Introduction to Databases
CSEP 544

Lecture #1
March 29, 2004
Alon Halevy

Staff
• Instructor: Alon Halevy
  – Allen Center, Room 576, alon@cs.washington.edu
  – Office hours: Just before class (or by email)
• TA: Stefan (Stebbi) Sigurdsson
  – stebbi@cs.washington.edu
  – Office hours: TBA

Communications
• Web page:
  http://www.cs.washington.edu/education/courses/csep544/04sp/
• Mailing list: follow the directions at:

Textbook(s)
Main textbook, available at the bookstore:

Almost identical, and also available at the bookstore:
• A First Course in Database Systems, Jeffrey Ullman and Jennifer Widom
• Database Implementation, Hector Garcia-Molina, Jeffrey Ullman and Jennifer Widom

Comments on the textbook

Other Texts
• Database Management Systems, Ramakrishnan
  – Very comprehensive
• Fundamentals of Database Systems, Elmasri, Navathe
  – Very widely used
• Foundations of Databases, Abiteboul, Hull, Vianu
  – Mostly theory of databases
• Data on the Web, Abiteboul, Buneman, Suciu
  – XML and other new advanced stuff

Other Required Readings
There will be reading assignments from the Web:
• SQL for Web Nerds, by Philip Greenspun,
  http://philip.greenspun.com/sql/
• Others, especially for XML

For SQL, a good source of information is the MSDN library (on your Windows machine)
Course Structure

- **Prerequisites:** Data structures course
- **Work & Grading:**
  - Homework: 30% - 3 of them, some light programming.
  - Project: 35% - coming up next.
  - Final: 35% (Discuss date)

The Project

- Important component of the course.
- **2 Phases.**
- I’ll tell you about phase 2 later.
- **Phase 1:**
  - You build a database application on your own.
  - The domain of the application is inventory of some sort.
  - The application will have a simple web interface.
  - Done by the end of week 4.

Today

- Motivation: why do we want databases.
- Overview of database systems
  - Reading assignment from SQL for Web Nerds, by Philip Greenspun, Introduction http://philip.greenspun.com/sql/
- Course Outline.
- Basic elements of SQL

What Is a Relational Database Management System?

- Database Management System = DBMS
- Relational DBMS = RDBMS

- A program that makes it easy for you to manipulate large amounts of data.
- Frees you from thinking about details. Enables you to focus on your challenges.

Where are RDBMS used?

- Backend for traditional "database" applications
  - Students and courses at a university
  - Bank accounting
  - Airline reservations
  - Movie listings
- Backend for large websites
- Backend for web services

Example of a Traditional Database Application

Suppose we are building a system to store the information about:

- **students**
- **courses**
- **professors**
- who takes what, who teaches what
Data Management

- Data management is more than databases.
- Imagine:
  - Complete Traffic Information Availability
  - MyNeededBits Anytime, Anywhere
  - Your favorite visionary application here
- The techniques we learn are the principles of managing data anywhere.

Can we do it without a DBMS?

Sure we can! Start by storing the data in files:

students.txt  courses.txt  professors.txt

Now write C or Java programs to implement specific tasks.

Doing it without a DBMS...

- Enroll "Mary Johnson" in "CSE444": Write a C program to do the following:
  - Read 'students.txt'
  - Read 'courses.txt'
  - Find and update the record "Mary Johnson"
  - Write 'students.txt'
  - Write 'courses.txt'

Problem sw ithout a DBMS...

- System crashes:
  - What is the problem?
- Large data sets (say 50GB):
  - What is the problem?
- Simultaneous access by many users:
  - Need locks: we know them from OS, but now data on disk; and is there any fun to re-implement them?

