SQL

- SQL is the Structured Query Language
  - originally developed for IBM’s System/R in 1970s
  - now an open standard (actually, a bunch of them)
- a common interface for relational DB’s
  - manipulation
    - creating tables, updating them, adding data
  - examination
    - looking data up: queries

SQL

- queries have three basic components
  - select something
    - what aspects of the data do we want to see
  - from somewhere
    - what tables contain it
  - where condition
    - filtering of results
- basic syntax
  - select attribute1, attribute2, ...
    - from relation1, relation2, ...
    - where predicate

- some basic examples
  - select title from books
  - select title from books where author='dourish'
  - select title from books where author='dourish' and price < 35.00
  - select grade from students where id='12312312'
  - select id,name from students where grade='f'

SQL

- queries across multiple tables
  - relational model splits data into different tables
  - queries need to integrate across multiple tables
  - selects that combine table are called joins
- example
  - tables: "students" (id, name), "grades" (id, score)
  - select name, grade
    - from students, grades
    - where students.id = grades.id

SQL

- joins aren’t as clever as you’d think
  - a basic pairwise combination of possible elements
    - select name,grade from students,grades where grade='A'
• joins aren’t as clever as you’d think
  – a basic pairwise combination of possible elements
    • select name, grade from students, grades where grade = 'A'
    • select name, grade from students, grades where grade = 'A' and students.id = grades.id

• combining results
  – union, intersect, except
  – these are operators over selections
  • examples
    - select title from books where author = 'dourish' except select title from books where title = 'context-aware computing'
    - select id from homework1 where score > 85 intersect select id from homework2 where score > 85
    - NB: neither of these are the easiest ways to do them...

• postprocessing (order, group)
  – need to organise results
  – order (sort), group (clustering)
  • examples
    - select id, name, score from students order by score
    - select id, name, score from students order by score limit 10
    - select model, price from products where price < 100 order by price desc
    - select manufacturer from price_list group by manufacturer

• some functions over results
  – e.g. avg(), sum(), count(), min(), max() ...
  – functions apply to single columns
  • collapse multiple entries to a single value
  • examples
    - select count(*) from students where grade = 'A'
    - select avg(score) from grades

processing stages

[Diagram of processing stages: join, select, operate, arrange]
SQL

• more complex processing
  – where there are multiple fields, this is not enough
  – need to specify two things
    • the processing to perform (avg, sum, etc)
    • how to group elements for processing
      – why?
  • example
    – select author, avg(price) from books group by author

SQL

• working with computed fields
  – remember, computed values look like columns
  – sometimes need to refer to outputs of operations
    – "as" operator provides naming
      • think of the output of any select as a temporary relation
      • "as" creates the names of the attributes/columns
  • example
    – select author, avg(price) as average from books group by author order by average

SQL

• working with computed fields
  – need a way to refer to the outputs of operations
  – "as" operator provides naming
    • think of the output of any select as a temporary relation
    • "as" creates the names of the attributes/columns
  • example
    – select author, avg(price) as average from books group by author order by average

SQL

• summary
  – selecting, combining, processing
  • there’s more, of course...
    – subqueries
    – update and modification as well as querying

using SQL

• what SQL is not
  – not a full programming language
  – not a development environment
• sql queries normally embedded in programs
  – e.g. from java, using JDBC
  – languages differ in their degrees of integration

using SQL

```java
Class.forName("JDBC_CLASS");
Connection conn = DriverManager.getConnection("DB_URL", "login", "password");
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select title, author from books");
ResultSetMetaData md = rs.getMetaData();
out.println("<TABLE BORDER=2>\n TR<BR>
 for (int i = 1; i < md.getColumnCount() + 1; i++) {
 out.println("<TD><B>");
 out.println(md.getColumnName(i).trim());
 out.println("</B></TD>\n for (int i = 1; i < md.getColumnCount() + 1; i++) {
 out.println("<TD>");
 out.println(rs.getString(i));
 out.println("</TD>\n for (int i = 1; i < md.getColumnCount() + 1; i++) {
 out.println("<TD>");
 out.println("</TD>\n out.println("</TABLE>");
```
normalization

- again, relationship between defn and queries
  - the structure of your database is intimately tied to the queries you will perform against it
  - sql has certain expectations
    - column names and references
    - how joins work
  - database normalization
    - ensure database meets a set of structural criteria
    - enshrined as a set of “normal forms”

- there’s a whole set of normal forms...
- we’ll just look at three
  - first normal form
    - rule: no repeating groups
  - second normal form
    - rule: no non-key attribute depends on part of the key
  - third normal form
    - rule: no non-key attribute depends on another non-key attribute

first normal form

- no repeating groups
  - essentially, normalise the record length
  - imagine you were trying to do a join on author:

<table>
<thead>
<tr>
<th>Title</th>
<th>Price</th>
<th>Author1</th>
<th>Author2</th>
<th>Author3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where the Action Is</td>
<td>$30.00</td>
<td>Jpd</td>
<td>Ullman</td>
<td>Ullman</td>
</tr>
<tr>
<td>Analyzing Social Settings</td>
<td>$31.95</td>
<td>Lofland</td>
<td>Lofland</td>
<td></td>
</tr>
<tr>
<td>Compilers</td>
<td>$72.00</td>
<td>Aho</td>
<td>Sethi</td>
<td>Ullman</td>
</tr>
</tbody>
</table>

- no non-key attributes depend on part of the key
  - essentially, make key as small as it can be
  - express only a single relationship per table

<table>
<thead>
<tr>
<th>Author</th>
<th>Title</th>
<th>Price</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dourish</td>
<td>Where the Action Is</td>
<td>$30.00</td>
<td><a href="mailto:jpd@ics.uci.edu">jpd@ics.uci.edu</a></td>
</tr>
<tr>
<td>Baldi</td>
<td>Bioinformatics</td>
<td>$49.95</td>
<td><a href="mailto:bald@ics.uci.edu">bald@ics.uci.edu</a></td>
</tr>
</tbody>
</table>

second normal form

- no non-key attributes depend on part of the key
  - essentially, make key as small as it can be
  - express only a single relationship per table

<table>
<thead>
<tr>
<th>Author</th>
<th>Title</th>
<th>Price</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dourish</td>
<td>Where the Action Is</td>
<td>$30.00</td>
<td><a href="mailto:jpd@ics.uci.edu">jpd@ics.uci.edu</a></td>
</tr>
<tr>
<td>Baldi</td>
<td>Bioinformatics</td>
<td>$49.95</td>
<td><a href="mailto:bald@ics.uci.edu">bald@ics.uci.edu</a></td>
</tr>
</tbody>
</table>
second normal form

- no non-key attributes depend on part of the key
- essentially, make key as small as it can be
- express only a single relationship per table

<table>
<thead>
<tr>
<th>Author</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dourish</td>
<td><a href="mailto:jpd@ics.uci.edu">jpd@ics.uci.edu</a></td>
</tr>
<tr>
<td>Baldi</td>
<td><a href="mailto:baldi@ics.uci.edu">baldi@ics.uci.edu</a></td>
</tr>
</tbody>
</table>

third normal form

- no attributes depend on other non-key attributes
- again, a row should be about just one relationship

<table>
<thead>
<tr>
<th>Author</th>
<th>Title</th>
<th>Price</th>
<th>Purchaser</th>
<th>Seller</th>
<th>Employed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dourish</td>
<td>Where the Action Is</td>
<td>$30.00</td>
<td>Hans</td>
<td>Hans</td>
<td>1/1/03</td>
</tr>
<tr>
<td>Dourish</td>
<td>Where the Action Is</td>
<td>$30.00</td>
<td>Joey</td>
<td>Amy</td>
<td>1/1/02</td>
</tr>
<tr>
<td>Baldi</td>
<td>Bioinformatics</td>
<td>$49.95</td>
<td>Lisa</td>
<td>Jaime</td>
<td>7/1/01</td>
</tr>
</tbody>
</table>

third normal form

<table>
<thead>
<tr>
<th>Author</th>
<th>Title</th>
<th>Price</th>
<th>Purchaser</th>
<th>Seller</th>
<th>Employed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dourish</td>
<td>Where the Action Is</td>
<td>$30.00</td>
<td>Maria</td>
<td>Hans</td>
<td>1/1/03</td>
</tr>
<tr>
<td>Dourish</td>
<td>Where the Action Is</td>
<td>$30.00</td>
<td>Joey</td>
<td>Amy</td>
<td>1/1/02</td>
</tr>
<tr>
<td>Baldi</td>
<td>Bioinformatics</td>
<td>$49.95</td>
<td>Lisa</td>
<td>Jaime</td>
<td>7/1/01</td>
</tr>
</tbody>
</table>

third normal form

<table>
<thead>
<tr>
<th>Title</th>
<th>Purchaser</th>
<th>Seller</th>
<th>Seller</th>
<th>Employed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where the Action Is</td>
<td>Maria</td>
<td>Hans</td>
<td>Hans</td>
<td>1/1/03</td>
</tr>
<tr>
<td>Where the Action Is</td>
<td>Joey</td>
<td>Amy</td>
<td>Amy</td>
<td>1/1/02</td>
</tr>
<tr>
<td>Bioinformatics</td>
<td>Lisa</td>
<td>Jaime</td>
<td>Jaime</td>
<td>7/1/01</td>
</tr>
</tbody>
</table>

normalization

- normalization transforms database structure
  - eliminates repetition
  - disentangles dependencies
  - clarifies relationships
- two benefits of these transformations
  - semantic
    - cleaner definitions
    - clarifies "meaning"
  - practical
    - optimizes for SQL-based queries

next time

- an assignment on this stuff
  - to be done online
- moving on from machine metaphor
  - organisms
    - performance and competition
    - communication and interaction