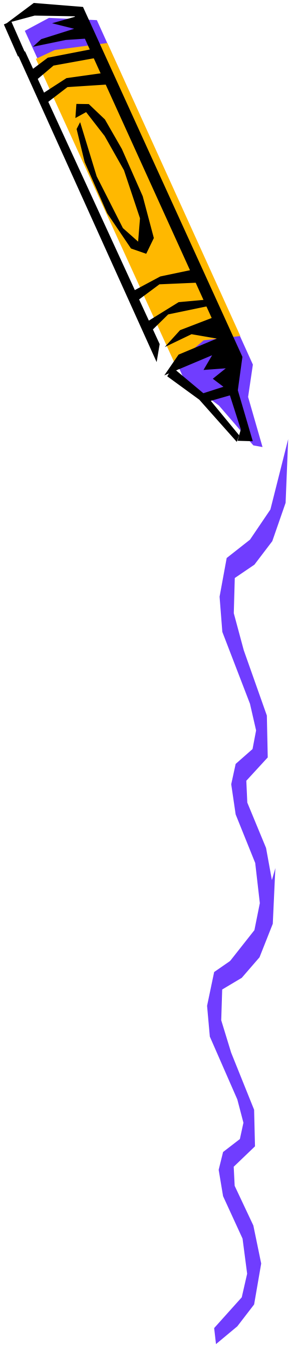
Relational DB, SQL, Efficient Design & JDBC

CSC 631/831, Spring 2013

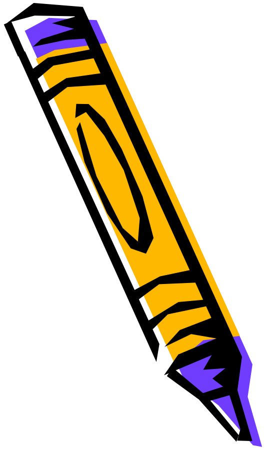
Dr. Ilmi Yoon

Topics Covered



- Database Design
- Normalization
- De-Normalization
- Primary key, indexing
- SQL
- Stored Procedures
- JDBC

Database Design



- The process of producing a detailed data model of a database.
- Logical design of the base data structures used to store the data.
- Accurate design is crucial to the operation of a reliable and efficient information system.
- How data is stored and how that data is related.

Database Design

Problems Resulting from Poor Design

- The database and/or application may not function properly.
- Data may be unreliable or inaccurate.
- Performance may be degraded.
- Flexibility may be lost.

Database Design

The process of doing database design generally consists of a number of steps which will be carried out by the database designer:

- Determine the purpose of your database
- Find and organize the information required
- Divide the information into tables
- Turn information items into columns

The design process consists of the following steps:

1)**Determine the purpose of your database** - This is the simple process which helps you to decide what functionality you need from your application.

The first method for planning for a database is to simply brainstorm, on paper or otherwise, concerning what the database will need to store, and what the application will need out of it.

The goal is to start with a general and complete view, and narrow down.

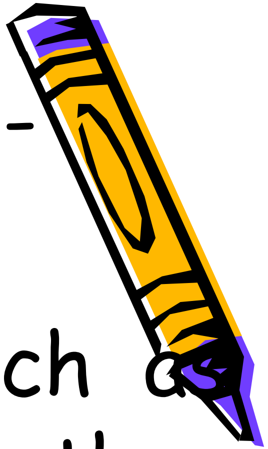
2) Find and organize the information required -

Collect all of the types of information you might want to record in the database, such as user's information and product ID.

In web applications like a online video store is necessary to store the customer id, the information about the membership duration, membership charges etc.

The type of information you want to save in the database entirely depends in the application you are developing.

3) Divide the information into tables - Divide your information items into major entities or subjects, such as Products or Orders. Each subject then becomes a table.



Example:

A student tracking database would probably include the following entities:

Students – who is the database keeping track of

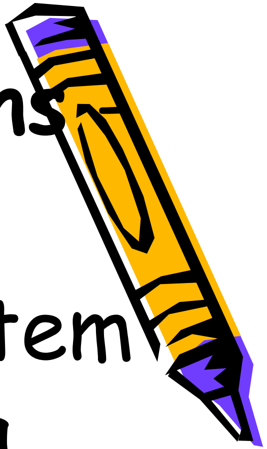
Courses – which courses are available

Classes – which classes are available

Instructors – who is teaching the courses

Schedules – putting students into classes

4) Turn information items into columns Decide what information you want to store in each table. Each item becomes a field, and is displayed as a column in the table.



The students table would include:

Student ID

Last name

First name

Address

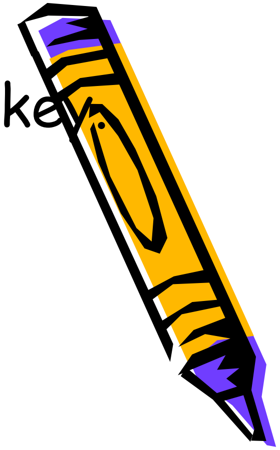
City

State

Zip

5) Specify primary keys - Choose each table's primary key.

The primary key is a column that is used to uniquely identify each row.



An example might be Product ID or Order ID.

A **foreign key** is a referential constraint between two tables.[

Say we have two tables, a CUSTOMER table that includes all customer data, and an ORDER table that includes all customer orders. The intention here is that all orders must be associated with a customer that is already in the CUSTOMER table. To do this, we will place a foreign key in the ORDER table and have it relate to the primary key of the CUSTOMER table.

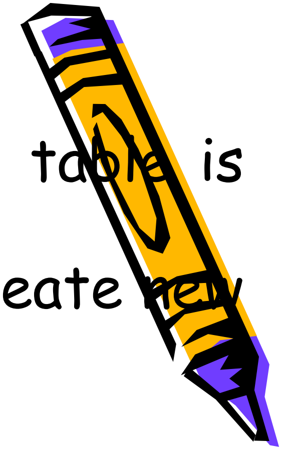
6) Set up the table relationships -

Look at each table and decide how the data in one table is related to the data in other tables. Add fields to tables or create new tables to clarify the relationships, as necessary.

The relationships can be developed between entities by looking at common data. Relationships fall into three basic categories:

One to one, One to many, Many to many

After the set up of different entities for each subject in the database, you need a way of telling the database how to bring that information back together again. The first step in this process is to define relationships between your entities.



A relationship works by matching data in key attributes. In most cases, matching attributes are the primary key from one table, which provides a unique identifier for each record, and a foreign key in the other table. For example, key attributes such as the student ID, course ID, and class ID can relate student, class, and course entities.

- ***A one-to-many relationship***

  A one-to-many relationship is the most common type of relationship. In a one-to-many relationship, an entity in Table A can have many matching entities in Table B, but a entity in Table B has only one matching entity in Table A.

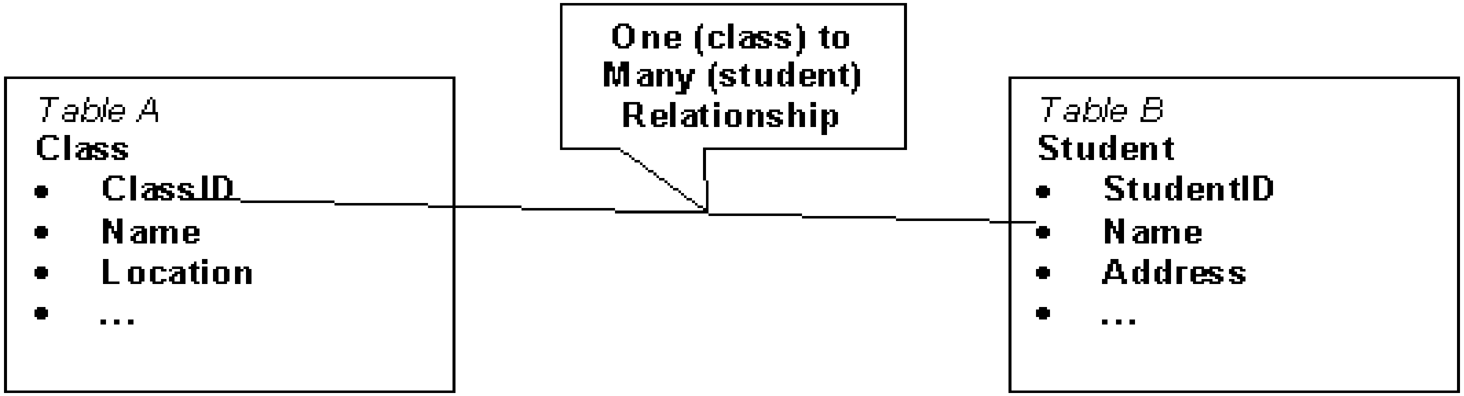


Table A
Class
- **ClassID**
- **Name**
- **Location**
- **...**

One (class) to Many (student) Relationship

Table B
Student
- **StudentID**
- **Name**
- **Address**
- **...**

- **A many-to-many relationship**

In a many-to-many relationship, an entity in Table A can have many matching entities in Table B, and a record in Table B can have many matching entities in Table A. This type of relationship is only possible by defining a third entity (called a junction) whose primary key consists of two attributes including the foreign keys from both Tables A and B. A many-to-many relationship is really two one-to-many relationships with a third entity.

