERTION NoRipOffSuppliers

CHECK (

NOT EXISTS (

SELECT supplName

FROM Sells

GROUP BY supplName

HAVING AVG(price) > 1000

));

# Triggers

* Allows user to specify when the checking of a condition occurs
* When a trigger can be executed: BEFORE, AFTER, INSTEAD OF (view modification)
  + BEFORE triggers aren’t allowed to modify the database – must either accept or abort the triggering event
* Trigger events: INSERT, UPDATE, DELETE
  + When an event occurs, the trigger tests the condition
  + If the condition is satisfied, the trigger action is performed
    - Trigger actions: INSERT, UPDATE, DELETE, ROLLBACK
* Granularity: FOR EACH ROW or STATEMENT
  + Row-level triggers are executed once per modified tuple
  + Statement-level triggers are executed once per SQL statement (regardless of how many tuples are modified)

**CREATE TRIGGER [name]**

**[BEFORE|AFTER|INSTEAD OF] [INSERT|UPDATE OF [attribute]|DELETE] ON [relation]**

**REFERENCING [OLD|NEW ROW|TABLE] AS [attributeName]**

**FOR EACH [ROW|STATEMENT]**

**WHEN (conditions)**

**[actions]**

Examples:

CREATE TRIGGER LimitSalaryRaise

AFTER UPDATE OF salary ON Employee

REFERENCING OLD ROW AS o

NEW ROW AS n

FOR EACH ROW

WHEN (n.salary – o.salary > 0.05 \* o.salary)

UPDATE Employee

SET salary = 1.05 \* o.salary

WHERE id = o.id;

CREATE TRIGGER RoomCapacityCheck

BEFORE INSERT ON Transcript

REFERENCING NEW ROW AS n

FOR EACH ROW

WHEN (

(SELECT COUNT(T.studentId) FROM Transcript T WHERE T.course = n.course AND T.semester = n.semester)

>=

(SELECT L.limit FROM CourseLimits L WHERE L.courseId = n.course AND L.semester = n.semester)

)

**ROLLBACK**;

### Triggers on Views

* Views are virtual queries; INSTEAD OF triggers can be used to insert/update underlying relations instead of views themselves

Example:

CREATE VIEW Synergy AS

SELECT L.custName, L.prodName, S.supplName

FROM Likes L, Sells S, FrequentCust F

WHERE L.custName = F.custName

AND L.prodName = S.prodName

AND S.supplName = F.supplName

CREATE TRIGGER ViewTrigger

INSTEAD OF INSERT ON Synergy

REFERENCING NEW ROW AS n

FOR EACH ROW

BEGIN

INSERT INTO Likes VALUES(n.custName, n.prodName)

INSERT INTO Sells VALUES(n.prodName, n.supplName)

INSERT INTO FrequentCust VALUES(n.custName, n.supplName)

END;

# Speeding Up Queries

* Data is stored in **pages** – pieces of contiguous information in files (typically 4096 bytes)
* Disk and main memory exchange a page of data each time
* Disks are divided into a sequence of page-sized **blocks** for storage
  + Reading a block costs one time unit
  + Writing a block costs two time units (retrieve and write data)
* Tuples are stored unordered

## Indexes

**CREATE INDEX [name] ON [tableName](attributes)**

* Tree-structured, hash-based data structure that allows data to be found more quickly
* Useful for range queries and join queries
* **Order matters** in multi-attribute indexes
  + Example: CREATE INDEX doubleIndex ON Customer(age, city, name)
    - Useful for queries on age
    - Useful for queries on (age, city)
    - Useful for queries on (age, city, name)
    - Not useful for queries on city or name
* Pros: can speed up queries and joins
* Cons: makes insert/delete/updates more complex and time-consuming

# Embedded SQL

* Some steps of conventional host languages require SQL statements
* A preprocessor turns SQL statements into host-language code procedure calls
* All embedded SQL statements begin with **EXEC SQL**
  + Any SQL statements that don’t return a result (INSERT, UPDATE, DELETE) can be embedded directly
  + SELECT statements must use SELECT INTO a shared variable

## Shared Variables

* Variables that appear in both host language and SQL statements
  + Must be types that both languages can deal with
* To use a shared variable in a SQL statement, must be preceded with a colon

Example: Insert

EXEC SQL BEGIN DECLARE SECTION;

float price;

char prodName[30], supplName[20];

EXEC SQL END DECLARE SECTION;

EXEC SQL INSERT INTO Sells VALUES(:prodName, :supplName, :price);

Example: Select

EXEC SQL BEGIN DECLARE SECTION;

char prodName[30];

float avgPrice;

EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT AVG(price)

INTO :avgPrice

FROM Sells

WHERE prodName = :prodName;