Enters a DBMS

"Two-tier database system"

Functionality of a DBMS

The program makes SQL, which has two components:

- Data Definition Language - DDL
- Data Manipulation Language - DML
- Query language

Behind the scenes the DBMS has:

- Query optimizer
- Query engine
- Storage management
- Transaction management (concurrency, recovery)
How the Programmer Sees the DBMS

- Start with DDL to create tables:
  ```sql
  CREATE TABLE Students (
      Name CHAR(30)
      SSN CHAR(9) PRIMARY KEY NOT NULL,
      Category CHAR(20)
  );
  ...
  ``

- Continue with DML to populate tables:
  ```sql
  INSERT INTO Students
  VALUES('Charles', '123-45-6789', 'undergraduate');
  ...
  ``

Still implemented as files, but behind the scenes can be quite complex:

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-45-6789</td>
<td>Charles</td>
<td>undergraduate</td>
</tr>
<tr>
<td>234-56-7890</td>
<td>Dan</td>
<td>grad</td>
</tr>
</tbody>
</table>

Students: Takes:

<table>
<thead>
<tr>
<th>SSN</th>
<th>CID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-45-6789</td>
<td>CSE444</td>
<td></td>
</tr>
<tr>
<td>123-45-6789</td>
<td>CSE444</td>
<td></td>
</tr>
<tr>
<td>234-56-7890</td>
<td>CSE142</td>
<td></td>
</tr>
</tbody>
</table>

Transactions

- Enroll "Mary Johnson" in "CSE444":
  ```sql
  BEGIN TRANSACTION;
  INSERT INTO Takes
  SELECT Students.SSN, Courses.CID
  FROM Students, Courses
  WHERE Students.name = 'Mary Johnson' and Courses.name = 'CSE444';
  -- More updates here....
  IF everything-went-OK
  THEN COMMIT;
  ELSE ROLLBACK;
  ``

If system crashes, the transaction is still either committed or aborted.

Queries

- Find all courses that "Mary" takes:
  ```sql
  SELECT C.name
  FROM Students S, Takes T, Courses C
  WHERE S.name = 'Mary' and
        T.ssn = S.ssn and
        T.cid = C.cid
  ``

What happens behind the scene?
- Query processor figures out how to answer the query efficiently.

Building an Application with a DBMS

- Requirements modeling (conceptual, pictures)
  - Describe what entities should be part of the application and how they should be linked.
- Schema design and implementation
  - Decide on a set of tables, attributes.
  - Define the tables in the database system.
  - Populate database (insert tuples).
- Write application programs using the DBMS
  - Way easier now that the data management is taken care of.

Transactions

- A transaction = sequence of statements that either all succeed, or all fail.
- Transactions have the ACID properties:
  - A = Atomicity
  - C = Consistency
  - I = Independence
  - D = Durability

Queries

- Find all courses that "Mary" takes:
  ```sql
  SELECT C.name
  FROM Students S, Takes T, Courses C
  WHERE S.name = 'Mary' and
        T.ssn = S.ssn and
        T.cid = C.cid
  ```
Queries, behind the scene

Declarative SQL query → Imperative query execution plan:

SELECT  C.name
FROM Students S, Takes T, Courses C
WHERE S.name = "Mary" and S.ssn = T.ssn and T.cid = C.cid

The optimizer chooses the best-execution plan for a query.

Database Systems

- The big commercial database vendors:
  - Oracle
  - IBM (with DB2) bought Informix recently
  - Microsoft (SQL Server)
  - Sybase
- Some free database systems (Unix):
  - Postgres
  - My SQL
  - Predator
- In CSEP544 we use SQL Server. You may use something else, but you are on your own.

New Trends in Databases

- Object-relational databases
- Main memory database systems
  - XML
  - XML support
  - Middleware between XML and relational databases
  - Native XML database systems
  - Lots of research here at UW on XML and databases
- Data integration
- Peer to peer, stream data management – still research

The Study of DBMS

- Several aspects:
  - Modeling and design of databases
  - Database programming: querying and update operations
  - Database implementation
- DBMS study cuts across many fields of Computer Science: OS, languages, AI, Logic, multimedia, theory...