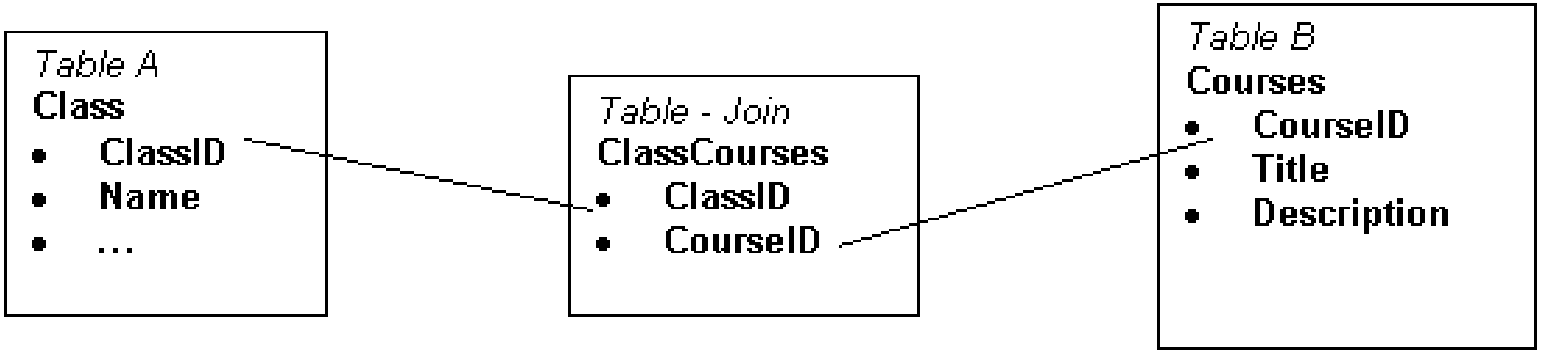


Table A
Class
- **ClassID**
- **Name**
- **...**

Table - Join
ClassCourses
- **ClassID**
- **CourseID**

Table B
Courses
- **CourseID**
- **Title**
- **Description**

A one-to-one relationship:

In a one-to-one relationship, each record in Table A can have only one matching entity in Table B, and each record in Table B can have only one matching entity in Table A. This type of relationship is not common, because most information related in this way would be in one entity. You might use a one-to-one relationship to divide a table with many attributes, to isolate part of a table for security reasons, or to store information that applies only to a subset of the main entity.

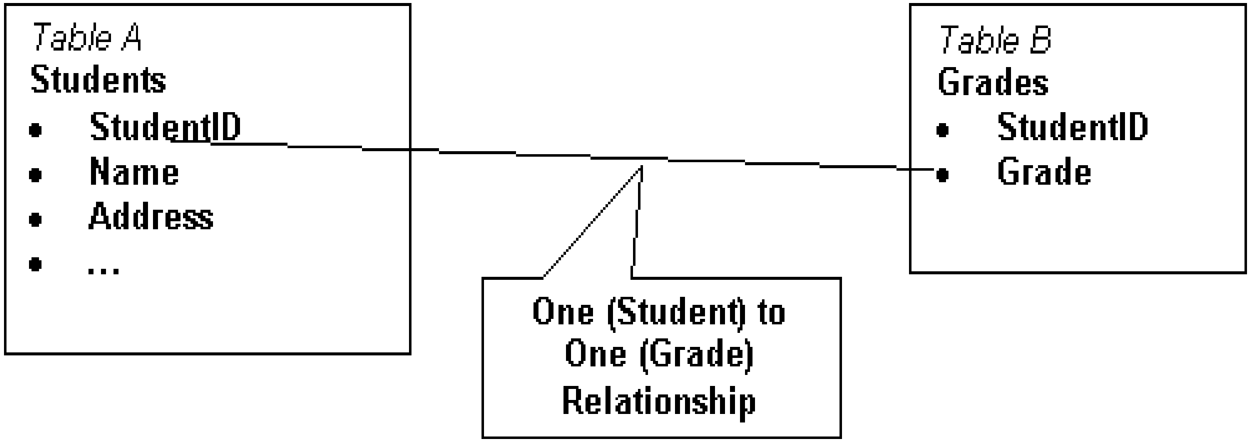
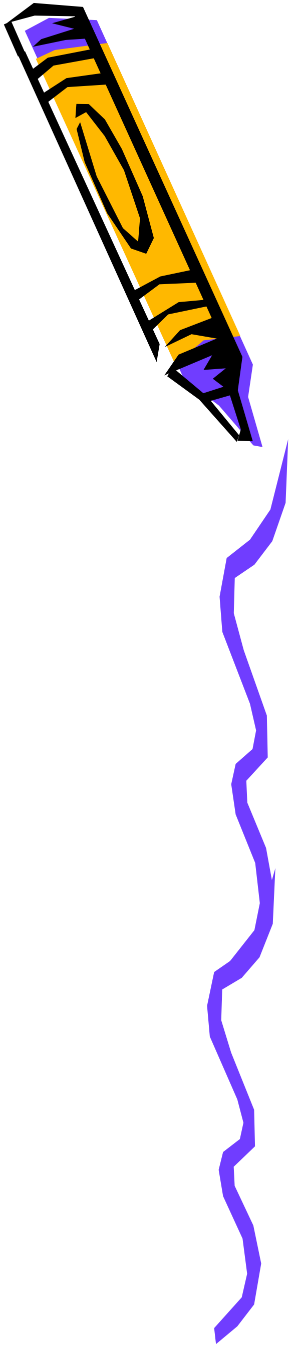


Table A
Students
- StudentID
- Name
- Address
- ...

Table B
Grades
- StudentID
- Grade

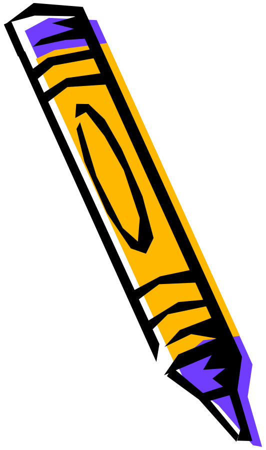
One (Student) to One (Grade) Relationship

Refine the Design



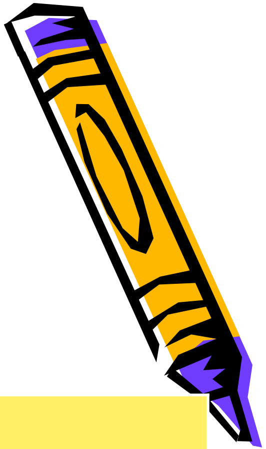
- Check primary keys
- Check the table relationships
- Apply the normalization rules

Un-normalized Design



- Redundant Data
- Modification Anomalies
  - Update Anomaly
  - Deletion Anomaly
  - Insertion Anomaly

Redundant Data



Prod ID	Description	Supplier	Address	City	Region	Country
34	Sasquatch Ale	Bigfoot Breweries	3400 - 8th Avenue	Bend	OR	USA
27	Schoggi Schokolade	Heli Süßwaren GmbH	Tiergarten straße 5	Berlin		Germany

Suppose you wanted to add another Item for same supplier Bigfoot Breweries?

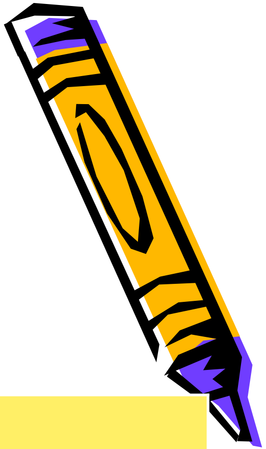
37	**Lumberman 's Lager**	**Bigfoot Breweries**	**3400 - 8th Avenue**	**Bend**	**OR**	**USA**

Update Anomaly

Prod ID	Description	Supplier	Address	City	Region	Country
34	Sasquatch Ale	Bigfoot Breweries	3400 - 8th Avenue	Bend	OR	USA
27	Schoggi Schokolade	Bigfoot Breweries	3400 - 8th Avenue	Bend	OR	USA

- Imagine the issues surrounding modifications of hundreds of rows of data for one supplier.

