CMPT 354

Structured Query Language

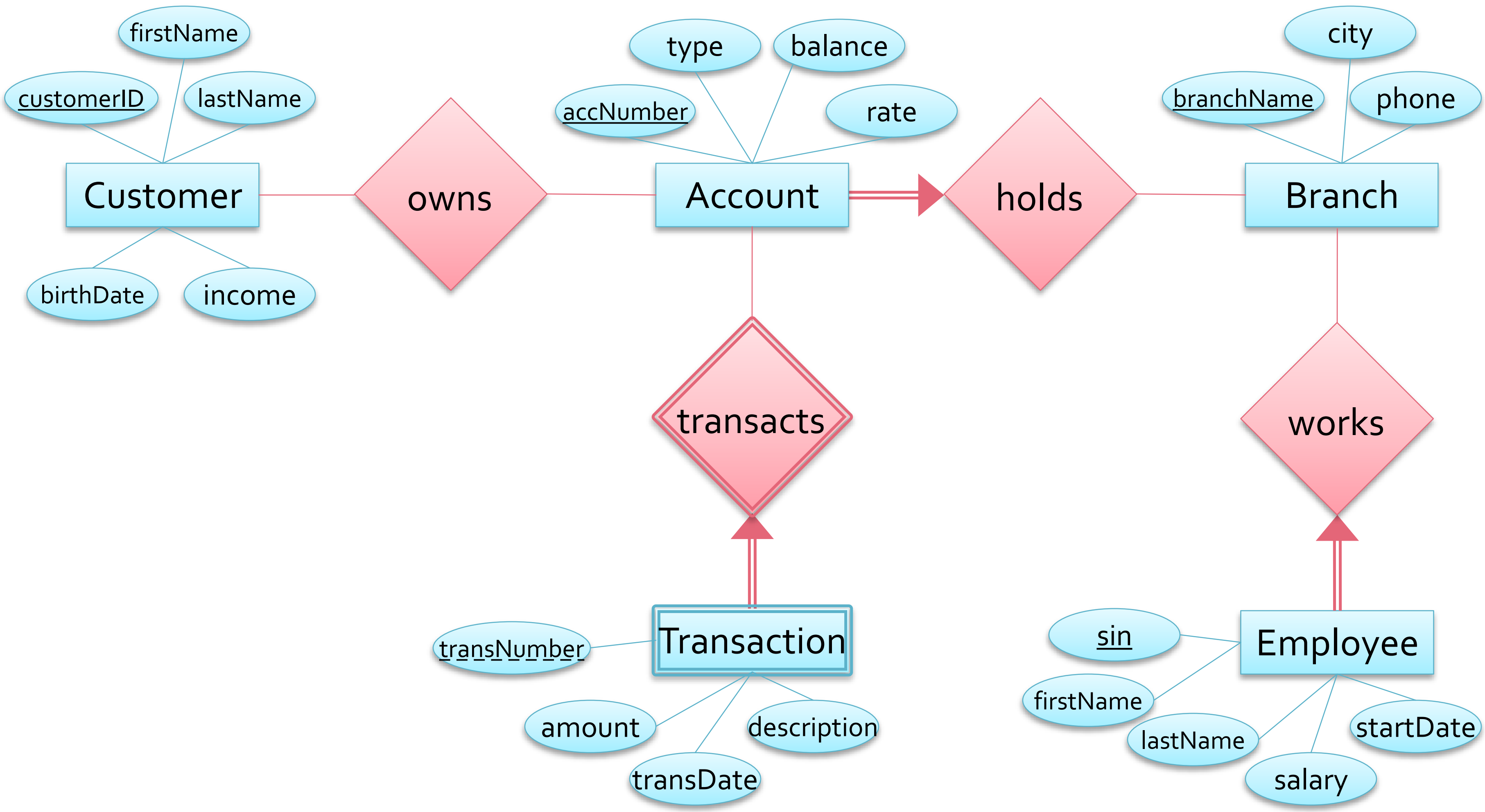
SQL

- Simple queries
- Set operations
- Aggregate operators
- Null values
- Joins
- Query Optimization

Components of SQL

- Data Manipulation Language (DML) to
  - Write queries
  - Insert, delete and modify records
- Data Definition Language (DDL) to
  - Create, modify and delete table definitions
  - Define integrity constraints
  - Define views

Bank ERD



firstName
customerID
lastName
Customer
birthDate
income
owns
type
balance
accNumber
rate
Account
holds
city
branchName
phone
Branch
transacts
works
transNumber
Transaction
amount
description
transDate
sin
Employee
firstName
lastName
startDate
salary

Bank Schemata

- Customer = {*customerID*, *firstName*, *lastName*, *birthDate*, *income*}
- Account = {*accNumber*, *type*, *balance*, *rate*, *branchName*}
  - *branchName* is a foreign key referencing Branch
- Owns = {*customerID*, *accNumber*}
  - *customerID* and *accNumber* are foreign keys referencing Customer and Account
- Transaction = {*accNumber*, *transNumber*, *amount*, *transDate*, *description*}
  - *accNumber* is a foreign key referencing Account
- Branch = {*branchName*, *city*, *phone*}
- Employee = {*sin*, *firstName*, *lastName*, *salary*, *startDate*, *branchName*}
  - *branchName* is a foreign key referencing Branch