Course Outline

(may vary slightly)

Part I
- SQL (Chapter 7) and its advanced features.
- Database design (Chapters 2, 3, 7)
- XML, XQuery, XQuery
- Data storage, indexes (Chapters 11-13)
- Query execution and optimization (Chapter 15, 16)
- Data integration, metadata management

The Relational Model (Codd)

<table>
<thead>
<tr>
<th>Product</th>
<th>Attribute names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Price</td>
</tr>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
</tr>
<tr>
<td>Powaygimio</td>
<td>$29.99</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
</tr>
</tbody>
</table>
SQL Introduction

Standard language for querying and manipulating data

Structured Query Language

Many standards out there:
- ANSI SQL
- SQL-92 (a.k.a. SQL2)
- SQL-99 (a.k.a. SQL3)
- Vendors support various subsets of these
- What we discuss is common to all of them

Data in SQL

1. Atomic types, a.k.a. data types
2. Tables built from atomic types

Unlike XML, no nested tables, only flat tables are allowed:
- We will see later how to decompose complex structures into multiple flat tables

Data Types in SQL

- Characters:
  - CHAR(20) — fixed length
  - VARCHAR(40) — variable length
- Numbers:
  - BIGINT, INT, SMALLINT, TINYINT
  - REAL, FLOAT — differ in precision
  - MONEY
- Times and dates:
  - DATE
  - DATETIME — SQL Server
- Others... All are simple

Tables Explained

- A tuple = a record
  - Restriction: all attributes are of atomic type
- A table = a set of tuples
  - Like a list.
  - ... but it is unordered: no first(), no next(), no last().
**SQL Query**

Basic form: (plus many many more bells and whistles)

```
SELECT attributes
FROM relations (possibly multiple)
WHERE conditions (selections)
```

**Simple SQL Query**

```
SELECT *
FROM Product
WHERE Category = "Gadgets"
```

**A Notation for SQL Queries**

```
SELECT PName, Price, Manufacturer
FROM Product
WHERE Price > 100
```

**Selections**

What goes in the WHERE clause:
- \( x = y, x < y, x < y, \text{etc} \)
  - For numbers, they have the usual meanings
  - For CHAR and VARCHAR: lexicographic ordering
  - Expected conversion between CHAR and VARCHAR
  - For dates and times, what you expect...
- Pattern matching on strings...

**The LIKE operator**

- LIKE \( p \): pattern matching on strings
- \( p \) may contain two special symbols:
  - \( * \) = any sequence of characters
  - \( _ \) = any single character

```
SELECT * FROM Products
WHERE PName LIKE '%gizmo%
```

```
SELECT PName, Price, Manufacturer
FROM Product
WHERE Price > 100
```
Eliminating Duplicates

```
SELECT DISTINCT category
FROM Product
```

Compare to:

```
SELECT category
FROM Product
```

Ordering the Results

```
SELECT DISTINCT category
FROM Product
ORDER BY category
```

Compare to:

```
SELECT category
FROM Product
ORDER BY pname
```

Ordering the Results

```
SELECT category
FROM Product
ORDER BY pname
```

Ordering the Results

```
SELECT DISTINCT category
FROM Product
ORDER BY price, pname
```

Orders ascending, unless you specify the DESC keyword.

Ties are broken by the second attribute on the ORDER BY list, etc.

Joins in SQL

- Connect two or more tables:

```
Product (pname, price, category, manufacturer)
```

```
Company (cname, stockPrice, country)
```

Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT pname, price
FROM Product, Company
WHERE manufacturer = cname AND country = 'Japan'
AND price <= 200
```

Joins

Product (pname, price, category, manufacturer)

Company (cname, stockPrice, country)

Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECTpname, price
FROM Product, Company
WHERE manufacturer = cname AND country = 'Japan'
AND price <= 200
```

```
SELECT profile, price, s manufacturer
FROM Product
WHERE category = 'gizmo' AND price > 50
ORDER BY price, profile
```

Orders ascending, unless you specify the DESC keyword.