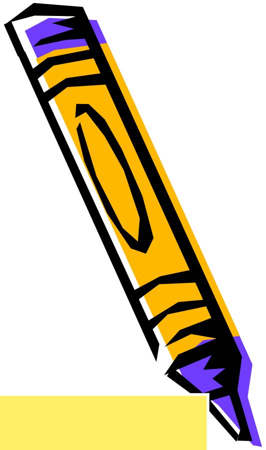
Deletion Anomaly



Prod ID	Description	Supplier	Address	City	Region	Country
34	Sasquatch Ale	Bigfoot Breweries	3400 - 8th Avenue	Bend	OR	USA
27	Schoggi Schokolade	Heli Süßwaren GmbH	Tiergarten straße 5	Berlin		Germany

- We decide to delete the row 34 (the only item from Bigfoot).
- A deletion anomaly means that we lose more information than we want.

Insertion Anomaly



Prod ID	Description	Supplier	Address	City	Region	Country
34	Sasquatch Ale	Bigfoot Breweries	3400 - 8th Avenue	Bend	OR	USA
27	Schoggi Schokolade	Heli Süßwaren GmbH	Tiergarten straße 5	Berlin		Germany
??	?????	StarStruck	101 Mariposa	Seattle	WA	USA

- You want to add a new supplier, StarStruck (no specific item yet).

Normalization

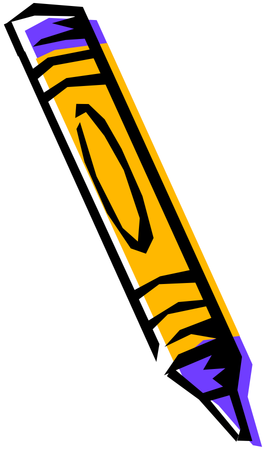
- The process of organizing data to minimize redundancy is called **normalization.**
- Edgar F. Codd, the inventor of the relational model, introduced the concept of normalization.
  - First Normal Form
  - Second Normal Form
  - Third Normal Form
  - Boyce Codd Normal Form
  - Fourth Normal Form
  - Fifth Normal Form
  - Sixth Normal Form

- **Apply the normalization rules** - Apply the data normalization rules to see if your tables are structured correctly. Make adjustments to the tables

Database normalization is the process of organizing the fields and tables of a relational database to minimize redundancy and dependency.

Normalization usually involves dividing large tables into smaller (and less redundant) tables and defining relationships between them.

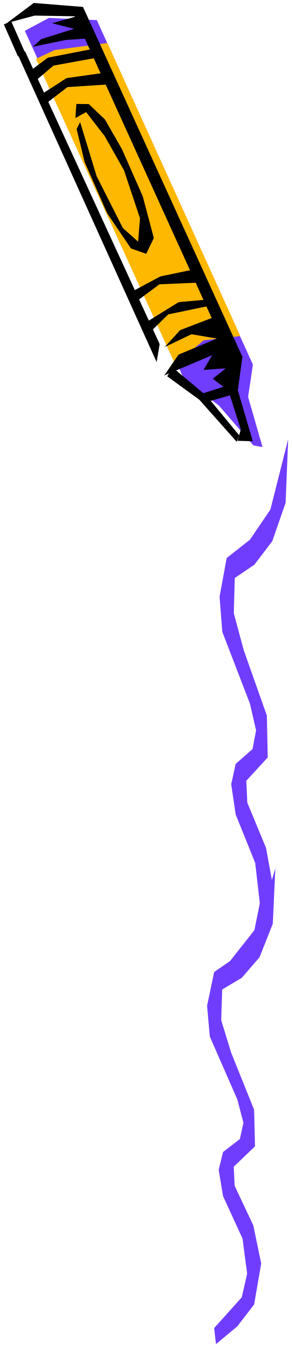
Objectives of Normalization



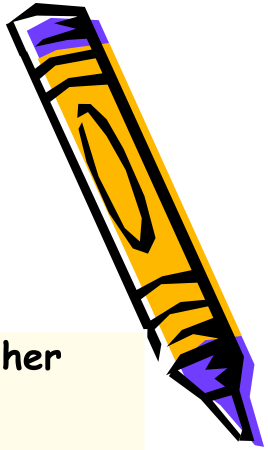
- To permit data to be queried.
- To free insertion, update and deletion dependencies.
- To reduce the need for restructuring.
- To make the data model more informative to users.
- To make the collection of relations neutral to the query statistics.

First Normal Form

- Table has a primary key
- Table has no repeating groups



Let us consider a table:



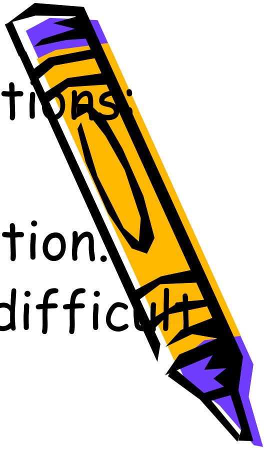
Title	Author1	Author2	ISBN	Subject	Pages	Publisher
Database System Concepts	Abraham Silberschatz	Henry F. Korth	0072958863	MySQL, Computers	1168	McGraw-Hill
Operating System Concepts	Abraham Silberschatz	Henry F. Korth	0471694665	Computers	944	McGraw-Hill

After having the look at the table, we found that this table has some problems. The problems include that the table does not scale well, it does not provide data integrity and it is not efficient with storage.

According to the **First Normal Form** , the table has two violations:

- The table has more than one author field,
- The Subject field contains more than one piece of information.

With more than one value in a single field, it would be very difficult to search for all books on a given subject.



So we refine the table as:

Title	**Author**	**ISBN**	**Subject**	**Pages**	**Publisher**
Database System Concepts	Abraham Silberschatz	0072958863	MySQL	1168	McGraw-Hill
Database System Concepts	Henry F. Korth	0072958863	Computers	1168	McGraw-Hill
Operating System Concepts	Henry F. Korth	0471694665	Computers	944	McGraw-Hill
Operating System Concepts	Abraham Silberschatz	0471694665	Computers	944	McGraw-Hill

We, now have two rows of the same book which means we are violating the **second form.**

Second Normal Form:

- Table must be in First Normal Form
- Remove vertical redundancy: The same value should not repeat across rows

A better solution to the problem would be to separate the data into separate tables- an Author table and a Subject table to store our information, removing that information from the Book table:

Subject table:

Subject_ID	Subject
1	MySQL
2	Computers

Author table:

Author_ID	Last Name	First Name
1	Silberschatz	Abraham
2	Korth	Henry

Book Table:

ISBN	Title	Pages	Publisher
0072958863	Database System Concepts	1168	McGraw-Hill
0471694665	Operating System Concepts	944	McGraw-Hill

Each table has a primary key, used for joining tables together when querying the data. A primary key value must be unique with in the table (no two books can have the same ISBN number), and a primary key is also an index, which speeds up data retrieval based on the primary key.

Now to define relationships between the tables:

Book_author Table:

ISBN	Author_ID
0072958863	1
0072958863	2
0471694665	1
0471694665	2

Book_subject Table:

ISBN	Subject_ID
0072958863	1
0072958863	2
0471694665	2

- As the First Normal Form deals with redundancy of data across a horizontal row, Second Normal Form (or 2NF) deals with redundancy of data in vertical columns.
- The normal forms are progressive, so to achieve Second Normal Form, the tables must already be in First Normal Form.
- The Book Table will be used for the 2NF example

Book table:

ISBN	Title	Pages	Publisher_ID
0072958863	Database System Concepts	1168	1
0471694665	Operating System Concepts	944	1

Publisher Table:

Publisher_ID	Publisher Name
1	McGraw-Hill

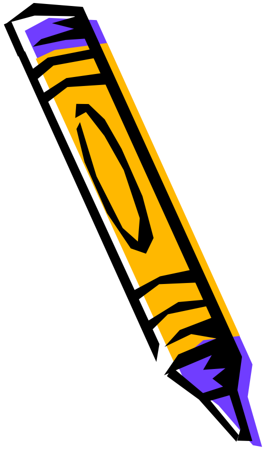
Here there is one-to-many relationship between the book table and the publisher. A book has only one publisher, and a publisher will publish many books. When we have a one-to-many relationship, we place a foreign key in the Book Table, pointing to the primary key of the Publisher Table.