SQL Standards 1

- There have been a number of SQL standards
  - SQL-92: the third revision of the standard
    - New data types and operations
  - SQL 1999: fourth revision, also known as SQL 3
    - More data types
    - User-defined types
    - Object-relational extensions
    - Regular expression matching
    - Some OLAP extensions (rollup, cube and grouping)

SQL Standards 2

- Even more SQL standards
  - SQL 2003
    - Modifications to SQL-1999
    - XML features
    - More OLAP capabilities including a window function
  - SQL 2008
    - Additions to triggers
    - XQuery pattern matching
  - SQL 2011
    - Support for temporal databases
  - SQL 2016
    - Row pattern matching, polymorphic table functions, JSON

Simple Queries

Basic SQL Syntax

- The basic form of an SQL query is

```
SELECT select-list
FROM from-list
WHERE condition
```

- Which is equivalent to

$\pi_{\text{select-list}}(\sigma_{\text{condition}}(\text{from-list}_1 \times \ldots \times \text{from-list}_n))$

Basic SQL Query

- The *select-list* is a list of column names belonging to tables named in the *from-list*
  - Column names may be prefixed by their table name or a *range variable* to remove ambiguity
- The *from-list* is a list of table names
  - A table name may be followed by a range variable
- The *condition* is a Boolean expression
  - Using the **<**, **<=**, **=**, **<>**, **>=**, and **>** operators, and
  - **AND**, **NOT**, and **OR** to join expressions

Sets and Bags and DISTINCT

- Although derived from relational algebra SQL does *not* remove duplicates unless instructed to do so
  - This is because removing duplicates entails sorting the result which can be very expensive
  - An SQL query returns a *multiset*, or *bag*, of rows
  - A bag allows more than one occurrence of an element, but the elements are unordered
    - e.g. **`{1,2,1}`** and **`{2,1,1}`** are the same bag
- The DISTINCT keyword specifies that duplicates *are* to be removed from a result

Basic Query Evaluation

```
SELECT DISTINCT firstName, lastName, income
FROM Customer
WHERE birthdate < '1950-09-27'  AND income > 100000
```

- Calculate the Cartesian product of tables in the *from-list*
  - Not necessary since this query only refers to the Customer table
- Remove any rows from the resulting table that do not meet the specified *condition*
- Remove all columns that do not appear in the *select-list*
- Remove duplicates if *DISTINCT* is specified

Two Tables

- Return the types, owner customerIDs, account numbers, and balances of accounts
- Where the balance is greater than $80,000 and the account is held at the Lonsdale branch

```
SELECT O.customerID, A.type, A.accNumber, A.balance
FROM Owns O, Account A
WHERE A.accNumber = O.accNumber AND A.balance > 80000
    AND A.branchName = 'Lonsdale'
```

Tuple Variables

Selecting all Columns

- Return customer information relating to customers whose last name is 'Summers'

i.e. all columns

```
SELECT *
FROM Customer
WHERE lastName = 'Summers'
```

Tuple Variables

- Tuple (or range) variables allow tables to be referred to in a query using an alias
  - The tuple variable is declared in the FROM clause by writing it immediately after the table it refers to
- If columns in two tables have the same names, tuple variables *must* be used to refer to them
  - Tuple variables are *not required* if a column name is *unambiguous*
- The full name of a table is an implicit tuple variable
  - e.g. SELECT Customer.firstName …

Why Use Tuple Variables?

- There are a number of reasons to use short explicit tuple variables
  - Distinguishing between columns with the same name but from different tables
  - Readability
  - Laziness
    - **C.birthDate** is less to type than **Customer.birthDate**
  - To make it easier to re-use queries
  - Because a query refers to the *same* table twice in the **FROM** clause

Required Tuple Variables

- Find the names and customerIDs of customers who have an income greater than Rupert Giles
  - By comparing each customer's income to Rupert's income
  - And using two references to the customer table

```
SELECT C.customerID, C.firstName, C.lastName
FROM Customer C, Customer RG
WHERE RG.firstName = 'Rupert' AND RG.lastName = 'Giles'
    AND C.income > RG.income
```

The implicit range name, *Customer*, cannot distinguish between the two references to the *Customer* table

What would the query return if there are two customers called Rupert Giles?

