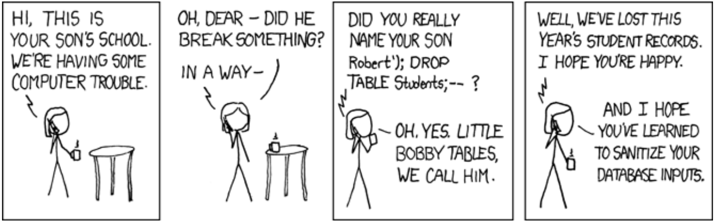
SQL Tutorial

A Brief Introduction

Darrell Aucoin

Stats Club

October 13, 2014



Contents

1. What is SQL?
2. Relational Algebra
3. Clauses
4. Data Types
5. Constraints
6. Filter Clauses
7. Grouping and Aggregate Functions
8. Joins
9. Subqueries
10. Set Operators
11. Functions and Operators
12. Using SQL with R
13. Thanks

What Do We Want to Do With Data?

Important

Data processing/manipulation should only be done after we formulate a question and find relevant data to our question.

What Do We Want to Do With Data?

Important

Data processing/manipulation should only be done after we formulate a question and find relevant data to our question.

Filter Data: Filter out entities not relevant to our question.

What Do We Want to Do With Data?

Important

Data processing/manipulation should only be done after we formulate a question and find relevant data to our question.

Filter Data: Filter out entities not relevant to our question.

- WHERE/HAVING clause in SQL

What Do We Want to Do With Data?

Important

Data processing/manipulation should only be done after we formulate a question and find relevant data to our question.

Filter Data: Filter out entities not relevant to our question.

- WHERE/HAVING clause in SQL

Select Attributes: Narrow down on attributes of interest on the entities we want to examine.

What Do We Want to Do With Data?

Important

Data processing/manipulation should only be done after we formulate a question and find relevant data to our question.

Filter Data: Filter out entities not relevant to our question.

- WHERE/HAVING clause in SQL

Select Attributes: Narrow down on attributes of interest on the entities we want to examine.

- SELECT clause in SQL

What Do We Want to Do With Data?

Important

Data processing/manipulation should only be done after we formulate a question and find relevant data to our question.

Filter Data: Filter out entities not relevant to our question.

- WHERE/HAVING clause in SQL

Select Attributes: Narrow down on attributes of interest on the entities we want to examine.

- SELECT clause in SQL

Manipulate Variables: Do calculations, concatenate strings, change variables to different datatypes.

What Do We Want to Do With Data?

Important

Data processing/manipulation should only be done after we formulate a question and find relevant data to our question.

Filter Data: Filter out entities not relevant to our question.

- WHERE/HAVING clause in SQL

Select Attributes: Narrow down on attributes of interest on the entities we want to examine.

- SELECT clause in SQL

Manipulate Variables: Do calculations, concatenate strings, change variables to different datatypes.

- Done with various functions in SQL

What Do We Want to Do With Data?

Important

Data processing/manipulation should only be done after we formulate a question and find relevant data to our question.

Filter Data: Filter out entities not relevant to our question.

- WHERE/HAVING clause in SQL

Select Attributes: Narrow down on attributes of interest on the entities we want to examine.

- SELECT clause in SQL

Manipulate Variables: Do calculations, concatenate strings, change variables to different datatypes.

- Done with various functions in SQL

Summarize Values: Perform counts, sums, averages, etc. on your data.

What Do We Want to Do With Data?

Important

Data processing/manipulation should only be done after we formulate a question and find relevant data to our question.

Filter Data: Filter out entities not relevant to our question.

- WHERE/HAVING clause in SQL

Select Attributes: Narrow down on attributes of interest on the entities we want to examine.

- SELECT clause in SQL

Manipulate Variables: Do calculations, concatenate strings, change variables to different datatypes.

- Done with various functions in SQL

Summarize Values: Perform counts, sums, averages, etc. on your data.

- Aggregate functions associated with GROUP BY clause in SQL

What Do We Want to Do With Data?

Combine Information: Piece information together from other sources

What Do We Want to Do With Data?

Combine Information: Piece information together from other sources

- JOIN clauses in SQL, joining two tables together

What Do We Want to Do With Data?

Combine Information: Piece information together from other sources

- JOIN clauses in SQL, joining two tables together

Arrange Data: Sort our data to help identify patterns

What Do We Want to Do With Data?

Combine Information: Piece information together from other sources

- JOIN clauses in SQL, joining two tables together

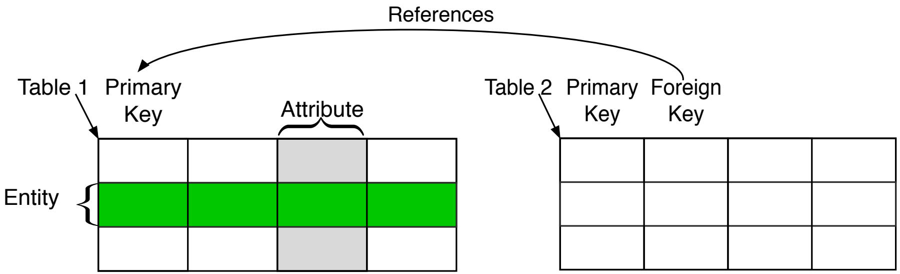
Arrange Data: Sort our data to help identify patterns