Third normal form (3NF) requires

- Table must be in Second Normal Form
- All columns must relate directly to the primary key

- If your table is 2NF, there is a good chance it is 3NF

Impact of Normalization



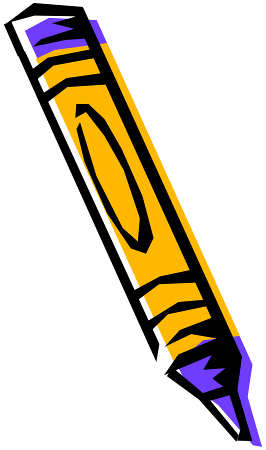
- Greater overall database organization
- Reduction of redundant data
- Data consistency within the database
- A much more flexible database design
- A better handle on database security
- Faster sorting and index creation.
- Fewer indexes per table, which improves the performance of INSERT, UPDATE, and DELETE statements.

Impact of Normalization

- Normalization simplifies updates, but **reads** are more common!

User Table		
Name	**Address Line 1**	**State**
XYZ	ABC	USA
123	DEF	USA

Denormalization



- The process of attempting to optimize the read performance of a database by adding redundant data or by grouping data.
- Utilize both the normalized and denormalized approaches depending on situations.

DENORMALIZATION:

- The process of attempting to optimize the read performance of a database by adding redundant data or by grouping data.

- Utilize both the normalized and denormalized approaches depending on situations.

Denormalization:

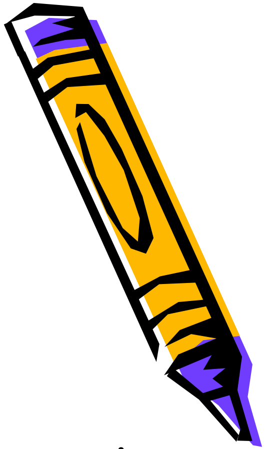
- Use with caution
- Normalize first, then de-normalize
- Use only when you cannot optimize
- Try temp tables, UNIONs, VIEWs, subselects first

DATABASE INDEX-
AN IMPORTANT CONCEPT IN DATABASE DESIGN:
Why is it needed?

When data is stored on disk based storage devices, it is stored as blocks of data. These blocks are accessed in their entirety, making them the atomic disk access operation. Disk blocks are structured in much the same way as linked lists; both contain a section for data, a pointer to the location of the next node (or block), and both need not be stored contiguously.

Due to the fact that a number of records can only be sorted on one field, we can state that searching on a field that isn't sorted requires a Linear Search which requires N/2 block accesses, where N is the number of blocks that the table spans. If that field is a non-key field (i.e. doesn't contain unique entries) then the entire table space must be searched at N block accesses.

What is Indexing?



Indexing is a way of sorting a number of records on multiple fields.

Creating an index on a field in a table creates another data structure which holds the field value, and pointer to the record it relates to.

This index structure is then sorted, allowing Binary Searches to be performed on it.

Whereas with a sorted field, a Binary Search may be used, this has log2 N block accesses. Also since the data is sorted given a non-key field, the rest of the table doesn't need to be searched for duplicate values, once a higher value is found. Thus the performance increase is substantial.

When indexing should be used?

Since indexes are only used to speed up the searching for a matching field within the records, it stands to reason that indexing fields used only for output would be simply a waste of disk space and processing time when doing an insert or delete operation.

Database systems usually implicitly create an index on a set of columns declared **PRIMARY KEY**, and some are capable of using an already existing index to police this constraint. Many database systems require that both referencing and referenced sets of columns in a **FOREIGN KEY** constraint are indexed, thus improving performance of inserts, updates and deletes to the tables participating in the constraint.

JOINS:

- Joining data together is one of the most significant strengths of a relational database.
- Joins allow database users to combine data from one table with data from one or more other tables as long as they are relations.
- A join condition is usually used to limit the combinations of table data to just those rows containing columns that match columns in the other table.
- Most joins are “equi-joins” where the data from a column in one table exactly matches data in the column of another table.

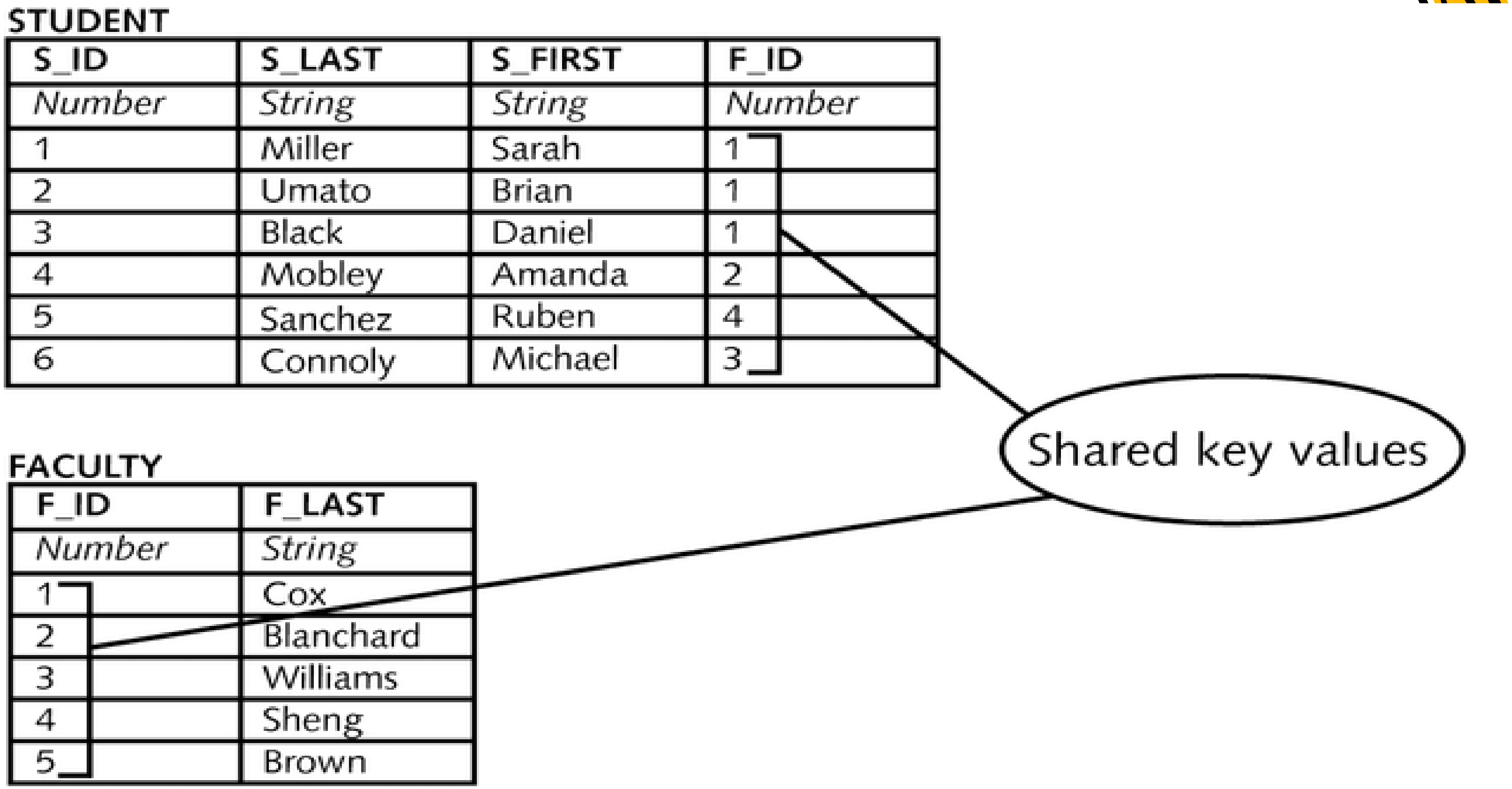
INNER JOINS:

An inner join (sometimes called a simple join) is a join of two or more tables that returns only those rows that satisfy the join condition.