Expressions in SELECT

- Both the select and condition statements can include arithmetical operations
  - Arithmetical operations can be performed on numeric data in the **SELECT** statement
  - Arithmetical operations can also be included in the operands to Boolean operators in the **WHERE** condition
- Column names in the result table can be named, or renamed using the **AS** keyword

Renaming Using AS

- Return the balance of each account, and an estimate of the balance at the end of the year
  - Assuming an interest rate of 5%

```
SELECT accNumber, balance AS current,
            balance * 1.05 AS yearEndBalance
FROM Account
```

This query would return a table whose columns were titled *accNumber*, *current*, and *yearEndBalance*

Strings

- SQL strings are enclosed in single quotes
  - e.g. **firstName = 'Buffy'**

    Don't use “smart quotes”

  - Single quotes in a string can be specified using an initial single quote character as an escape
    - **author = 'O''Brian'**
- Strings can be compared lexicographically with the comparison operators
  - e.g. **'fodder' < 'foo'** is **TRUE**

Pattern Matching with LIKE

- SQL provides pattern matching support with the LIKE operator and two symbols
  - The % symbol stands for zero or more arbitrary characters
  - The _ symbol stands for exactly one arbitrary character
  - The % and _ characters can be escaped with \
    - LIKE 'C:\\Program Files\\%'
- Most comparison operators ignore white space, however LIKE does not, hence
  - 'Buffy' = 'Buffy ' is TRUE but
  - 'Buffy' LIKE 'Buffy ' is FALSE
  - Actual implementations of LIKE vary

Like This …

- Return the customerIDs, and first and last names of customers whose last name is similar to 'Smith'

```
SELECT customerID, firstName, lastName
FROM Customer
WHERE lastName LIKE 'Sm_t%'
```

This query would return customers whose last name is *Smit*, or *Smith*, or *Smythe*, or *Smittee*, or *Smut* (!) but not *Smeath*, or *Smart*, or *Smt*

SQL 1999 – `SIMILAR`

- The SQL standard allows for more powerful pattern matching using **SIMILAR**
- This operator allows regular expressions to be used as patterns while searching text
  - Similar to Unix regular expressions
- Again, note that not all SQL implementations may fully support the SQL standard
  - And some may extend it

Dates and Times

- The SQL standard includes date and time types
  - DATE – for dates, e.g. **2015-10-17**
  - TIME – for times, e.g. **17:30:29**
    - More precise time can be represented
  - TIMEZ – time with time zone information
  - TIMESTAMP – date and time together
    - e.g. **2015-10-17 17:30:29**
  - TIMESTAMPZ – date and time with time zone information
- Dates and times are enclosed in single quotes
  - They can be compared using comparison operators

SQL Implementations

- The implementation of SQL in a DBMS product may differ from the standard
- As an example consider the Transact-SQL date and time types used in MS SQL Server
  - date – date
  - datetime – date and time combined
  - datetime2 – extension of datetime
  - datetimeoffset – date and time with time zone
  - smalldatetime – date and time with smaller range and less precision, that requires less memory
  - time – time

Null Values

- A database may contain NULL values where
  - Data is unknown, or
  - Data does not exist for an attribute for a particular row
- There are special operators to test for null values
  - IS NULL tests for the presence of nulls and
  - IS NOT NULL tests for the absence of nulls
- Other comparisons with nulls evaluate to UNKNOWN
  - Even when comparing two nulls
  - Except that two rows are evaluated as duplicates, if all their corresponding attributes are equal or both null
- Arithmetic operations on nulls return NULL

Operations with NULL

- 23 < NULL
  - UNKNOWN
- NULL >= 47
  - UNKNOWN
- NULL = NULL
  - UNKNOWN

So don't do this to find nulls!

- NULL IS NULL
  - TRUE
- 23 + NULL - 3
  - NULL
- NULL * 0
  - NULL

Evaluation of Unknown

- Truth values for *unknown* results
  - *true* **OR** *unknown* = *true*,
  - *false* **OR** *unknown* = *unknown*,
  - *unknown* **OR** *unknown* = *unknown* ,
  - *true* **AND** *unknown* = *unknown*,
  - *false* **AND** *unknown* = *false*,
  - *unknown* **AND** *unknown* = *unknown*
  - **NOT** *unknown* = *unknown*
- The result of a **WHERE** clause is treated as *false* if it evaluates to *unknown*

More Fun with Nulls

- Let's say there are 2,753 records in Customer
  - How many rows would be returned by selecting customers whose incomes are 50,000 or less, or more than 50,000?

```
SELECT customerID, firstName, lastName
FROM Customer
WHERE income <= 50000 OR income > 50000
```

This should return 2,753 customers, but what happens when we don't know a customer's income (i.e. it is *null*)?

null <= 50000 = *unknown* and *null* > 50000 = *unknown* and *unknown or unknown* = *unknown* which is treated as *false*!