- ORDER BY clause in SQL

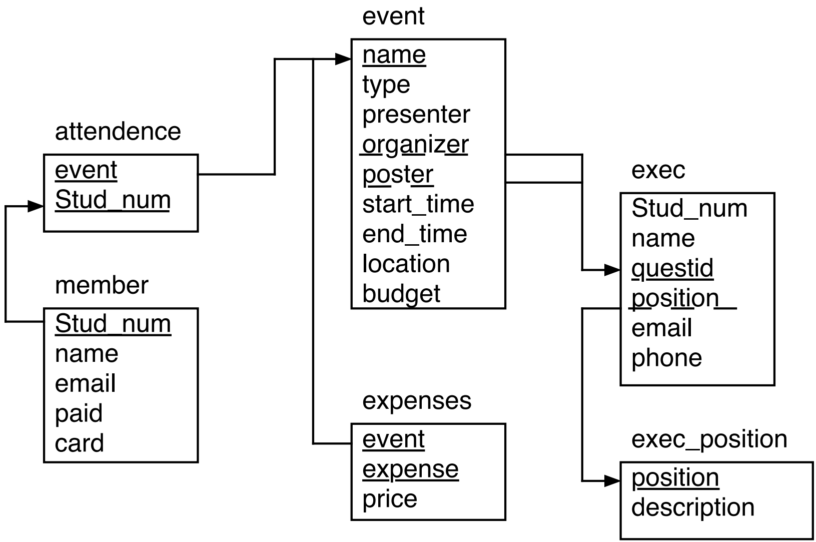
Relational Databases

Definition

Relational Database: A relational database is a system organized in tables containing entities (rows) related to other entries in other tables. Each entity having attributes (columns also called fields) which give additional information on an entity.



Relational Databases



SQL is a non-procedural language, meaning the order of execution of clauses is not determined by the SQL language but rather the programmer specifies what needs to be done and is sent to an optimizer which handles how to do the task.

SQL is made of sub-languages:

1. Data Manipulation Language

The SQL Language

SQL is a non-procedural language, meaning the order of execution of clauses is not determined by the SQL language but rather the programmer specifies what needs to be done and is sent to an optimizer which handles how to do the task.

SQL is made of sub-languages:

1. Data Manipulation Language

   SELECT Statements that perform queries

SQL is a non-procedural language, meaning the order of execution of clauses is not determined by the SQL language but rather the programmer specifies what needs to be done and is sent to an optimizer which handles how to do the task.

SQL is made of sub-languages:

1. Data Manipulation Language

   SELECT Statements that perform queries

   INSERT, UPDATE, DELETE Statements that modify the instance of a table

The SQL Language

SQL is a non-procedural language, meaning the order of execution of clauses is not determined by the SQL language but rather the programmer specifies what needs to be done and is sent to an optimizer which handles how to do the task.

SQL is made of sub-languages:

1. Data Manipulation Language

   SELECT Statements that perform queries

   INSERT, UPDATE, DELETE Statements that modify the instance of a table

2. Data Definition Language

The SQL Language

SQL is a non-procedural language, meaning the order of execution of clauses is not determined by the SQL language but rather the programmer specifies what needs to be done and is sent to an optimizer which handles how to do the task.

SQL is made of sub-languages:

1. Data Manipulation Language

   - SELECT: Statements that perform queries
   - INSERT, UPDATE, DELETE: Statements that modify the instance of a table

2. Data Definition Language

   - CREATE, DROP, ALTER: Statements that modify the database schema

The SQL Language

SQL is a non-procedural language, meaning the order of execution of clauses is not determined by the SQL language but rather the programmer specifies what needs to be done and is sent to an optimizer which handles how to do the task.

SQL is made of sub-languages:

1. Data Manipulation Language

   SELECT: Statements that perform queries

   INSERT, UPDATE, DELETE: Statements that modify the instance of a table

2. Data Definition Language

   CREATE, DROP, ALTER: Statements that modify the database schema

3. Data Control Language

The SQL Language

SQL is a non-procedural language, meaning the order of execution of clauses is not determined by the SQL language but rather the programmer specifies what needs to be done and is sent to an optimizer which handles how to do the task.

SQL is made of sub-languages:

1. Data Manipulation Language

   SELECT Statements that perform queries

   INSERT, UPDATE, DELETE Statements that modify the instance of a table

2. Data Definition Language

   CREATE, DROP, ALTER Statements that modify the database schema

3. Data Control Language

   GRANT, REVOKE Statements that enforce the security model

Conventions

For the purposes of this talk:

- SQL commands are in UPPER CASE, SQL in general is not case sensitive

Conventions

For the purposes of this talk:

- SQL commands are in UPPER CASE, SQL in general is not case sensitive

  <> will display tables name, variables names, variable types

Conventions

For the purposes of this talk:

- SQL commands are in UPPER CASE, SQL in general is not case sensitive

<> will display tables name, variables names, variable types

[] will mean optional arguments

Conventions

For the purposes of this talk:

- SQL commands are in UPPER CASE, SQL in general is not case sensitive

<> will display tables name, variables names, variable types

[] will mean optional arguments

... will just mean additional code

Conventions

For the purposes of this talk:

- SQL commands are in UPPER CASE, SQL in general is not case sensitive

  <> will display tables name, variables names, variable types

  [] will mean optional arguments

  ... will just mean additional code

- Note on SQL Coding

Conventions

For the purposes of this talk:

- SQL commands are in UPPER CASE, SQL in general is not case sensitive

  <> will display tables name, variables names, variable types

  [] will mean optional arguments

  ... will just mean additional code

- Note on SQL Coding

  ; ends a command, a command is interpreted as a whole and only executed after it reads the ;

Conventions

For the purposes of this talk:

- SQL commands are in UPPER CASE, SQL in general is not case sensitive

  - **<>** will display tables name, variables names, variable types
  - **[]** will mean optional arguments
  - **...** will just mean additional code

- Note on SQL Coding

  - **;** ends a command, a command is interpreted as a whole and only executed after it reads the ;
    - The uses of the newline is only for readability purposes.