- Traditional inner joins look for rows that match rows in the other table(s), i.e. to join two tables based on values in one table being equal to values in another table

- Also known as equality join, equijoin or natural join

- Returns results only if records exist in both tables



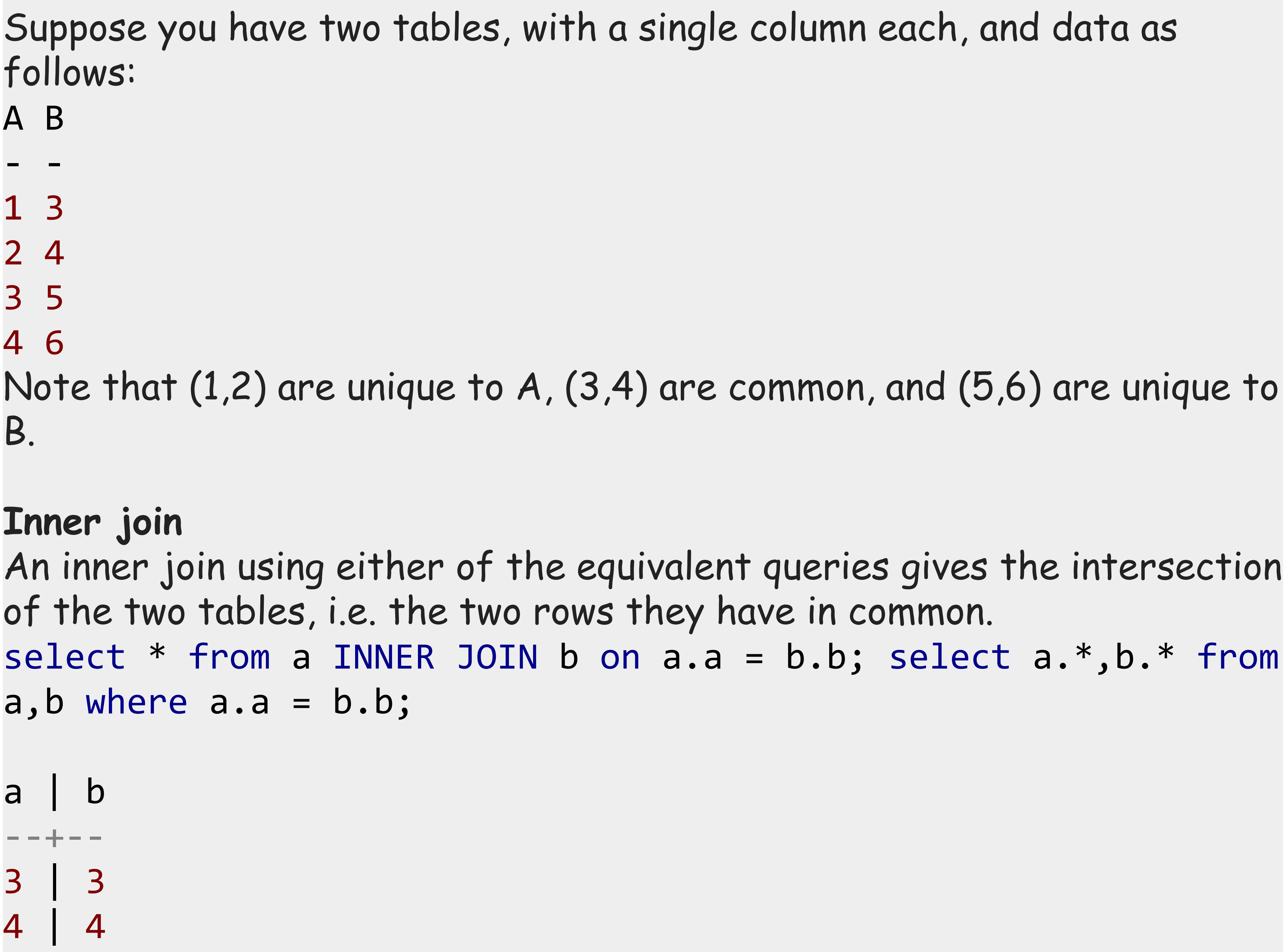
STUDENT

S_ID	S_LAST	S_FIRST	F_ID
Number	*String*	*String*	*Number*
1	Miller	Sarah	1
2	Umato	Brian	1
3	Black	Daniel	1
4	Mobley	Amanda	2
5	Sanchez	Ruben	4
6	Connoly	Michael	3

FACULTY

F_ID	F_LAST
Number	*String*
1	Cox
2	Blanchard
3	Williams
4	Sheng
5	Brown

Figure 3-42 Joining two tables based on shared key values



Suppose you have two tables, with a single column each, and data as follows:

```
A  B
-  -
1  3
2  4
3  5
4  6
```

Note that (1,2) are unique to A, (3,4) are common, and (5,6) are unique to B.

Inner join

An inner join using either of the equivalent queries gives the intersection of the two tables, i.e. the two rows they have in common.

```
select * from a INNER JOIN b on a.a = b.b; select a.*,b.* from
a,b where a.a = b.b;

a | b
--+--
3 | 3
4 | 4
```

Left outer join

A left outer join will give all rows in A, plus any common rows in B.

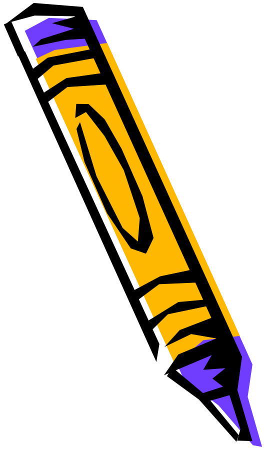
```
select * from a LEFT OUTER JOIN b on a.a = b.b; select a.*,b.* from a,b
where a.a = b.b(+);
a | b
--+-----
1 | null
2 | null
3 | 3
4 | 4
```

Full outer join

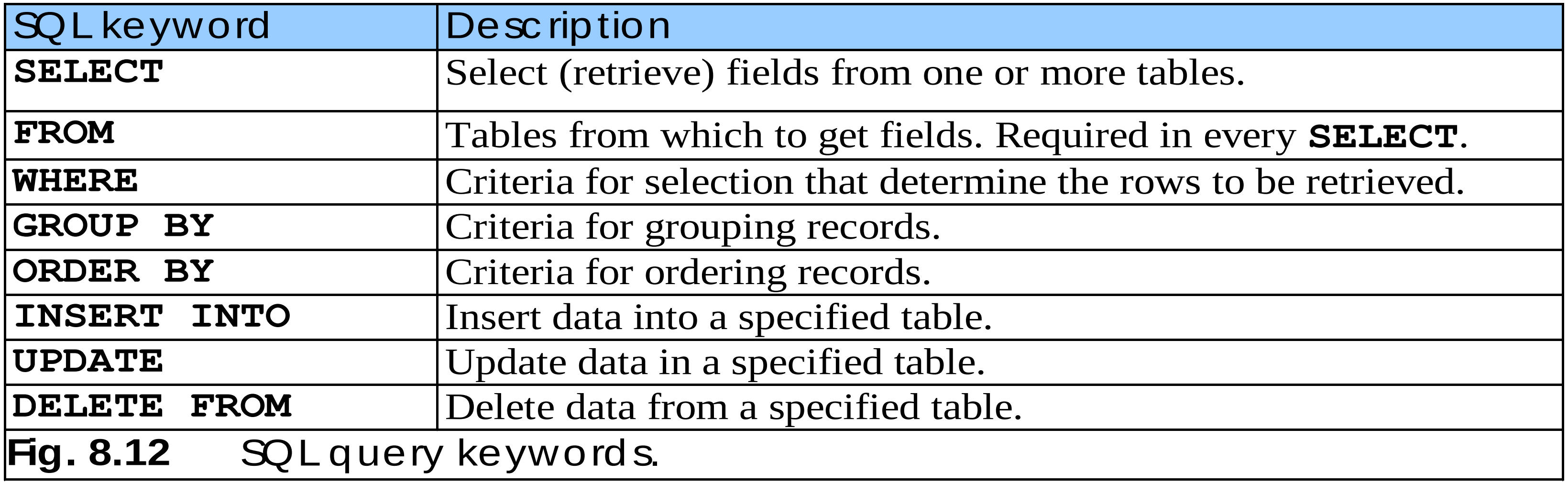
A full outer join will give you the union of A and B, i.e. all the rows in A and all the rows in B. If something in A doesn't have a corresponding datum in B, then the B portion is null, and vice versa.

```
select * from a FULL OUTER JOIN b on a.a = b.b;
   a | b
-----+-----
   1 | null
   2 | null
   3 | 3
   4 | 4
null | 6
null | 5
```

Structured Query Language (SQL)



- SQL overview
- SQL keywords



SQL keyword	Description
SELECT	Select (retrieve) fields from one or more tables.
FROM	Tables from which to get fields. Required in every **SELECT**.
WHERE	Criteria for selection that determine the rows to be retrieved.
GROUP BY	Criteria for grouping records.
ORDER BY	Criteria for ordering records.
INSERT INTO	Insert data into a specified table.
UPDATE	Update data in a specified table.
DELETE FROM	Delete data from a specified table.