Ties are broken by the second attribute on the ORDER BY list, etc.
## Joins in SQL

**Product** (pname, price, category, manufacturer)
**Company** (cname, stockPrice, country)

1. Find all countries that manufacture some product in the 'Gadgets' category.
   - SELECT country
     FROM Product, Company
     WHERE manufacturer = cname AND category = 'Gadgets'

2. Find names of people living in Seattle that bought some product in the 'Gadgets' category, and the names of the stores they bought such product from.
   - SELECT DISTINCT person, store
     FROM Person, Purchase, Product
     WHERE person = buyer AND product = pname AND
       city = 'Seattle' AND category = 'Gadgets'

## Joins

**Product** (pname, price, category, manufacturer)
**Company** (cname, stockPrice, country)
**Purchase** (buyer, seller, store, product)
**Person** (person, phoneNumber, city)

- Find names of people living in Seattle that bought some product in the 'Gadgets' category, and the names of the stores they bought such product from.
  - SELECT DISTINCT person, store
    FROM Person, Purchase, Product
    WHERE person = buyer AND product = pname AND
      city = 'Seattle' AND category = 'Gadgets'

## When are two tables related?

- **Foreign keys are a method for schema designers to tell you so** (7.1)
  - A foreign key states that a column is a reference to the key of another table
  - Example: Product.manufacturer is foreign key of Company
  - Gives information and enforces constraint

## Disambiguating Attributes

- Sometimes two relations have the same attribute:
  - Person (person, address, worksfor)
  - Company (cname, address)
  - SELECT DISTINCT person, address
    FROM Person, Company
    WHERE worksfor = cname

## Tuple Variables

**Product** (pname, price, category, manufacturer)
**Purchase** (buyer, seller, store, product)
**Person** (person, phone, city)

- Find stores that sold at least one product that the store 'BestBuy' also sold:
  - select distinct x.store
    from Purchase AS x, Purchase AS y
    where x.product = y.product AND y.store = 'BestBuy'

When are two tables related?

- You guess they are
- I tell you so

Foreign keys are a method for schema designers to tell you so (7.1):

- A foreign key states that a column is a reference to the key of another table
- Example: Product.manufacturer is foreign key of Company
- Gives information and enforces constraint
Tuple Variables

General rule:
tuple variables introduced automatically by the system:
Product (name, price, category, manufacturer)

Becomes:

Doesn’t work when Product occurs more than once:
In that case the user needs to define variables explicitly.

SELECT name
FROM Product
WHERE price > 100

M eaning (Semantics) of SQL Queries

SELECT a1, a2, ..., ak
FROM R1 AS x1, R2 AS x2, ..., Rn AS xn
WHERE Conditions

1. Nested loops:

```sql
Answer = {}
for x1 in R1 do
for x2 in R2 do
  ...
  for xn in Rn do
    if Conditions
      then Answer = Answer ∪ \{(a1, ..., ak)\}
return Answer
```

First Unintuitive SQLism

SELECT R.A
FROM R, S, T
WHERE R.A = S.A OR R.A = T.A
Looking for R (S T)
But what happens if T is empty?

Exercises

Product (name, price, category, manufacturer)
Purchase (buyer, seller, store, product)
Company (name, stock price, country)
Person (name, phone number, city)

Ex #1: Find people who bought telephony products.
Ex #2: Find names of people who bought American products
Ex #3: Find names of people who bought American products and they live in Seattle.
Ex #4: Find people who have both bought and sold something.
Ex #5: Find people who bought stuff from Joe or bought products from a company whose stock prices is more than $50.

Union, Intersection, Difference

```sql
(SELECT name
FROM Person
WHERE City = "Seattle")
UNION
(SELECT name
FROM Person, Purchase
WHERE buyer = name AND store = "The Bon")
```

Similarly, you can use INTERSECT and EXCEPT.
You must have the same attribute names (scheme as: name e).

Conserving Duplicates

```sql
(SELECT name
FROM Person
WHERE City = "Seattle")
UNION ALL
(SELECT name
FROM Person, Purchase
WHERE buyer = name AND store = "The Bon")
```
**Subqueries**

A subquery producing a single value:

```sql
SELECT Purchase.product
FROM Purchase
WHERE buyer = (SELECT name FROM Person WHERE ssn = '123456789');
```

In this case, the subquery returns one value. If it returns more, it's a run-time error.