Ordering Output

- The output of an SQL query can be ordered
  - By any number of attributes, and
  - In either ascending or descending order
- Return the name and incomes of customers, ordered alphabetically by name

```
SELECT lastName, firstName, income
FROM Customer
ORDER BY lastName, firstName
```

The default is to use ascending order, the keywords *ASC* and *DESC*, following the column name, sets the order

Set Operations and Joins

Set Operations

- SQL supports union, intersection and set difference operations
  - Called **UNION**, **INTERSECT**, and **EXCEPT**
  - These operations must be performed on *union compatible* tables
- Although these operations are supported in the SQL standard, implementations may vary
  - **EXCEPT** may not be implemented
    - When it is, it is sometimes called **MINUS**

One of Two Branches

- Find customerIDs, and first and last names of customers
  - Who have accounts in either the Robson or the Lonsdale branches

```
SELECT C.customerID, C.firstName, C.lastName
FROM Customer C, Owns O, Account A
WHERE C.customerID = O.customerID AND
        A.accNumber = O.accNumber AND
       (A.branchName = 'Lonsdale' OR
        A.branchName = 'Robson')
```

This query would return the desired result, note that the brackets around the disjunction (*or*) are important

One of Two Branches – UNION

```
SELECT C1.customerID, C1.firstName, C1.lastName
FROM Customer C1, Owns O1, Account A1
WHERE C1.customerID = O1.customerID AND
          A1.accNumber = O1.accNumber AND
          A1.branchName = 'Lonsdale'
UNION
SELECT C2.customerID, C2.firstName, C2.lastName
FROM Customer C2, Owns O2, Account A2
WHERE C2.customerID = O2.customerID AND
          A2.accNumber = O2.accNumber AND
          A2.branchName = 'Robson'
```

This query returns the same result as the previous version, there are often many equivalent queries

Both Branches

- Now find customers who have accounts in *both* of the Lonsdale or Robson branches

```
SELECT C.customerID, C.firstName, C.lastName
FROM Customer C, Owns O, Account A
WHERE C.customerID = O.customerID AND
          A.accNumber = O.accNumber AND
         (A.branchName = 'Lonsdale' AND
          A.branchName = 'Robson')
```

wrong

Exactly which records does this query return?

A single account can be held at only one branch, therefore this query returns the empty set

Both Branches Again

- And here is another version …

```
SELECT C.customerID, C.firstName, C.lastName
FROM Customer C, Owns O1, Account A1,
          Owns O2, Account A2
WHERE C.customerID = O1.customerID AND
           O1.customerID = O2.customerID AND
           O1.accNumber = A1.accNumber AND
           O2.accNumber = A2.accNumber AND
           A1.branchName = 'Lonsdale' AND
           A2.branchName = 'Robson'
```

This query would return the desired result, but it is not pretty, nor is it very efficient - there are five tables in the FROM clause!

Both Branches – INTERSECT

```
SELECT C1.customerID, C1.firstName, C1.lastName
FROM Customer C1, Owns O1, Account A1
WHERE C1.customerID = O1.customerID AND
        A1.accNumber = O1.accNumber AND
        A1.branchName = 'Lonsdale'
INTERSECT
SELECT C2.customerID, C2.firstName, C2.lastName
FROM Customer C2, Owns O2, Account A2
WHERE C2.customerID = O2.customerID AND
        A2.accNumber = O2.accNumber
        AND A2.branchName = 'Robson'
```

What if you don't want customerID in the result ?

No Account in Robson

- Find the customerIDs of customers who have an account in the Lonsdale branch
- But who do *not* have an account in the Robson branch

```
SELECT O.customerID
FROM Owns O, Account A
WHERE O.accNumber = A.accNumber AND
      A.branchName = 'Lonsdale' AND
      A.branchName <> 'Robson'
```

wrong

What does this query return?

Customers who own an account at Lonsdale (and note that Lonsdale is not the same as Robson …)

No Account in Robson Again

- Find the customerIDs of customers who have an account in the Lonsdale branch but don't have one in Robson
  - And get it right this time!

```
SELECT O1.customerID
FROM Owns O1, Account A1,  Owns O2, Account A2
WHERE O1.customerID = O2.customerID AND
            O1.accNumber = A1.accNumber AND
            O2.accNumber = A2.accNumber AND
            A1.branchName = 'Lonsdale' AND
            A2.branchName <> 'Robson'
```

wrong again!

What does this query return?

Customers who own *any* account that isn't at the Robson branch

Accounts EXCEPT Robson

- *This time* find the customerIDs of customers who have an account at the Lonsdale branch but not at Robson

```
SELECT O1.customerID
FROM Owns O1, Account A1
WHERE A1.accNumber = O1.accNumber AND
            A1.branchName = 'Lonsdale'
EXCEPT
SELECT O2.customerID
FROM Owns O2, Account A2
WHERE A2.accNumber = O2.accNumber AND
            A2.branchName = 'Robson'
```

Set Operations and Duplicates

- Unlike other SQL operations, UNION, INTERSECT, and EXCEPT queries eliminate duplicates by default
- SQL allows duplicates to be *retained* in these three operations using the ALL keyword
- If ALL is used, and there are *m* copies of a row in the upper query and *n* copies in the lower
  - UNION returns $m + n$ copies
  - INTERSECT returns min(m, n) copies
  - EXCEPT returns $m - n$ copies
- It is generally advisable not to specify ALL

EXCEPT, without ALL

$\pi_{\text{firstName,lastName}}$(Customer)

firstName	lastName
Arnold	Alliteration
Bob	Boyd
Bob	Boyd
Charlie	Clements
Bob	Boyd