Fig. 8.12 SQL query keywords.

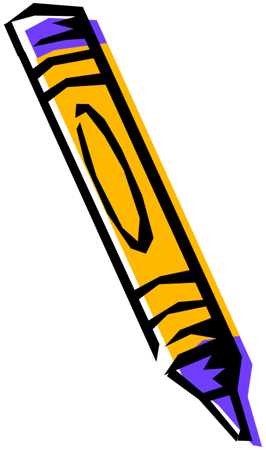
Basic SELECT Query

- Simplest format of a SELECT query
  - **SELECT** * **FROM** tableName
    - **SELECT** * **FROM** **authors**
- Select specific fields from a table
  - **SELECT** **authorID**, **lastName** **FROM** **authors**

authorID	lastName		
1	Deitel		
2	Deitel		
3	Nieto		
4	Santry		

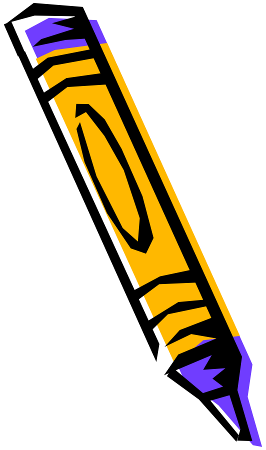
Fig. 8.13 authorID and lastName from the authors table.

WHERE Clause



- specify the selection criteria
  - **SELECT** fieldName1, fieldName2, … **FROM** tableName **WHERE** criteria
    - **SELECT** title, editionNumber, copyright
      FROM titles
      WHERE copyright > 1999
- **WHERE** clause condition operators
  - <, >, <=, >=, =, <>
  - **LIKE**
    - wildcard characters `%` and `_`

WHERE Clause (Cont.)

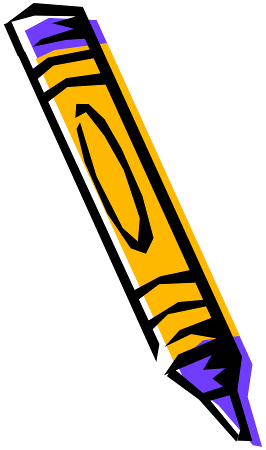


- **SELECT** authorID, firstName, lastName
  FROM authors
  WHERE lastName **LIKE** 'D%'

authorID	firstName	lastName
1	Harvey	Deitel
2	Paul	Deitel

Fig. 8.15 Authors whose last name starts with **D** from the **authors** table.

WHERE Clause (Cont.)

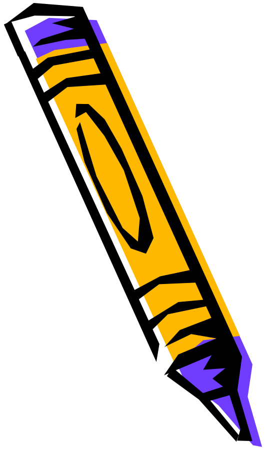


-
  ```
  SELECT authorID, firstName, lastName
    FROM authors
    WHERE lastName LIKE '_i%'
  ```

authorID	firstName	lastName
3	Tem	Nieto

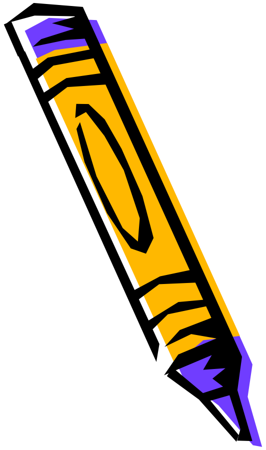
Fig. 8.16 The only author from the **authors** table whose last name contains **i** as the second letter.

ORDER BY Clause



- Optional **ORDER BY** clause
  - **SELECT** fieldName1, fieldName2, … **FROM** tableName **ORDER BY** field **ASC**
  - **SELECT** fieldName1, fieldName2, … **FROM** tableName **ORDER BY** field **DESC**
- **ORDER BY** multiple fields
  - **ORDER BY** field1 sortingOrder, field2 sortingOrder, …
- Combine the **WHERE** and **ORDER BY** clauses

ORDER BY Clause (Cont.)

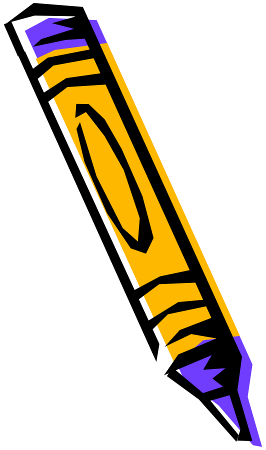


- **SELECT** authorID, firstName, lastName
  FROM authors
  ORDER BY lastName **ASC**

authorID	firstName	lastName
2	Paul	Deitel
1	Harvey	Deitel
3	Tem	Nieto
4	Sean	Santry

Fig. 8.17 Authors from table **authors** in ascending order by **lastName**.

ORDER BY Clause (Cont.)

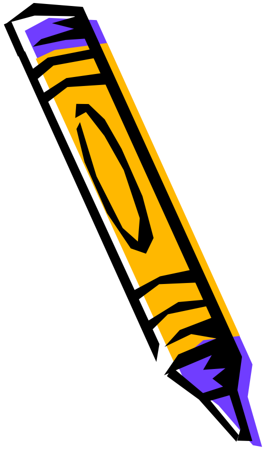


- **SELECT** authorID, firstName, lastName
  FROM authors
  ORDER BY lastName **DESC**

authorID	**firstName**	**lastName**
4	Sean	Santry
3	Tem	Nieto
2	Paul	Deitel
1	Harvey	Deitel

Fig. 8.18 Authors from table **authors** in descending order by **lastName**.

ORDER BY Clause (Cont.)



- **SELECT** authorID, firstName, lastName

  FROM authors

  ORDER BY lastName, firstName

authorID	firstName	lastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry

Fig. 8.19 Authors from table `authors` in ascending order by `lastName` and by `firstName`.

ORDER BY Clause (Cont.)

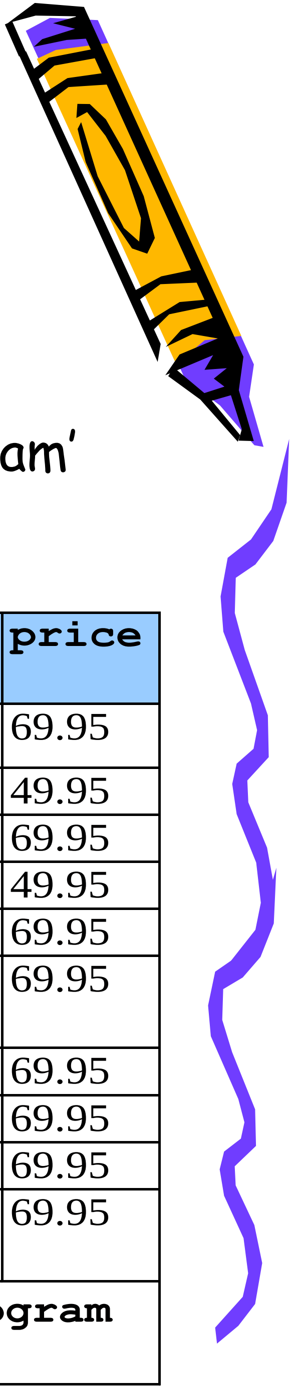
- **SELECT** isbn, title, editionNumber, copyright, price

  FROM titles **WHERE** title **LIKE** '%How to Program'

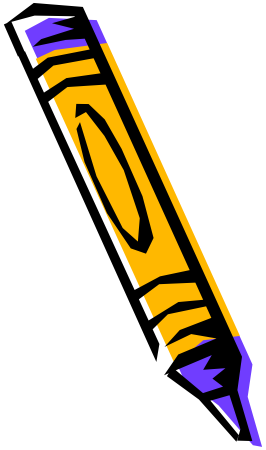
  ORDER BY title **ASC**

isbn	title	edition-Number	copy-right	price
0130895601	Advanced Java 2 Platform How to Program	1	2002	69.95
0132261197	C How to Program	2	1994	49.95
0130895725	C How to Program	3	2001	69.95
0135289106	C++ How to Program	2	1998	49.95
0130895717	C++ How to Program	3	2001	69.95
0130161438	Internet and World Wide Web How to Program	1	2000	69.95
0130284181	Perl How to Program	1	2001	69.95
0134569555	Visual Basic 6 How to Program	1	1999	69.95
0130284173	XML How to Program	1	2001	69.95
013028419x	e-Business and e-Commerce How to Program	1	2001	69.95

Fig. 8.20 Books from table `titles` whose title ends with `How to Program` in ascending order by `title`.