Conventions

For the purposes of this talk:

- SQL commands are in UPPER CASE, SQL in general is not case sensitive

<>	will display tables name, variables names, variable types
[]	will mean optional arguments
...	will just mean additional code

- Note on SQL Coding

  ; ends a command, a command is interpreted as a whole and only executed after it reads the ;

  - The uses of the newline is only for readability purposes.

/* ... */ Programming comments (in most SQL implementations).

Example

```
mysql> SELECT name, position
    -> FROM exec;
```

Example

```
mysql> SELECT name, position
    -> FROM exec;

+-----------------------------+-------------------+
| name                        | position          |
+-----------------------------+-------------------+
| Ajanthan Thavaraja (Aj)     | Events            |
| Darrell Aucoin              | President         |
| JinCheng Wong               | Events            |
| Massey Cashore              | Events            |
| Jacob Burns                 | President         |
| Ming Pan                    | Technology        |
| Zixin Nie                   | Events            |
| Simon wang                  | Senior Advisor    |
| Alice Wang                  | Finance           |
+-----------------------------+-------------------+
9 rows in set (0.00 sec)
```

Definitions

Entity Something of interest in the data base: a person, account, etc.

Definitions

Entity Something of interest in the data base: a person, account, etc.

Column (Attribute, Field) An individual piece of data stored in a table

Definitions

Entity Something of interest in the data base: a person, account, etc.

Column (Attribute, Field) An individual piece of data stored in a table

Row (Record) A tuple of columns describing an entity or action of an entity.

Definitions

Entity Something of interest in the data base: a person, account, etc.

Column (Attribute, Field) An individual piece of data stored in a table

Row (Record) A tuple of columns describing an entity or action of an entity.

Table A collection of rows.

Definitions

Entity Something of interest in the data base: a person, account, etc.

Column (Attribute, Field) An individual piece of data stored in a table

Row (Record) A tuple of columns describing an entity or action of an entity.

Table A collection of rows.

- Usually in reference a persistent saved permanently to memory

Definitions

Entity Something of interest in the data base: a person, account, etc.

Column (Attribute, Field) An individual piece of data stored in a table

Row (Record) A tuple of columns describing an entity or action of an entity.

Table A collection of rows.

- Usually in reference a persistent saved permanently to memory

  Result set A non-persistent table, usually the result of a query.

Definitions

Entity Something of interest in the data base: a person, account, etc.

Column (Attribute, Field) An individual piece of data stored in a table

Row (Record) A tuple of columns describing an entity or action of an entity.

Table A collection of rows.

- Usually in reference a persistent saved permanently to memory

  Result set A non-persistent table, usually the result of a query.

  Subquery A query that returns a table to another query.

Definitions

Entity Something of interest in the data base: a person, account, etc.

Column (Attribute, Field) An individual piece of data stored in a table

Row (Record) A tuple of columns describing an entity or action of an entity.

Table A collection of rows.

- Usually in reference a persistent saved permanently to memory

  Result set A non-persistent table, usually the result of a query.

  Subquery A query that returns a table to another query.

  (Virtual) View A named query saved into memory performed whenever it is named. Some SQL servers have materialized views that permanently save the data for faster access.

Primary key A tuple of columns that uniquely define each row in a table.

Definitions

Primary key A tuple of columns that uniquely define each row in a table.

Foreign key A tuple of columns identifying a relationship to another table.

Important Information on SQL

Additional information:

- SQL is designed with the relational model (see: relational algebra) in mind

Important Information on SQL

Additional information:

- SQL is designed with the relational model (see: relational algebra) in mind
- SQL is **normalized**: Tables contain only primitive data types: strings, numbers, dates, etc.

Important Information on SQL

Additional information:

- SQL is designed with the relational model (see: relational algebra) in mind
- SQL is **normalized**: Tables contain only primitive data types: strings, numbers, dates, etc.
  - A table cannot contain a relation or another table as an attribute (the table is 'flat')

Important Information on SQL

Additional information:

- SQL is designed with the relational model (see: relational algebra) in mind
- SQL is **normalized**: Tables contain only primitive data types: strings, numbers, dates, etc.
  - A table cannot contain a relation or another table as an attribute (the table is 'flat')
- A good database is designed so that there is no redundancy in data:

Important Information on SQL

Additional information:

- SQL is designed with the relational model (see: relational algebra) in mind
- SQL is **normalized**: Tables contain only primitive data types: strings, numbers, dates, etc.
  - A table cannot contain a relation or another table as an attribute (the table is 'flat')
- A good database is designed so that there is no redundancy in data:
  - A specific piece of data for a specific entity is only ever in one place in one table (unless it's a key)

Important Information on SQL

Additional information:

- SQL is designed with the relational model (see: relational algebra) in mind
- SQL is **normalized**: Tables contain only primitive data types: strings, numbers, dates, etc.
    - A table cannot contain a relation or another table as an attribute (the table is 'flat')
- A good database is designed so that there is no redundancy in data:
    - A specific piece of data for a specific entity is only ever in one place in one table (unless it's a key)
- Relationships in SQL can be described as:

Important Information on SQL

Additional information:

- SQL is designed with the relational model (see: relational algebra) in mind
- SQL is **normalized**: Tables contain only primitive data types: strings, numbers, dates, etc.
  - A table cannot contain a relation or another table as an attribute (the table is 'flat')
- A good database is designed so that there is no redundancy in data:
  - A specific piece of data for a specific entity is only ever in one place in one table (unless it's a key)
- Relationships in SQL can be described as:
  1. **One to one relationships**: A student has one mailing address

Important Information on SQL

Additional information:

- SQL is designed with the relational model (see: relational algebra) in mind
- SQL is **normalized**: Tables contain only primitive data types: strings, numbers, dates, etc.
    - A table cannot contain a relation or another table as an attribute (the table is 'flat')
- A good database is designed so that there is no redundancy in data:
    - A specific piece of data for a specific entity is only ever in one place in one table (unless it's a key)
- Relationships in SQL can be described as:
    1. **One to one relationships**: A student has one mailing address
    2. **One to many relationships**: A student takes many courses per term

Important Information on SQL

Additional information:

- SQL is designed with the relational model (see: relational algebra) in mind
- SQL is **normalized**: Tables contain only primitive data types: strings, numbers, dates, etc.
  - A table cannot contain a relation or another table as an attribute (the table is 'flat')
- A good database is designed so that there is no redundancy in data:
  - A specific piece of data for a specific entity is only ever in one place in one table (unless it's a key)
- Relationships in SQL can be described as:
  1. **One to one relationships**: A student has one mailing address
  2. **One to many relationships**: A student takes many courses per term
  3. **Many to one relationships**: A course section has many students

Important Information on SQL

Additional information:

- SQL is designed with the relational model (see: relational algebra) in mind
- SQL is **normalized**: Tables contain only primitive data types: strings, numbers, dates, etc.
  - A table cannot contain a relation or another table as an attribute (the table is 'flat')
- A good database is designed so that there is no redundancy in data:
  - A specific piece of data for a specific entity is only ever in one place in one table (unless it's a key)
- Relationships in SQL can be described as:
  1. **One to one relationships**: A student has one mailing address
  2. **One to many relationships**: A student takes many courses per term
  3. **Many to one relationships**: A course section has many students
  4. **Many to many relationships**: A University has many students and students may have many Universities (Waterloo and Laurier)

Different Implementations of SQL

MySQL: Highly popular open source SQL implementation. Most of the examples here use MySQL.

Different Implementations of SQL

MySQL: Highly popular open source SQL implementation. Most of the examples here use MySQL.

PostgreSQL: Open source SQL designed around letting users create User Defined Functions (UDF).

Different Implementations of SQL

MySQL: Highly popular open source SQL implementation. Most of the examples here use MySQL.

PostgreSQL: Open source SQL designed around letting users create User Defined Functions (UDF).

SQLite: Open sources light weight SQL usually used as an embedded database for applications (web browsers, mobile applications, etc.), or light to medium traffic websites. Database is saved as a single file making it a good alternative to csv for sharing data. Various OS's have SQLite preinstalled (type sqlite3 in terminal for mac)

Different Implementations of SQL

MySQL: Highly popular open source SQL implementation. Most of the examples here use MySQL.

PostgreSQL: Open source SQL designed around letting users create User Defined Functions (UDF).

SQLite: Open sources light weight SQL usually used as an embedded database for applications (web browsers, mobile applications, etc.), or light to medium traffic websites. Database is saved as a single file making it a good alternative to csv for sharing data. Various OS's have SQLite preinstalled (type sqlite3 in terminal for mac)

Oracle: SQL implementation produced and marketed by Oracle Corporation.

Different Implementations of SQL

MySQL: Highly popular open source SQL implementation. Most of the examples here use MySQL.

PostgreSQL: Open source SQL designed around letting users create User Defined Functions (UDF).

SQLite: Open sources light weight SQL usually used as an embedded database for applications (web browsers, mobile applications, etc.), or light to medium traffic websites. Database is saved as a single file making it a good alternative to csv for sharing data. Various OS's have SQLite preinstalled (type sqlite3 in terminal for mac)

Oracle: SQL implementation produced and marketed by Oracle Corporation.

Microsoft SQL Server: SQL implementation developed by Microsoft.

Different Implementations of SQL

MySQL: Highly popular open source SQL implementation. Most of the examples here use MySQL.

PostgreSQL: Open source SQL designed around letting users create User Defined Functions (UDF).

SQLite: Open sources light weight SQL usually used as an embedded database for applications (web browsers, mobile applications, etc.), or light to medium traffic websites. Database is saved as a single file making it a good alternative to csv for sharing data. Various OS's have SQLite preinstalled (type sqlite3 in terminal for mac)

Oracle: SQL implementation produced and marketed by Oracle Corporation.

Microsoft SQL Server: SQL implementation developed by Microsoft.

DB2: SQL developed by IBM. The database used by University of Waterloo.

Different Implementations of SQL

MySQL: Highly popular open source SQL implementation. Most of the examples here use MySQL.

PostgreSQL: Open source SQL designed around letting users create User Defined Functions (UDF).

SQLite: Open sources light weight SQL usually used as an embedded database for applications (web browsers, mobile applications, etc.), or light to medium traffic websites. Database is saved as a single file making it a good alternative to csv for sharing data. Various OS's have SQLite preinstalled (type sqlite3 in terminal for mac)

Oracle: SQL implementation produced and marketed by Oracle Corporation.

Microsoft SQL Server: SQL implementation developed by Microsoft.

DB2: SQL developed by IBM. The database used by University of Waterloo.

- Each implementation of SQL is slightly different

Basics of Relational Algebra

SQL uses what is called **Relational Algebra**: a set of operations on tables that in turn returns a table. This is analogous to regular algebra where we take in a number perform an operation and then return a number.

- This homogeneity is the critical aspect to the power of relational databases and SQL

SQL uses what is called **Relational Algebra**: a set of operations on tables that in turn returns a table. This is analogous to regular algebra where we take in a number perform an operation and then return a number.

- This homogeneity is the critical aspect to the power of relational databases and SQL

Basics of Relational Algebra

SQL uses what is called **Relational Algebra**: a set of operations on tables that in turn returns a table. This is analogous to regular algebra where we take in a number perform an operation and then return a number.