---

**Subqueries Returning Relations**

**Find companies who manufacture products bought by Joe Blow.**

```sql
SELECT Company.name
FROM Company, Product
WHERE Company.name = Product.manufacturer
AND Product.name IN
(SELECT Purchase.product
FROM Purchase
WHERE Purchase.buyer = 'Joe Blow');
```

Here the subquery returns a set of values: no more runtime errors.

**Removing Duplicates**

```sql
SELECT DISTINCT Company.name
FROM Company, Product
WHERE Company.name = Product.manufacturer
AND Product.name IN
(SELECT Purchase.product
FROM Purchase
WHERE Purchase.buyer = 'Joe Blow');
```

Now they are equivalent.

---

**Subqueries Returning Relations**

**Equivalent to:**

```sql
SELECT Company.name
FROM Company, Product, Purchase
WHERE Company.name = Product.manufacturer
AND Product.name = Purchase.product
AND Purchase.buyer = 'Joe Blow';
```

Is this query equivalent to the previous one? Beware of duplicates!

---

**Removing Duplicates**

```sql
SELECT DISTINCT Company.name
FROM Company, Product
WHERE Company.name = Product.manufacturer
AND Product.name IN
(SELECT Purchase.product
FROM Purchase
WHERE Purchase.buyer = 'Joe Blow');
```

Now they are equivalent.
Subqueries Returning Relations

You can also use: 
- s > ALL R
- s > ANY R
- EXISTS R

Product (pname, price, category, maker)
Find products that are more expensive than all those produced by "Gizmo-Works"

```sql
SELECT name
FROM Product
WHERE price > ALL (SELECT price
                      FROM Purchase
                      WHERE maker='Gizmo-Works')
```

Question for Database Fans and their Friends

- Can we express this query as a single SELECT-FROM-WHERE query, without subqueries?

  Hint: show that all SFW queries are monotone (figure out what this means). A query with ALL is not monotone.

Conditions on Tuples

```sql
SELECT DISTINCT Company.name
FROM Company, Product
WHERE Company.name=Product.maker
AND (Product.name, price) IN
  (SELECT Purchase.product, Purchase.price
   FROM Purchase
   WHERE Purchase.buyer = 'Joe Blow')
```

May not work in SQL server...

Correlated Queries

```sql
SELECT DISTINCT title
FROM Movie AS x
WHERE year <> ANY
  (SELECT year
   FROM Movie
   WHERE title = x.title)
```

Movie (title, year, director, length)
Find movies whose title appears more than once.

Note (1) scope of variables (2) this can still be expressed as single SFW correlation

Complex Correlated Query

```sql
SELECT DISTINCT pname, maker
FROM Product AS x
WHERE price > ALL (SELECT price
                   FROM Product AS y
                   WHERE x.maker = y.maker AND y.year < 1972)
```

Product (pname, price, category, m, aler, year)
- Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

Aggregation

```sql
SELECT Avg(price)
FROM Product
WHERE maker='Toyota'
```

SQL supports several aggregation operations:

- SUM, MIN, MAX, AVG, COUNT
**Aggregation: Count**

```
SELECT Count(*)
FROM Product
WHERE year > 1995
```

Except COUNT, all aggregations apply to a single attribute.

**Aggregation: Count**

```
COUNT applies to duplicates, unless otherwise stated:
SELECT Count(category) as Count(*)
FROM Product
WHERE year > 1995
```

Better:
```
SELECT Count(DISTINCT category)
FROM Product
WHERE year > 1995
```

---

**Simple Aggregation**

```
Purchase(product, date, price, quantity)
```

Example 1: find total sales for the entire database
```
SELECT Sum(price * quantity)
FROM Purchase
```

Example 1': find total sales of bagels
```
SELECT Sum(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```

---

**Grouping and Aggregation**

Usually, we want aggregations on certain parts of the relation.

```
Purchase(product, date, price, quantity)
```

Example 2: find total sales after 10/1 per product.
```
SELECT product, Sum(price * quantity) AS TotalSales
FROM Purchase
WHERE date > "10/1"
GROUP BY product
```

Let's see what this means:

---

**Grouping and Aggregation**

1. Compute the FROM and WHERE clauses.
2. Group by the attributes in the GROUPBY.
3. Select one tuple for every group (and apply aggregation).