## SQLSTATE

* Array of 5 characters, that indicates if any problems occurred during execution of SQL statements
  + ‘00000’ – no error
  + ‘02000’ – a requested tuple was not found

# Cursors

* Declare cursor:

**EXEC SQL DECLARE [cursorName] CURSOR FOR [SQL query];**

* Open/close cursor:

**EXEC SQL OPEN CURSOR [cursorName];**

**EXEC SQL CLOSE CURSOR [cursorName];**

* Fetch next tuple:

**EXEC SQL FETCH [cursorName] INTO [variables];**

* Cursors can also be used to update or delete the current tuple:

**EXEC SQL DELETE FROM [table]**

**WHERE CURRENT OF [cursorName];**

# Dynamic SQL

* Allows queries to be read and executed at run-time, instead of compile-time
* To prepare the query:

**EXEC SQL PREPARE [queryName] FROM [query text];**

* To execute the query:

**EXEC SQL EXECUTE [queryName];**

* If only using the query once, can combine the two steps:

**EXEC SQL EXECUTE IMMEDIATE [query text];**

# Persistent Stored Modules (PSM)

* Stores procedures and functions as database schema elements
  + Programs that are executed in a single SQL statement – allows application logic to be performed directly at database server

### Parameters

* **[mode] [name] [data type]**
* Modes:
  + **IN** – procedure uses value, does not change it
    - Function parameters may only be of type IN
  + **OUT** – procedure does not use value, changes it
  + **INOUT** – both used and updated

### Local Declarations

* **DECLARE [name] [type]** – declares a local variable
* **SET [name] = [expression]** – assigns a value to a variable

## Stored Procedures

**CREATE PROCEDURE (**

**[parameter list]**

**)**

**[local declarations (optional)]**

**[body]**

Example: Write a procedure that takes two arguments b and p, and adds a tuple to Sells with supplName = ‘Gangnam’, prodName = b and price = p.

CREATE PROCEDURE AddGangnam (

IN b CHAR(50),

IN p REAL

)

INSERT INTO Sells

VALUES(‘Gangnam’, b, p);

## Functions

**CREATE FUNCTION (**

**[parameter list]**

**)**

**RETURNS [return value data type]**

**[local declarations (optional)]**

**[body]**

Example: Write a function that takes in product name and supplier and returns the price.

CREATE FUNCTION GetPrice (

IN pName CHAR(50)

IN sName CHAR(50)

)

RETURNS REAL

DECLARE p REAL;

BEGIN

SELECT price INTO p

FROM Sells

WHERE prodName = pName AND supplName = sName;

RETURN p;

END;

### Invoking Stored Procedures

**EXEC SQL CALL [stored procedure name];**

* Can only be used to invoke stored procedures
* Both functions and stored procedures can be called from within other procedures, functions, or triggers
* Only functions can be called in SELECT statements in the WHERE or HAVING clause

## Conditional Statements

**IF [condition] THEN**

**[action]**

**ELSEIF [condition] THEN**

**[action]**

**ELSE**

**[action]**

**END IF;**

## Loops

**[name]: LOOP**

**[actions]**

**LEAVE [name];**

**END LOOP;**

## Cursors

**DECLARE NotFound CONDITION FOR SQLSTATE ‘02000’;**

**DECLARE [name] CURSOR FOR (SQL query);**

**BEGIN**

**OPEN [name];**

**listLoop: LOOP**

**FETCH [name] INTO [variables];**

**IF NotFound THEN LEAVE listLoop END IF;**

**[actions]**

**END LOOP;**

**CLOSE [name];**

**END;**

# Call-Level Interface

* **Statement**: accepts SQL string to run as a query
* **PreparedStatement**: object containing a SQL statement to execute
* Functions:
  + executeQuery(Q)
  + executeQuery()
  + executeUpdate(U)
  + executeUpdate()
* Accessing results:
  + next() – advance cursor
  + get[type](i) – type is type of variable, i is component index in SELECT statement

# Extensible Markup Language (XML)

* Every element is defined by a start and end tag
  + Every start tag must have a matching end tag
  + Elements cannot overlap
  + There can only be one root element
* Attributes are name-value pairs separated by an equals sign
  + All attributes in an element must have a unique name
    - Names may contain letters, numbers, underscore \_, hyphen -, or period .
  + Values are enclosed in single or double quotes
  + No un-escaped < or & may appear in character data
* **XML schema**: used for agreeing on data format and type
* **Document Type Definitions (DTDs)**: specify the constraints that any document that is an instance of its type must satisfy

## Element Declaration

**<!ELEMENT [name]([model])>**

* Defines the child elements that make up a parent element
  + **name**: name of element being declared
  + **model**: the allowed contents for the element