- This homogeneity is the critical aspect to the power of relational databases and SQL

First the data is normalized: removing redundant data and repeating groups of data, then a set of relational algebra operations can be applied.

Relational Algebra Operations

Projection (π): Returns a subset of columns.

Selection (σ): Returns only entities where some condition is true.

Rename (ρ): Rename an attribute.

Natural Join ($\bowtie$): Tuples from one table is joined to tuples from another table based on common attributes (at least one column with the same name and possible values is common between them)

θ-Join and Equijoin: Join tuples from two different tables where some binary condition ($\theta = \{\geq, \leq, <, >, =\}$) between two tables attributes is true. When θ is $=$, the join is called an equijoin.

Set Operations: Set theory's unions, set difference, and cartesian product of tuples performed on tuples of different tables.

$\vdots$

Power of the Relational Model

These relational algebra operations can be rearranged much like regular algebra (distributive law, commutative law etc.). The SQL optimization engine finds various combinations of these operations via the relational algebra model and selects the combination that (usually) has the lowest computational cost.

- SQL optimization engine allows the user to simply define their question and not have to worry (too much) about performance

Power of the Relational Model

These relational algebra operations can be rearranged much like regular algebra (distributive law, commutative law etc.). The SQL optimization engine finds various combinations of these operations via the relational algebra model and selects the combination that (usually) has the lowest computational cost.

- SQL optimization engine allows the user to simply define their question and not have to worry (too much) about performance
- This homogeneity of operations not only allow powerful chains of operations to occur, but allows the user to perform queries that even relational algebra's creator Edgar F. Codd could not think of.

Power of the Relational Model

These relational algebra operations can be rearranged much like regular algebra (distributive law, commutative law etc.). The SQL optimization engine finds various combinations of these operations via the relational algebra model and selects the combination that (usually) has the lowest computational cost.

- SQL optimization engine allows the user to simply define their question and not have to worry (too much) about performance
- This homogeneity of operations not only allow powerful chains of operations to occur, but allows the user to perform queries that even relational algebra's creator Edgar F. Codd could not think of.
    - Queries performing linear regression, and even Monte Carlo simulations are possible using SQL

Power of the Relational Model

These relational algebra operations can be rearranged much like regular algebra (distributive law, commutative law etc.). The SQL optimization engine finds various combinations of these operations via the relational algebra model and selects the combination that (usually) has the lowest computational cost.

- SQL optimization engine allows the user to simply define their question and not have to worry (too much) about performance
- This homogeneity of operations not only allow powerful chains of operations to occur, but allows the user to perform queries that even relational algebra's creator Edgar F. Codd could not think of.
    - Queries performing linear regression, and even Monte Carlo simulations are possible using SQL
    - It is usually better to get SQL to do these operations on the server

Power of the Relational Model

These relational algebra operations can be rearranged much like regular algebra (distributive law, commutative law etc.). The SQL optimization engine finds various combinations of these operations via the relational algebra model and selects the combination that (usually) has the lowest computational cost.

- SQL optimization engine allows the user to simply define their question and not have to worry (too much) about performance
- This homogeneity of operations not only allow powerful chains of operations to occur, but allows the user to perform queries that even relational algebra's creator Edgar F. Codd could not think of.
  - Queries performing linear regression, and even Monte Carlo simulations are possible using SQL
  - It is usually better to get SQL to do these operations on the server
  - Queries in practice can get to be 100+ lines

Power of the Relational Model

These relational algebra operations can be rearranged much like regular algebra (distributive law, commutative law etc.). The SQL optimization engine finds various combinations of these operations via the relational algebra model and selects the combination that (usually) has the lowest computational cost.

- SQL optimization engine allows the user to simply define their question and not have to worry (too much) about performance
- This homogeneity of operations not only allow powerful chains of operations to occur, but allows the user to perform queries that even relational algebra's creator Edgar F. Codd could not think of.
    - Queries performing linear regression, and even Monte Carlo simulations are possible using SQL
    - It is usually better to get SQL to do these operations on the server
    - Queries in practice can get to be 100+ lines
- Relational Algebra uses set notation but SQL uses bag semantics

Definition

SELECT Statement: The SELECT statement returns a table of values (sometimes empty) as specified in the statement.

```
SELECT <listOfColumn(s)>
FROM <table>
[WHERE <condition>];
```

Data Definition: SELECT

Definition

SELECT Statement: The SELECT statement returns a table of values (sometimes empty) as specified in the statement.

```
SELECT <listOfColumn(s)>
FROM <table>
[WHERE <condition>];
```

- Use * instead of <listOfColumn(s)> if you wish to return all columns

Data Definition: SELECT

Definition

SELECT Statement: The SELECT statement returns a table of values (sometimes empty) as specified in the statement.

```
SELECT <listOfColumn(s)>
FROM <table>
[WHERE <condition >];
```

- Use * instead of <listOfColumn(s)> if you wish to return all columns
- We can also use expressions and functions in <listOfColumn(s)> to return values based on the resultant of each row.