$\pi_{\text{firstName,lastName}}$(Employee)

firstName	lastName
Bob	Boyd
Charlie	Clements
Susie	SummerTree
Desmond	Dorchester

```
SELECT firstName, lastName
FROM Customer
EXCEPT
SELECT firstName, lastName
FROM Employee
```

firstName	lastName
Arnold	Alliteration

EXCEPT ALL

$\pi_{firstName,lastName}(Customer)$

firstName	lastName
Arnold	Alliteration
Bob	Boyd
Bob	Boyd
Charlie	Clements
Bob	Boyd

$\pi_{firstName,lastName}(Employee)$

firstName	lastName
Bob	Boyd
Charlie	Clements
Susie	SummerTree
Desmond	Dorchester

```
SELECT firstName, lastName
FROM Customer
EXCEPT ALL
SELECT firstName, lastName
FROM Employee
```

firstName	lastName
Arnold	Alliteration
Bob	Boyd
Bob	Boyd

Joins

- SQL allows table to be *joined* in the **FROM** clause
- SQL joins implement relational algebra joins
  - Natural joins and theta joins are implemented by combining join *types* and *conditions*
- SQL also allows *outer joins* which retain records of one or both tables that do not match the condition
- Exact syntax and implementation of joins for a DBMS may differ from the SQL standard

Join Types

- **INNER** – only includes records where attributes from both tables meet the join condition
- **LEFT OUTER** – includes records from the *left table* that do not meet the join condition
- **RIGHT OUTER** – includes records from the *right table* that do not meet the join condition
- **FULL OUTER** – includes records from *both tables* that do not meet the join condition
- In outer joins, results are padded with **NULL** values for the attributes of records in only one of the tables

Join Conditions

- NATURAL – equality on all attributes in common
  - Similar to a relational algebra natural join
- USING (A_1, …, A_n) – equality on all the attributes in the attribute list
  - Similar to a relational algebra theta join on equality
- ON(condition) – join using condition
  - Similar to a relational algebra theta join
- Join conditions can be applied to outer or inner joins
  - If no condition is specified for an inner join the Cartesian product is returned
  - A condition must be specified for an outer join

Employees who are Customers

- Return the SINs and salaries of all employees, if they are customers also return their income

```
SELECT E.sin, E.salary, C.income
FROM Employee E LEFT OUTER JOIN Customer C ON
                         E.sin = C.customerID
```

- A *left outer join* is often preferred to a *right outer join*
  - So that nulls appear on the right hand side of the result

In this example the income column will contain nulls for those employees who are not also customers

Left Outer Join Results

all employees

sin	salary	income
111	29000	29000
222	73000	null
333	48000	null
444	83000	150000
555	48000	null
666	53000	53000
777	3200	null

income of customers who are also employees

```
SELECT E.sin, E.salary, C.income
FROM Employee E LEFT OUTER JOIN Customer C ON E.sin = C.customerID
```

Customer Accounts

- Return the customerIDs, first and last names, and account numbers of customers who own accounts

```
SELECT C.customerID, c.firstName, C.lastName, O.accNumber
FROM Customer C NATURAL INNER JOIN Owns O
```

No records will be returned for customers who do not have accounts

A natural join can be used here because the Owns and Customer table both contain attributes called *customerID*

Customers, also Employees?

- Return the SINs of employees, and customerIDs and first names of customers with the same last name

```
SELECT E.sin, C.customerID, C.firstName
FROM Employee E INNER JOIN Customer C USING (lastName)
```

In this case there will (probably) be many rows with repeated data for both the left and right tables

Nested SQL Queries

Nested Queries

- A nested query is a query that contains an embedded query, called a *sub-query*
- Sub-queries can appear in a number of places
  - In the **FROM** clause,
  - In the **WHERE** clause, and
  - In the **HAVING** clause
- Sub-queries referred to in the **WHERE** clause are often used in additional set operations
- Multiple levels of query nesting are allowed

Additional Set Operations

- IN
- NOT IN
- EXISTS
- NOT EXISTS
- UNIQUE
- ANY
- ALL

Accounts IN Lonsdale

- Find customerIDs, birth dates and incomes of customers with an account at the Lonsdale branch

```
SELECT C.customerID, C.birthDate, C.income
FROM Customer C
WHERE C.customerID IN
    (SELECT O.customerID
     FROM Account A, Owns A
     WHERE A.accNumber = O.accNumber AND
           A.branchName = 'Lonsdale')
```

Replacing IN with NOT IN in this query would return the customers who do not have an account at Lonsdale

Uncorrelated Queries

- The query shown previously contains an *uncorrelated*, or *independent*, sub-query
  - The sub-query does not contain references to attributes of the outer query
- An independent sub-query can be evaluated before evaluation of the outer query
  - And needs to be evaluated only once
    - The sub-query result can be checked for each row of the outer query
  - The cost is the cost for performing the sub-query (once) and the cost of scanning the outer relation