```
SELECT can have (1) grouped attributes or (2) aggregates.
```
First compute the FROM-WHERE clauses (date > "10/1") then GROUP BY product:

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>10/19</td>
<td>0.52</td>
<td>17</td>
</tr>
<tr>
<td>Banana</td>
<td>10/22</td>
<td>0.52</td>
<td>7</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/20</td>
<td>0.85</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>0.85</td>
<td>15</td>
</tr>
</tbody>
</table>

Then, aggregate:

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>$29.75</td>
</tr>
<tr>
<td>Banana</td>
<td>$12.48</td>
</tr>
</tbody>
</table>

GROUP BY vs. Nested Queries

```
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "10/1"
GROUP BY product
```

```
SELECT DISTINCT x.product, (SELECT SUM(y.price*y.quantity)
FROM Purchase y
WHERE x.product = y.product
AND y.date > '10/1') AS TotalSales
FROM Purchase x
WHERE x.date > "10/1"
```

Another Example

<table>
<thead>
<tr>
<th>Product</th>
<th>SumSales</th>
<th>MaxQuantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>$12.48</td>
<td>17</td>
</tr>
<tr>
<td>Bagel</td>
<td>$29.75</td>
<td>20</td>
</tr>
</tbody>
</table>

HAVING Clause

```
SELECT product, SUM(price*quantity)
FROM Purchase
WHERE date > "9/1"
GROUP BY product
HAVING SUM(quantity) > 30
```

```
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "10/1"
GROUP BY product
HAVING SUM(quantity) > 30
```

General form of Grouping and Aggregation

```
SELECT S
FROM R1,…,Rn
WHERE C1
GROUP BY A1,…,An
HAVING C2
```

S may contain attributes A1,…,An and/or any aggregates but NO OTHER ATTRIBUTES

C1 = is any condition on the attributes in R1,…,Rn
C2 = is any condition on aggregate expressions
General form of Grouping and Aggregation

```
SELECT S
FROM R1, ..., Rn
WHERE C1
GROUP BY a1, ..., ak
HAVING C2
```

Evaluation steps:
1. Compute the FROM WHERE part, obtain a table with all attributes in R1, ..., Rn
2. Group by the attributes a1, ..., ak
3. Compute the aggregates in C2 and keep only groups satisfying C2
4. Compute aggregates in S and return the result.

Aggregation

- Author(login, name)
- Document(url, title)
- Wrote(login, url)
- Mentions(url, word)

- Find all authors who wrote at least 10 documents:
  - Attempt 1: with nested queries
    ```
    SELECT DISTINCT Author.name
    FROM Author
    WHERE count(SELECT Wrote.url
                FROM Wrote
                WHERE Author.login=Wrote.login)
    > 10
    ```
    This is SQL by a novice
  - Attempt 2: SQL style (with GROUP BY)
    ```
    SELECT Author.name
    FROM Author, Wrote
    WHERE Author.login=Wrote.login
    GROUP BY Author.name
    HAVING count(wrote.url) > 10
    ```
    This is SQL by an expert
  - No need for DISTINCT: automatically from GROUP BY

- Find all authors who have a vocabulary over 10000 words:
  ```
  SELECT Author.name
  FROM Author, Wrote, Mentions
  WHERE Author.login=Wrote.login AND Wrote.url=Mentions.url
  GROUP BY Author.name
  HAVING count(distinct Mentions.word) > 10000
  ```

Look carefully at the last two queries: you may be tempted to write them as nested queries, but in SQL we write them best with GROUP BY.

Exercises

- Product (pname, price, category, manufacturer)
- Purchase (buyer, seller, store, product)
- Company (cname, stock price, country)
- Person (pername, phone number, city)

- Ex #1: Find people who bought telephony products.
- Ex #2: Find names of people who bought American products.
- Ex #3: Find names of people who bought American products and they live in Seattle.
- Ex #4: Find people who have both bought and sold something.
- Ex #5: Find people who bought stuff from Joe or bought products from a company whose stock prices is more than $50.