Data Definition: SELECT

Definition

SELECT Statement: The SELECT statement returns a table of values (sometimes empty) as specified in the statement.

```
SELECT <listOfColumn(s)>
FROM <table>
[WHERE <condition>];
```

- Use * instead of <listOfColumn(s)> if you wish to return all columns
- We can also use expressions and functions in <listOfColumn(s)> to return values based on the resultant of each row.
  - Aggregate functions, on the other hand, take in many entries (sometimes grouped) and return a value per group

SELECT Example

```
mysql> SELECT name, start_time, location
    -> FROM event;
```

SELECT Example

```
mysql> SELECT name, start_time, location
    -> FROM event;

+-----------------+---------------------+-----------+
| name            | start_time          | location  |
+-----------------+---------------------+-----------+
| BOT             | 2014-09-23 18:00:00 | MC Comfy  |
| EOT             | NULL                | NULL      |
| Intro to Hadoop | NULL                | NULL      |
| Intro to Pig    | NULL                | NULL      |
| Intro to SQL    | 2014-10-16 17:30:00 | M3-3103   |
| Prof Talk       | 2014-10-23 15:00:00 | MC 1085   |
| R Tutorial      | NULL                | NULL      |
+-----------------+---------------------+-----------+
7 rows in set (0.00 sec)
```

Usual Execution Order of SELECT Statement

The **usual** order of execution of a SELECT statement:

1. **FROM**: Identifies what table to retrieve the data from

Usual Execution Order of SELECT Statement

The **usual** order of execution of a SELECT statement:

1. **FROM**: Identifies what table to retrieve the data from
2. **ON**: (used with FROM) Describes how to join tables

Usual Execution Order of SELECT Statement

The **usual** order of execution of a SELECT statement:

1. **FROM**: Identifies what table to retrieve the data from
2. **ON**: (used with FROM) Describes how to join tables
3. **OUTER**: Related to JOIN

Usual Execution Order of SELECT Statement

The **usual** order of execution of a SELECT statement:

1. **FROM**: Identifies what table to retrieve the data from
2. **ON**: (used with FROM) Describes how to join tables
3. **OUTER**: Related to JOIN
4. **WHERE**: A filter clause

Usual Execution Order of SELECT Statement

The **usual** order of execution of a SELECT statement:

1. **FROM**: Identifies what table to retrieve the data from
2. **ON**: (used with FROM) Describes how to join tables
3. **OUTER**: Related to JOIN
4. **WHERE**: A filter clause
5. **GROUP BY**: Group the data by one or more attributes (columns)

Usual Execution Order of SELECT Statement

The **usual** order of execution of a SELECT statement:

1. **FROM**: Identifies what table to retrieve the data from
2. **ON**: (used with FROM) Describes how to join tables
3. **OUTER**: Related to JOIN
4. **WHERE**: A filter clause
5. **GROUP BY**: Group the data by one or more attributes (columns)
6. **ROLLUP** | **CUBE**

Usual Execution Order of SELECT Statement

The **usual** order of execution of a SELECT statement:

1. **FROM**: Identifies what table to retrieve the data from
2. **ON**: (used with FROM) Describes how to join tables
3. **OUTER**: Related to JOIN
4. **WHERE**: A filter clause
5. **GROUP BY**: Group the data by one or more attributes (columns)
6. **ROLLUP** | **CUBE**
7. **HAVING**: A filter clause, usually related to the GROUP BY clause, allowing aggregate functions in the filter clause

Usual Execution Order of SELECT Statement

The **usual** order of execution of a SELECT statement:

1. **FROM**: Identifies what table to retrieve the data from
2. **ON**: (used with FROM) Describes how to join tables
3. **OUTER**: Related to JOIN
4. **WHERE**: A filter clause
5. **GROUP BY**: Group the data by one or more attributes (columns)
6. **ROLLUP** | **CUBE**
7. **HAVING**: A filter clause, usually related to the GROUP BY clause, allowing aggregate functions in the filter clause
8. **SELECT**

Usual Execution Order of SELECT Statement

The **usual** order of execution of a SELECT statement:

1. **FROM**: Identifies what table to retrieve the data from
2. **ON**: (used with FROM) Describes how to join tables
3. **OUTER**: Related to JOIN
4. **WHERE**: A filter clause
5. **GROUP BY**: Group the data by one or more attributes (columns)
6. **ROLLUP** | **CUBE**
7. **HAVING**: A filter clause, usually related to the GROUP BY clause, allowing aggregate functions in the filter clause
8. **SELECT**
9. **DISTINCT**: Ensures only distinct values are returned

Usual Execution Order of SELECT Statement

The **usual** order of execution of a SELECT statement:

1. **FROM**: Identifies what table to retrieve the data from
2. **ON**: (used with FROM) Describes how to join tables
3. **OUTER**: Related to JOIN
4. **WHERE**: A filter clause
5. **GROUP BY**: Group the data by one or more attributes (columns)
6. **ROLLUP** | **CUBE**
7. **HAVING**: A filter clause, usually related to the GROUP BY clause, allowing aggregate functions in the filter clause
8. **SELECT**
9. **DISTINCT**: Ensures only distinct values are returned
10. **ORDER BY**: Order the rows by ascending or descending on the tuple of the columns given

Usual Execution Order of SELECT Statement

The **usual** order of execution of a SELECT statement:

1. **FROM**: Identifies what table to retrieve the data from
2. **ON**: (used with FROM) Describes how to join tables
3. **OUTER**: Related to JOIN
4. **WHERE**: A filter clause
5. **GROUP BY**: Group the data by one or more attributes (columns)
6. **ROLLUP** | **CUBE**
7. **HAVING**: A filter clause, usually related to the GROUP BY clause, allowing aggregate functions in the filter clause
8. **SELECT**
9. **DISTINCT**: Ensures only distinct values are returned
10. **ORDER BY**: Order the rows by ascending or descending on the tuple of the columns given
11. **LIMIT** | **TOP**: Display only a specified number of rows

The SELECT Clause

In SELECT clauses, we can specify more than just columns:

Literials Strings, numbers

Expressions Expressions of columns/literals

Functions Built in functions in SQL (ROUND(), etc.)