EXISTing Lonsdale Accounts

- Find customerIDs, birth dates and incomes of customers with an account at the Lonsdale branch

```
SELECT C.customerID, C.birthDate, C.income
FROM Customer C
WHERE EXISTS
          (SELECT *
           FROM Account A, Owns O
           WHERE C.customerID = O.customerID AND
                 A.accNumber = O.accNumber AND
                 A.branchName = 'Lonsdale')
```

EXISTS and NOT EXISTS test whether the associated sub-query is non-empty or empty

Correlated Queries

- The previous query contained a *correlated* sub-query
  - With references to attributes of the outer query
    - … WHERE C.customerID = O.customerID …
  - It is evaluated once *for each row* in the outer query
    - i.e. for each row in the Customer table
- Correlated queries are often inefficient
  - Unfortunately some DBMSs do not distinguish between the evaluation of correlated and uncorrelated queries

Division using NOT EXISTS

- Find the names and customerIDs of customers who have an account in all branches
  - This is an example of a query that would use division
  - However division is often not implemented in SQL
  - But can be computed using NOT EXISTS or EXCEPT
- To build a division query start by finding all the branch names
  - As this part is easy!

```
SELECT B.branchName
FROM Branch B
```

More Division and NOT EXISTS

- We can also find all of the branches that a particular customer has an account in

```
SELECT A.branchName
FROM Account A, Owns O
WHERE O.customerID = some_customer AND
            O.accNumber = A.accNumber
```

OK, so I'm cheating here by putting in "some customer" but we'll fix that part later using a correlated sub-query

Magic with EXCEPT

SQ1 – A list of all branch names

EXCEPT

SQ2 – A list of branch names that a customer has an account at

The result contains all of the branches that a customer does *not* have an account at, if the customer has an account at every branch then this result is empty

And Finally …

- Putting it all together we have

```
SELECT C.customerID , C.firstName, C.lastName
FROM Customer C
WHERE NOT EXISTS
          ((SELECT B.branchName
            FROM Branch B)
           EXCEPT
           (SELECT A.branchName
            FROM Account A, Owns O
            WHERE O.customerID = C.customerID AND
                      O.accNumber = A.accNumber))
```

UNIQUE and NOT UNIQUE

- The UNIQUE operator tests to see if there are no duplicate records in a query
  - UNIQUE returns TRUE if no row appears twice in the answer to the sub-query
    - Two rows are equal if, for each attribute, the attribute values in one row equal the values in the second
  - Also, if the sub-query contains only one row *or is empty* UNIQUE returns TRUE
- NOT UNIQUE tests to see if there are at least two identical rows in the sub-query

UNIQUE Accounts

- Find the customerIDs, first names, and last names of customers who have only one account

```
SELECT C.customerID, C.firstName, C.lastName
FROM Customer C
WHERE UNIQUE
          (SELECT O.customerID
           FROM Owns O
           WHERE C.customerID = O.customerID)
```

This is a correlated query, using NOT UNIQUE would return customers with at least two accounts

What happens if a customer doesn't have an account?

ANY and ALL

- The ANY (SOME in some DBMS) and ALL keywords allow comparisons to be made to *sets* of values
  - The keywords must be preceded by a Boolean operator
    - <, <=, =, <>, >=, or >
  - ANY and ALL are used in the WHERE clause to make a comparison to a sub-query
- ANY and ALL can be compared to IN and NOT IN
  - IN is equivalent to = ANY
  - NOT IN is equivalent to <> ALL

ANYone Richer Than Bruce

- Find the customerIDs and names, of customers who earn more than *any* customer called Bruce

```
SELECT C.customerID, C.firstName, C.lastName
FROM Customer C
WHERE C.income > ANY
          (SELECT Bruce.income
           FROM Customer Bruce
           WHERE Bruce.firstName = 'Bruce')
```

Customers in the result table must have incomes greater than at least one of the rows in the sub-query result

Richer Than ALL the Bruces

- Find the customerIDs and names, of customers who earn more than *any* customer called Bruce

```
SELECT C.customerID, C.firstName, C.lastName
FROM Customer C
WHERE C.income > ALL
          (SELECT Bruce.income
           FROM Customer Bruce
           WHERE Bruce.firstName = 'Bruce')
```

If there were no customers called Bruce this query would return all customers

Aggregations

Aggregate Operators

- SQL has a number of operators which compute *aggregate* values of columns
  - COUNT – the number of values in a column
  - SUM – the sum of the values in a column
  - AVG – the average of the values in a column
  - MAX – the maximum value in a column
  - MIN – the minimum value in a column
- DISTINCT can be applied to any of these operations but this should only be done with care!

AVG Income

- Find the average customer income

```
SELECT AVG (income) AS average_income
FROM Customer
```

The average column will be nameless unless it is given a name, hence the AS statement

Note that this is a query where using DISTINCT would presumably not give the intended results, as multiple identical incomes would only be included once

How Many Names For Smiths?