Merging Data from Multiple Tables: Joining



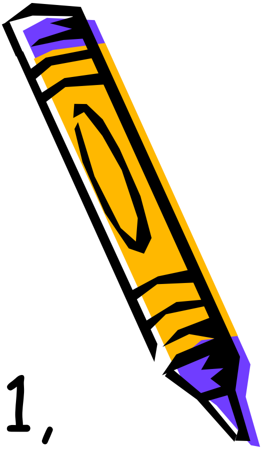
- Join the tables
  - Merge data from multiple tables into a single view
  - **SELECT** fieldName1, fieldName2, …
    FROM table1, table2
    WHERE table1.fieldName = table2.fieldName
  - **SELECT** firstName, lastName, isbn
    FROM authors, authorISBN
    WHERE authors.authorID = authorISBN.authorID
    ORDER BY lastName, firstName

Merging Data from Multiple Tables: Joining (Cont.)

firstName	lastName	isbn	firstName	lastName	isbn
Harvey	Deitel	0130895601	Harvey	Deitel	0130284173
Harvey	Deitel	0130284181	Harvey	Deitel	0130829293
Harvey	Deitel	0134569555	Paul	Deitel	0130852473
Harvey	Deitel	0130829277	Paul	Deitel	0138993947
Harvey	Deitel	0130852473	Paul	Deitel	0130125075
Harvey	Deitel	0138993947	Paul	Deitel	0130856118
Harvey	Deitel	0130125075	Paul	Deitel	0130161438
Harvey	Deitel	0130856118	Paul	Deitel	013028419x
Harvey	Deitel	0130161438	Paul	Deitel	0139163050
Harvey	Deitel	013028419x	Paul	Deitel	0135289106
Harvey	Deitel	0139163050	Paul	Deitel	0130895717
Harvey	Deitel	0135289106	Paul	Deitel	0132261197
Harvey	Deitel	0130895717	Paul	Deitel	0130895725
Harvey	Deitel	0132261197	Tem	Nieto	0130284181
Harvey	Deitel	0130895725	Tem	Nieto	0130284173
Paul	Deitel	0130895601	Tem	Nieto	0130829293
Paul	Deitel	0130284181	Tem	Nieto	0134569555
Paul	Deitel	0130284173	Tem	Nieto	0130856118
Paul	Deitel	0130829293	Tem	Nieto	0130161438
Paul	Deitel	0134569555	Tem	Nieto	013028419x
Paul	Deitel	0130829277	Sean	Santry	0130895601

Fig. 8.21 Authors and the ISBN numbers for the books they have written in ascending order by `lastName` and `firstName`.

INSERT INTO Statement



- Insert a new record into a table
  - **INSERT INTO** tableName (fieldName1, … , fieldNameN)

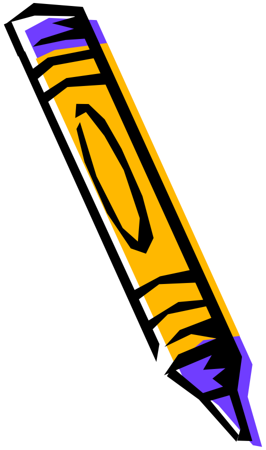
    VALUES (value1, … , valueN)
    - **INSERT INTO** authors (firstName, lastName)

      VALUES ('Sue', 'Smith')

authorID	**firstName**	**lastName**
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry
5	Sue	Smith

Fig. 8.22 Table `Authors` after an `INSERT INTO` operation to add a record.

`UPDATE` Statement

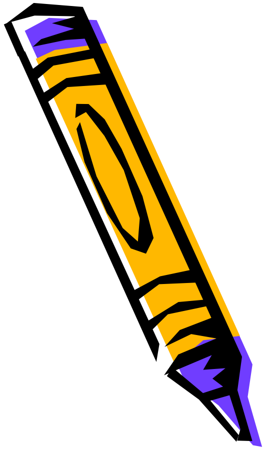


- Modify data in a table
  - **UPDATE** tableName
    SET fieldName1 = value1, … , fieldNameN = valueN
    WHERE criteria
    - **UPDATE** authors
      SET lastName = 'Jones'
      WHERE lastName = 'Smith' **AND** firstName = 'Sue'

authorID	**firstName**	**lastName**
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry
5	Sue	Jones

Fig. 8.23 Table **authors** after an **UPDATE** operation to change a record.

DELETE FROM Statement



- Remove data from a table
  - **DELETE FROM** tableName **WHERE** criteria
    - **DELETE FROM** authors
      WHERE lastName = 'Jones' **AND** firstName = 'Sue'

authorID	**firstName**	**lastName**
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry

Fig. 8.24 Table **authors** after a **DELETE** operation to remove a record.

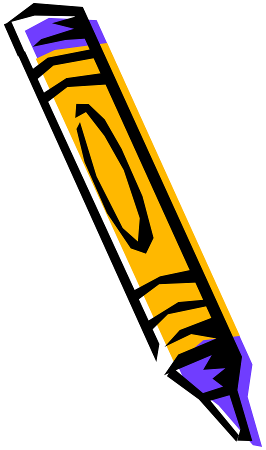
Stored Procedure

- A stored procedure is a subroutine available to applications accessing a relational database system.
- A procedure can be stored in the database as a database object for repeated execution
- Stored procedure can return multiple values using the OUT parameter or return no value at all.

Benefits of Stored Procedure

- Precompiled execution
- Reduced client/server traffic
- Efficient reuse of code and programming abstraction.
- Enhanced security controls.

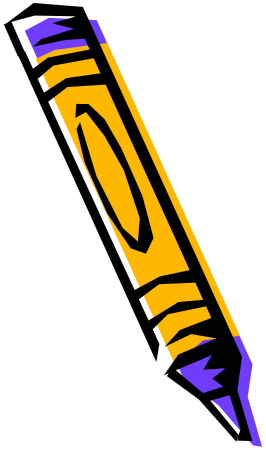
Stored Procedure



<SYNTAX>

```
CREATE [OR REPLACE] PROCEDURE <PROCEDURE NAME>
([MODE 1]  argument 1  datatype-1,
[MODE 2] argument 2  datatype-2,
…………………
BEGIN
Body
END
```

Stored Procedure

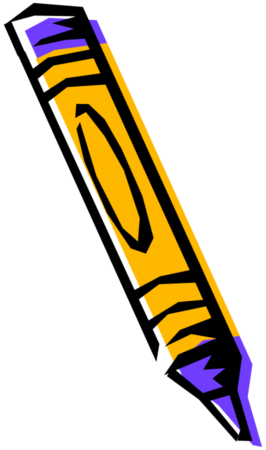


```
DELIMITER //
CREATE PROCEDURE GetAllProducts()
BEGIN
SELECT * FROM products;
END //
DELIMITER ;
```

Procedure Execution

```
Call GetAllProducts()
```

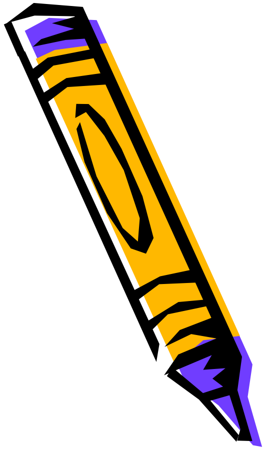
Stored Procedure



There are three types of modes for arguments

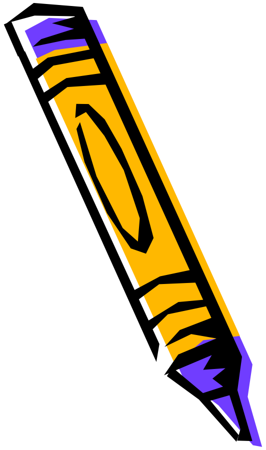
- IN
- OUT
- IN OUT

JDBC



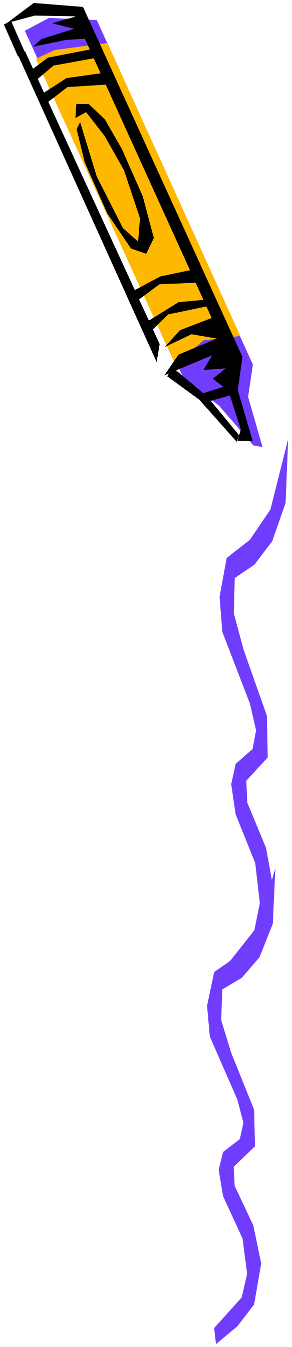
- Database
  - Collection of data
- DBMS
  - Database management system
  - Storing and organizing data
- SQL
  - Relational database
  - Structured Query Language
- JDBC
  - Java Database Connectivity
  - JDBC driver