User Defined Functions Functions that a user can create within SQL to run

```
mysql> SELECT 'str', num, num/4, ROUND(num, 2)
    -> FROM exampletable;
```

The SELECT Clause

In SELECT clauses, we can specify more than just columns:

Literials Strings, numbers

Expressions Expressions of columns/literals

Functions Built in functions in SQL (ROUND(), etc.)

User Defined Functions Functions that a user can create within SQL to run

```
mysql> SELECT 'str', num, num/4, ROUND(num, 2)
    -> FROM exampletable;
```

```
+-----+--------+--------------------+---------------+
| str | num    | num/4              | ROUND(num, 2) |
+-----+--------+--------------------+---------------+
| str |      1 |               0.25 |          1.00 |
| str |   3.14 | 0.7850000262260437 |          3.14 |
| str | 8.2223 |  2.055574893951416 |          8.22 |
+-----+--------+--------------------+---------------+
3 rows in set (0.00 sec)
```

Using DISTINCT within SELECT Clause

In the SELECT clause we can specify to return only distinct tuples of columns

```
SELECT DISTINCT <list of col>
FROM <table/joinedTables/subquery/view>
...;
```

Using DISTINCT within SELECT Clause

In the SELECT clause we can specify to return only distinct tuples of columns

```
SELECT DISTINCT <list of col>
FROM <table/joinedTables/subquery/view>
...;
```

```
mysql> SELECT DISTINCT title
    -> FROM employee;
+--------------------+
| title              |
+--------------------+
| President          |
| Vice President     |
| Treasurer          |
| Operations Manager |
| Loan Manager       |
| Head Teller        |
| Teller             |
+--------------------+
7 rows in set (0.00 sec)
```

Column Alias

To increase the readability of SQL, as well as give better explanation of what aliases are commonly used:

```
mysql> SELECT COUNT(*) AS Club_Size
    -> FROM member;
```

Column Alias

To increase the readability of SQL, as well as give better explanation of what aliases are commonly used:

```
mysql> SELECT COUNT(*) AS Club_Size
    -> FROM member;
```

```
+-----------+
| Club_Size |
+-----------+
|        29 |
+-----------+
1 row in set (0.00 sec)
```

Table Alias

Table aliases are also sued: often for queries using subqueries or multiple tables.

```
SELECT <table1>.<col1>, ..., <table2>.<col1>, ...
FROM <using various tables>
[WHERE <condition>];
```

Table Alias Example

```
mysql> SELECT e.name AS Name, e.position AS Position, p.duties AS Duties
    -> FROM exec AS e JOIN exec_position AS p ON e.position = p.position;
```

Table Alias Example

```
mysql> SELECT e.name AS Name, e.position AS Position, p.duties AS Duties
    -> FROM exec AS e JOIN exec_position AS p ON e.position = p.position;

+-------------------------+----------------+--------------------------------------------
| Name                    | Position       | Duties
+-------------------------+----------------+--------------------------------------------
| Ajanthan Thavaraja (Aj) | Events         | To assist the president and other vice-
| Ajanthan Thavaraja (Aj) | Events         | To chair the organization and promotion
| Darrell Aucoin          | President      | To be aware of MathSocs Policies and By
| Darrell Aucoin          | President      | To call and preside over general meetin
| Darrell Aucoin          | President      | To manage the executive team and the st
| Darrell Aucoin          | President      | To post announcements of all club meeti
| JinCheng Wong           | Events         | To assist the president and other vice-
| JinCheng Wong           | Events         | To chair the organization and promotion
| Massey Cashore          | Events         | To assist the president and other vice-
| Massey Cashore          | Events         | To chair the organization and promotion
| Jacob Burns             | President      | To be aware of MathSocs Policies and By
| Jacob Burns             | President      | To call and preside over general meetin
| Jacob Burns             | President      | To manage the executive team and the st
| Jacob Burns             | President      | To post announcements of all club meeti
| Ming Pan                | Technology     | Maintain and update the club website.
| Ming Pan                | Technology     | Maintain any hardware, software, or tec
| Ming Pan                | Technology     | Perform the duties of a Vice President
| Zixin Nie               | Events         | To assist the president and other vice-
| Zixin Nie               | Events         | To chair the organization and promotion
| Simon wang              | Senior Advisor | Have previous club management experienc
| Simon wang              | Senior Advisor | To be aware of MathSoc's␣Policies␣and␣E
| Alice Wang              | Finance        | To ensure membership fees are collected
| Alice Wang              | Finance        | To keep an up-to-date record of financi
| Alice Wang              | Finance        | To prepare a summary of the financial r
| Alice Wang              | Finance        | To prepare the budget at the beginning
| Alice Wang              | Finance        | To volunteer as president in the absenc
```

Data Definition: INSERT

Definition

INSERT Statement: The INSERT statement inserts a row entry into a table.

```
INSERT INTO <table> (<variableName1>, <variableName2>)
VALUES (<variable1>, <variable2>);
```

INSERT Example

```
mysql> INSERT INTO branch (name, address, city, state, zip)
    -> VALUES ('some branch', 'some address', 'city', 'ST', 90210);
```

INSERT Example

```
mysql> INSERT INTO branch (name, address, city, state, zip)
    -> VALUES ('some branch', 'some address', 'city', 'ST', 90210);

Query OK, 1 row affected (0.00 sec)
```