- Find the number of *different* first names for customers whose last name is Smith

```
SELECT COUNT (DISTINCT firstName) AS smith_names
FROM Customer
WHERE lastName = 'Smith' OR lastName = 'smith'
```

In this query it is important to use DISTINCT, otherwise the query will simply count the number of people whose last name is Smith

Aggregations and SELECT

- Find the customerID and income of the customer with the lowest income

SELECT customerID, MIN (income)
FROM Customer

Wrong

What is wrong with this query?

There may be two people with the same minimum income

This query is therefore illegal, if any aggregation is used in the SELECT clause it can only contain aggregations, unless the query also contains a GROUP BY clause

Aggregations and SELECT

- In the previous query a single aggregate value was, potentially, matched to a *set* of values
- In the query below, a set of pairs is returned
  - The income in each pair matches the single value returned by the sub-query

```
SELECT C1.customerID, C1.income
FROM Customer C1
WHERE C1.income =
          (SELECT MIN (C2.income)
           FROM Customer C2)
```

Grouping Aggregations

- Consider a query to find out how many accounts there are in *each* branch
  - This requires that there are multiple counts, one for each branch
  - In other words a series of aggregations, based on the value of the branch name
  - Given the syntax shown so far this is not possible to achieve in one query
  - But it is possible using the **GROUP BY** clause

Counting Accounts By Branch

- Find the number of accounts held by each branch

```
SELECT branchName, COUNT (accNumber) AS num_acc
FROM Account
GROUP BY branchName
```

Every column that appears in the SELECT list that is not an aggregation must also appear in the group list

Query Example – HAVING

- Find the number of accounts held by each branch
  - Only for branches that have budgets over $500,000 and total account balances greater than $1,000,000

```
SELECT B.branchName, COUNT (A.accNumber) AS accs
FROM Account A, Branch B
WHERE A.branchName = B.brNname AND
            B.budget > 500000
GROUP BY B.branchName
HAVING SUM (A.balance) > 1000000
```

The HAVING clause is a condition that is applied to each group rather than to each row

Order of Evaluation (Reprise)

- Create the Cartesian product of the tables in the **FROM** clause
- Remove rows not meeting the **WHERE** condition
- Remove columns not in the **SELECT** clause
- Sort records into groups by the **GROUP BY** clause
- Remove groups not meeting the **HAVING** clause
- Create one row for each group
- Eliminate duplicates if **DISTINCT** is specified

Query Optimization

A Brief Introduction

London Bob's Balance

- Find the customer IDs, last names and balances of customers called *Bob* who have an account at the *London* Branch

```
SELECT C.customerID, C.lastName, A.balance
FROM Customer C, Owns O, Account A
WHERE A.accNumber = O1.accNumber AND
      C.customerID = O.customerID AND
      A.branchName = 'London'  AND
      C.firstName = 'Bob'
```

join

This is a conceptually simple query, but how expensive is it (that is how long does it take to run)?

Data

- Some data about the (small) bank's data
  - 100,000 customer records in 10,000 disk pages
    - Barclays bank had 12 million UK customers in 2013
  - 120,000 accounts in 10,000 disk pages
  - 200,000 owns records in 2,000 disk pages
  - 10 ms to access a block (page) on a hard drive
  - 1,000 main memory pages available for the query
- Where did these numbers come from?
  - I made them up
    - But they are not completely unreasonable ...

Why Disk Reads

- The cost metric to be used is the number of disk reads and writes required for the query
  - Because reading a block from a disk is much slower than performing main memory operations
    - About 250,000 times slower!
  - So we will just consider the disk access costs of the query and ignore main memory costs
- 10ms is a reasonable estimate for the time to read 1 block from a hard disk
  - But reading multiple blocks is usually faster per block

Processing the Query

```
SELECT C.customerID, C.lastName, A.balance
FROM Customer C, Owns O, Account A
WHERE A.accNumber = O1.accNumber AND
      C.customerID = O.customerID AND
      A.branchName = 'London'  AND
      C.firstName = 'Bob'
```

- The process this query describes is
  - Compute the Cartesian product of the Customer, Owns and Account tables
  - Select records that match the condition
  - Remove all columns except those in the select list

Cartesian Product

- The Cartesian product operation is a binary operation so requires two tables as its operands
- We need an algorithm to compute the product
  - Every row in one table must be concatenated with every row in the other table
    - Doing this one row at a time would be bad …
  - Read as much of one table into main memory as possible (approximately 1,000 blocks)
  - Then scan the other table once for each such set of records

Customer × Owns

- For each 1,000 block portion of Owns
  - Scan the entire Customer table
  - Output the concatenated records
- Cost in disk reads or writes
  - Read Owns once – 2,000 reads
  - Read Customer twice – 20,000 reads
  - Write out result relation once – $2*10^9$ writes

Why $2*10^9$?