### , Operator (Sequence)

**<!ELEMENT book(title, author)>**

* Order of elements in a book must be title, then author

### | Operator (XOR)

**<!ELEMENT book(title|author)>**

* Books must contain a title or author, but not both

### \* Operator (Repetition)

**<!ELEMENT book(review\*)>**

* Books can contain zero or more reviews

### + Operator

**<!ELEMENT book(title, author+)>**

* Books must contain a title followed by one or more authors

### ? Operator

**<!ELEMENT book(bestseller?)>**

* Books can contain zero or one bestseller element

### ANY

**<!ELEMENT book ANY>**

* Books can contain any valid tags or character data

### #PCDATA

**<!ELEMENT book (#PCDATA)>**

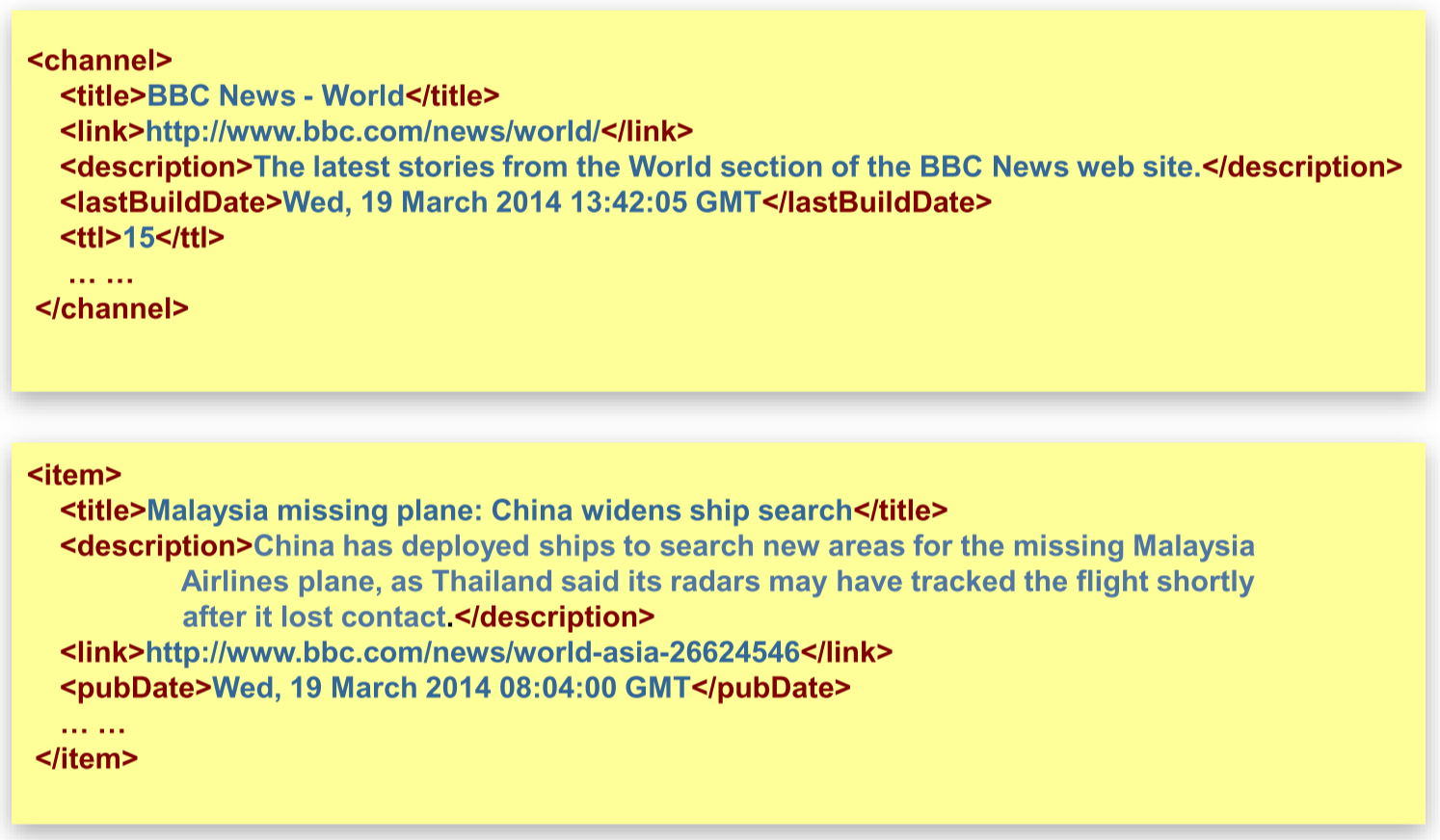
* Books can only contain character data

### EMPTY

**<!ELEMENT book EMPTY>**

* No content allowed

Example:



<!ELEMENT rss(channel)>

<!ELEMENT channel(title, link, description, lastBuildDate?, ttl?, item+)>

<!ELEMENT item(title, description, link?, pubDate?)>

<!ELEMENT title(#PCDATA)>

<!ELEMENT link(#PCDATA)>

<!ELEMENT description(#PCDATA)>

<!ELEMENT lastBuildDate(#PCDATA)>

<!ELEMENT ttl(#PCDATA)>

<!ELEMENT pubDate(#PCDATA)>

## Attribute Declaration

**<!ATTLIST [element-name] [attribute-name] [attribute-type] [default-value]>**

* Attributes can be required or not required
* They may have default values

### Attribute Types

* **CDATA** – any valid character data
* **ID** – an identifier that is unique to the document
* **IDREF** – a reference to a unique ID
* **IDREFS** – a reference to several unique IDs
* **(a|b|c)** – possible values (a or b or c)

### Default Values

* **#REQUIRED** – attribute must be present
* **#IMPLIED** – attribute is optional
* **#FIXED “x”** – attribute is optional, but if present, must be “x”
* **“x”** – value is “x” if not specified

Example:

<!ATTLIST oldjoke name ID #REQUIRED label CDATA #IMPLIED status (funny| notfunny) “funny”>

## Entities

* Entities are physical units such as characters, strings, or files
* They are used for:
  + References to non-keyboard characters
  + Abbreviations for frequently-used strings
  + Documents that need to be broken up into multiple parts

### Entity Declaration

**<!ENTITY [name] [value]>**

Entity reference: **&[name]** substitutes value of entity

Example:

<!ENTITY NTU “Nanyang Technological University”>

Entity reference: &NTU

## Referencing a DTD

* Internal: **<!DOCTYPE [element-name][ [declaration] ]>**
* External: **<!DOCTYPE [element-name] SYSTEM “[DTD file location]”>**

## Mapping Relational Data to XML

Example:



**Product view:**

<db>

<product>

<name></name>

<weight></weight>

<manufacturer>

<name></name>

<address></address>

</manufacturer>

<manufacturer>

<name></name>

<address></address>

</manufacturer>

</product>

</db>

**Company view:**

<db>

<company>

<name></name>

<address></address>

<product>

<name></name>

<weight></weight>

</product>

<product>

<name></name>

<weight></weight>

</product>

</company>

</db>

# JavaScript Object Notation (JSON)

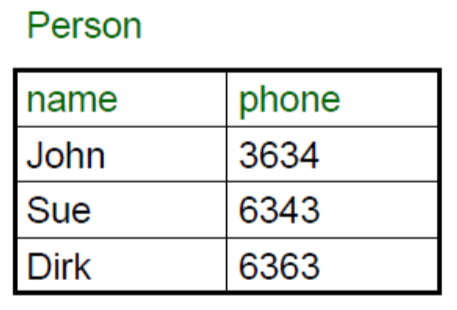
* Data made up of objects, lists, and atomic values (integers, floats, strings, Booleans)

### JSON Data Types

* Number
* String (double-quoted)
* Boolean
* Empty/null

## Mapping Relational Data to JSON

Example:



{

“person”: [

{“name”: “John”, “phone”: 3634},

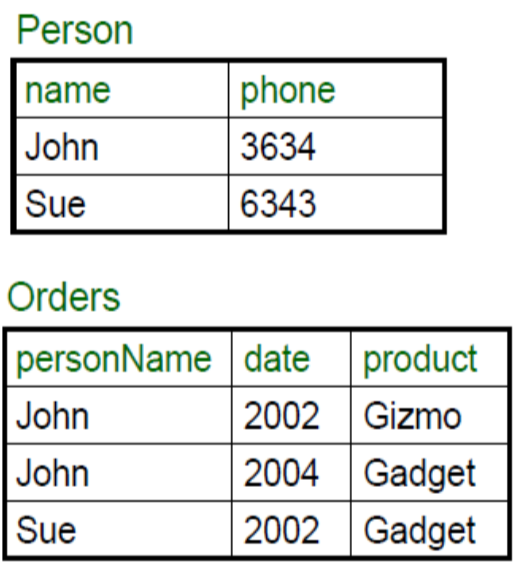
{“name”: “Sue”, “phone”: 6343},

{“name”: “Dirk”, “phone”: 6363}

]

}

Example: Inline foreign keys



{

“person”: [

{

“name”: “John”,

“phone”: 3634,

“Orders”: [

{“date”: 2002, “product”: “Gizmo”},

{“date”: 2004, “product”: “Gadget”}

],

},

{

“name”: “Sue”,

“phone”: 6343,

“Orders”: [

{“date”: 2002, “product”: “Gadget”}

]

}

]

}