Points to remember



- JDBC Driver – Load the proper driver
- DB connection
- Statement
- Executing the statements
- ResultSet
- Close – close connection Or connectionPool
- PreparedStatement

Relational-Database Model



- Relational database
  - Table
  - Record
  - Field, column
  - Primary key
    - Unique data
- SQL statement
  - Query
  - Record sets

Manipulating Databases with JDBC

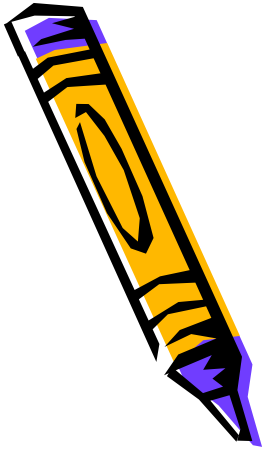
- Connect to a database
- Query the database
- Display the results of the query

Connecting to and Querying a JDBC Data Source

- DisplayAuthors
  - Retrieves the entire `authors` table
  - Displays the data in a `JTextArea`

Create Connection at Init()

```
public class SQLGatewayServlet extends HttpServlet{

    private Connection connection;

    public void init() throws ServletException{
        try{
            Class.forName("org.gjt.mm.mysql.Driver");
            String dbURL = "jdbc:mysql://localhost/murach";
            String username = "root";
            String password = "";
            connection = DriverManager.getConnection(
                dbURL, username, password);
        }
```

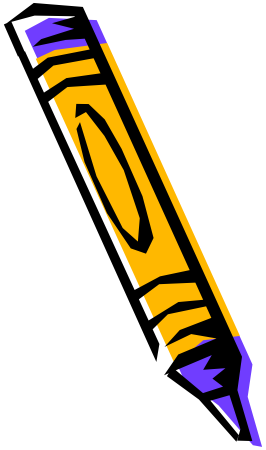


```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws IOException, ServletException{

      String sqlStatement = request.getParameter("sqlStatement");
      String message = "";

      try{    Statement statement = connection.createStatement();
              sqlStatement = sqlStatement.trim();
              String sqlType = sqlStatement.substring(0, 6);
              if  (sqlType.equalsIgnoreCase("select")){
                     ResultSet resultSet = statement.executeQuery(sqlStatement);
            // create a string that contains a HTML-formatted result set
              message = SQLUtil.getHtmlRows(resultSet);
         }  else      {
            int i = statement.executeUpdate(sqlStatement);
            if (i == 0) // this is a DDL statement
               message = "The statement executed successfully.";
            else        // this is an INSERT, UPDATE, or DELETE statement
               message = "The statement executed successfully.<br>"
                      + i + " row(s) affected.";
         }
         statement.close();
      }
```

From JDBC Example at course web page

```
 public void init() throws ServletException{
      connectionPool = MurachPool.getInstance();
   }

public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws IOException, ServletException{

      Connection connection = connectionPool.getConnection();

      String firstName = request.getParameter("firstName");
      String lastName = request.getParameter("lastName");
      String emailAddress =
         request.getParameter("emailAddress");
      User user = new User(firstName, lastName, emailAddress);

      HttpSession session = request.getSession();
      session.setAttribute("user", user);

      String message = "";
```

Processing Multiple ResultSets or Update Counts



- Execute the SQL statements
- Identify the result type
  - `ResultSets`
  - Update counts
- Obtain result
  - `getResultSet`
  - `getUpdateCount`

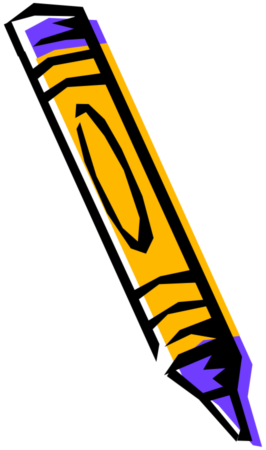
Prepared Statement

- Sometimes prepared statement is more convenient and more efficient for sending SQL statements to the database.
- When to use PreparedStatement
  - When you want to execute a Statement object many times, it will normally reduce execution time to use a PreparedStatement object instead

- The main feature of a PreparedStatement object is that unlike a Statement object, it is given an SQL statement when it is created. The advantage to this is that in most cases, this SQL statement will be sent to the DBMS right away, where it will be compiled. As a result, the PreparedStatement object contains not just an SQL statement, but an SQL statement that has been precompiled.

- This means that when the PreparedStatement is executed, the DBMS can just run the PreparedStatement's SQL statement without having to compile it first.

PreparedStatement example



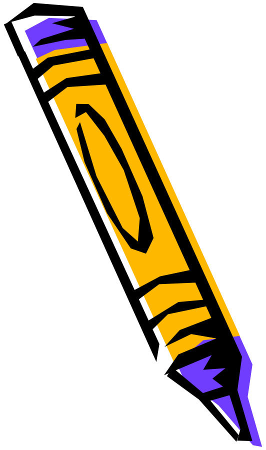
```
try{
        String _querySelect =
        "SELECT * FROM MOVIE WHERE title like ?";

        preStatement = connection.prepareStatement(
        _querySelect,
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE );
}
```

Prepared Statement Example

- https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/1938

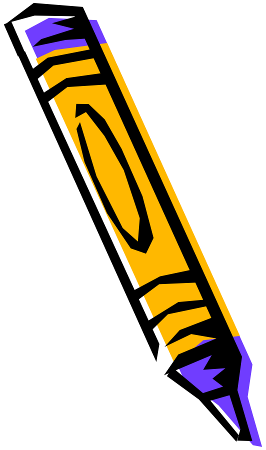
```
Connection connection = DriverManager.getConnection("jdbc:sapdb://" +
          Server + "/" +  Database, User, Password);
// Preparing the query to be executed
preStatement = connection.prepareStatement(
          "insert into addimage values(?,?,?)");

// Setting the actual values in the query
preStatement.setString(1,fileId);
preStatement.setString(2,fileDes);

// A file reader to get the contents of image
FileInputStream fi=new FileInputStream(fileName);
byte[] Img= new byte[fi.available()+1];
fi.read(Img);
preStatement.setBytes(3,Img);

// Executing the SQL Query
preStatement.execute();
System.out.println("Image Successfully inserted into MaxDB!");
```

JDBC 2.0 Optional Package
javax.sql

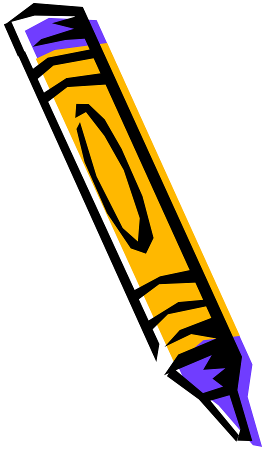


- Package javax.sql
  - Included with Java 2 Enterprise Edition
- Interfaces in package javax.sql
  - `DataSource`
  - `ConnectionPoolDataSource`
  - `PooledConnection`
  - `RowSet`

Connection Pooling



- Database connection
  - Overhead in both time and resources
- Connection pools
  - Maintain may database connections
  - Shared between the application clients



```
import util.MurachPool;

public class EmailServlet extends HttpServlet{

    private MurachPool connectionPool;

    public void init() throws ServletException{
        connectionPool = MurachPool.getInstance();
    }

    public void destroy() {
        connectionPool.destroy();
    }
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException, ServletException{

        Connection connection = connectionPool.getConnection();

        String firstName = request.getParameter("firstName");
        String lastName = request.getParameter("lastName");
        String emailAddress = request.getParameter("emailAddress");
        User user = new User(firstName, lastName, emailAddress);

        HttpSession session = request.getSession();
        session.setAttribute("user", user);
```