There are 200,000 * 100,000 = $2*10^{10}$ records in the result, let's say each is the same size as a Customer record so that's $2*10^{10}$ / 10 = $2*10^9$ blocks

(Customer × Owns) × Account

- For each 1,000 block portion of Account
  - Scan the entire Customer-Owns (CO) relation
  - Output the concatenated records
- Cost in disk reads or writes
  - Read Account once – 10,000 reads
  - Read CO 10 times – $2*10^{10}$ reads
  - What's the size of the result?

$2*10^{10} * 120,000 = 2.4*10^{15}$. Each record has all of the attributes from Customer, Owns and Account so a reasonable assumption is that there are around 5 records per block. Fortunately this doesn't really matter …

Applying Other Operations

- Apply other operations as Customer-Owns-Account (COA) records are computed
  - Instead of writing out the entire COA relation and then reading it in again
- The selection and projection therefore do not require any additional disk reads or writes
  - Since they are applied *on the fly*

```
SELECT C.customerID, C.lastName, A.balance
FROM Customer C, Owns O, Account A
WHERE A.accNumber = O1.accNumber AND
      C.customerID = O.customerID AND
      A.branchName = 'London'  AND
      C.firstName = 'Bob'
```

Total Cost

- We could add up all of the costs but the cost of repeatedly scanning the CO relation in the second product dominates
  - The CO relation was read 10 times to compute the COA relation
  - For a cost of $2*10^{10}$ disk reads
- Recall that each disk read takes 10 ms
  - $2*10^{10}$ / 100 = $2*10^{8}$ seconds
  - $2*10^{8}$ / 60 = 3.33 $*10^{6}$ minutes
  - $3.33*10^{6}$ / 60 = 55,556 hours
  - 55,556 / 24 = 2,315 days
  - 2,315 / 365 = 6.34 years

```
SELECT C.customerID, C.lastName, A.balance
FROM Customer C, Owns O, Account A
WHERE A.accNumber = O1.accNumber AND
      C.customerID = O.customerID AND
      A.branchName = 'London' AND
      C.firstName = 'Bob'
```

An Equivalent Query

- This query is equivalent to the original query

```
SELECT C.customerID, C.lastName, A.balance
FROM (SELECT accNumber, balance
          FROM Account
          WHERE branchName = 'London' ) AS A
          NATURAL INNER JOIN Owns
          NATUAL INNER JOIN
          (SELECT customerID, lastName
          FROM Customer
          WHERE C.firstName = 'Bob') AS C
```

More complex than the original but is it more efficient?

Equivalent Query Changes

- The new query makes two major changes
  - The selections and some preliminary projections occur before any relations are joined
  - And the Cartesian products are replaced by joins
- The two queries are equivalent
  - They compute the same result
  - But in different ways
    - And the order in which the operations are performed is very different

Account Sub-Query Cost

- Estimating the cost of the Account sub-query is difficult without additional information
  - However, the upper limit on its cost is the size of the Account relation – 10,000 block reads
  - Since the selection and projection can be satisfied in a single scan of the entire Account table
- More importantly the size of the result of these initial operations is much smaller
  - How much smaller?

```
SELECT accNumber, balance
FROM Account
WHERE branchName = 'London'
```

How Many London Accounts

- Estimate the number of London Accounts as 15% of the total
  - 18,000 Accounts
- The query's schema has just two attributes
  - Let's assume 100 records fit on one block
    - As each record is similar in size to an Owns record
  - The result is contained in just 180 blocks

```
SELECT accNumber, balance
FROM Account
WHERE branchName = 'London'
```

Customer Sub-Query Cost

- We can follow a similar process for the Customer sub-query
  - At worst the cost is of the query is 10,000 reads
- The size of the result depends on the size of each record and the number of *Bobs*
  - The first name condition is very selective
    - Babies were given 62,000 different first names in 2014
  - Let's assume that 0.01% of first names are Bob
  - 10 records on 1 disk block!

```
SELECT customerID, lastName
FROM Customer
WHERE C.firstName = 'Bob'
```

Estimated Total Cost

- The estimated total cost of the new query is roughly equal to the sum of the table sizes
  - Since Account was scanned to perform a sub-query
  - And Customer was scanned for its sub-query
  - The first join, at worst, requires reading all of Owns once
  - Then joining the relatively small result of that join to Account

Estimated Total Cost

- Total cost in the order of 22,000 disk reads
  - Let's be extravagant and call it 30,000
  - 30,000 / 100 = 300 seconds
  - 300 seconds = 5 minutes
    - This may seem like a long time but not as long as 6 years …
- Indexes might significantly reduce this cost
  - By how much depends on the type of index
    - An index on first name would greatly reduce the cost of the Customer selection
    - However, an index on city would have to be a clustered index to reduce the cost of the Account selection

Query Optimization

- Query optimization is the process of finding an efficient equivalent query and entails
  - Converting the original SQL query to relational algebra
  - Finding equivalent queries
  - Estimating the cost of each of the queries
    - Each query can have multiple different costs since there is more than one algorithm for many operations
  - Selecting the most efficient query
- Modern DBMS automatically optimize queries
  - This is a good thing ...