INSERT Example

```
mysql> INSERT INTO branch (name, address, city, state, zip)
    -> VALUES ('some branch', 'some address', 'city', 'ST', 90210);

Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM branch;
+-----------+---------------+------------------------+---------+-------+-------+
| branch_id | name          | address                | city    | state | zip   |
+-----------+---------------+------------------------+---------+-------+-------+
|         1 | Headquarters  | 3882 Main St.          | Waltham | MA    | 02451 |
|         2 | Woburn Branch | 422 Maple St.          | Woburn  | MA    | 01801 |
|         3 | Quincy Branch | 125 Presidential Way   | Quincy  | MA    | 02169 |
|         4 | So. NH Branch | 378 Maynard Ln.        | Salem   | NH    | 03079 |
|         5 | Headquarters  | 3882 Main St.          | Waltham | MA    | 02451 |
|         6 | Woburn Branch | 422 Maple St.          | Woburn  | MA    | 01801 |
|         7 | Quincy Branch | 125 Presidential Way   | Quincy  | MA    | 02169 |
|         8 | So. NH Branch | 378 Maynard Ln.        | Salem   | NH    | 03079 |
|         9 | some branch   | some address           | city    | ST    | 90210 |
+-----------+---------------+------------------------+---------+-------+-------+
9 rows in set (0.00 sec)
```

Data Definition: UPDATE

Definition

UPDATE Statement: The UPDATE statement changes entries in 0 or more rows.

```
UPDATE <table>
SET <variableName1> = <value1 >, <variableName2> = <
    value2>
WHERE <condition >;
```

UPDATE Example

```
mysql> UPDATE BRANCH
    -> SET address = '123 diff st', name = 'name'
    -> WHERE branch_id = 9;
```

UPDATE Example

```
mysql> UPDATE BRANCH
    -> SET address = '123 diff st', name = 'name'
    -> WHERE branch_id = 9;

Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

UPDATE Example

```
mysql> UPDATE BRANCH
    -> SET address = '123 diff st', name = 'name'
    -> WHERE branch_id = 9;

Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT *
     -> FROM branch;
```

UPDATE Example

```
mysql> UPDATE BRANCH
    -> SET address = '123 diff st', name = 'name'
    -> WHERE branch_id = 9;

Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT *
     -> FROM branch;

+-----------+----------------+------------------------+---------+-------+-------+
| branch_id | name           | address                | city    | state | zip   |
+-----------+----------------+------------------------+---------+-------+-------+
|         1 | Headquarters   | 3882 Main St.          | Waltham | MA    | 02451 |
|         2 | Woburn Branch  | 422 Maple St.          | Woburn  | MA    | 01801 |
|         3 | Quincy Branch  | 125 Presidential Way   | Quincy  | MA    | 02169 |
|         4 | So. NH Branch  | 378 Maynard Ln.        | Salem   | NH    | 03079 |
|         5 | Headquarters   | 3882 Main St.          | Waltham | MA    | 02451 |
|         6 | Woburn Branch  | 422 Maple St.          | Woburn  | MA    | 01801 |
|         7 | Quincy Branch  | 125 Presidential Way   | Quincy  | MA    | 02169 |
|         8 | So. NH Branch  | 378 Maynard Ln.        | Salem   | NH    | 03079 |
|         9 | name           | 123 diff st            | city    | ST    | 90210 |
+-----------+----------------+------------------------+---------+-------+-------+
9 rows in set (0.00 sec)
```

Data Definition: DELETE

Definition

DELETE Statement: The DELETE statement deletes a set of rows from a table.

```
DELETE FROM <table>
[WHERE <condition>];
```

Data Definition: DELETE

Definition

DELETE Statement: The DELETE statement deletes a set of rows from a table.

```
DELETE FROM <table>
[WHERE <condition >];
```

Warning

If no where clause is used, the DELETE statement will delete <u>**all**</u> rows in the table.

DELETE Example

```
mysql> DELETE FROM branch
    -> WHERE address = '123 diff st';
```

DELETE Example

```
mysql> DELETE FROM branch
    -> WHERE address = '123 diff st';

Query OK, 1 row affected (0.01 sec)
```

DELETE Example

```
mysql> DELETE FROM branch
    -> WHERE address = '123 diff st';

Query OK, 1 row affected (0.01 sec)

mysql> SELECT *
    -> FROM branch;
```

DELETE Example

```
mysql> DELETE FROM branch
    -> WHERE address = '123 diff st';

Query OK, 1 row affected (0.01 sec)

mysql> SELECT *
    -> FROM branch;

+-----------+---------------+--------------------------+---------+-------+-------+
| branch_id | name          | address                  | city    | state | zip   |
+-----------+---------------+--------------------------+---------+-------+-------+
|         1 | Headquarters  | 3882 Main St.            | Waltham | MA    | 02451 |
|         2 | Woburn Branch | 422 Maple St.            | Woburn  | MA    | 01801 |
|         3 | Quincy Branch | 125 Presidential Way     | Quincy  | MA    | 02169 |
|         4 | So. NH Branch | 378 Maynard Ln.          | Salem   | NH    | 03079 |
|         5 | Headquarters  | 3882 Main St.            | Waltham | MA    | 02451 |
|         6 | Woburn Branch | 422 Maple St.            | Woburn  | MA    | 01801 |
|         7 | Quincy Branch | 125 Presidential Way     | Quincy  | MA    | 02169 |
|         8 | So. NH Branch | 378 Maynard Ln.          | Salem   | NH    | 03079 |
+-----------+---------------+--------------------------+---------+-------+-------+
8 rows in set (0.00 sec)
```

Data Manipulation: CREATE

Definition

CREATE Statement: The CREATE statement creates a table or view.

```
CREATE TABLE <tableName>
(<variableName1> <variableType1> [<specifications >],
<variableName2> <variableType2> [<specifications >],
...
);
```

```
CREATE VIEW <view_name> [(<column_list>)]
AS <select_statement>
```

CREATE Example

```
mysql> CREATE TABLE student
    -> (stud_num INT UNSIGNED,
    -> fname VARCHAR(30),
    -> lname VARCHAR(30),
    -> CONSTRAINT pk_student PRIMARY KEY (stud_num));
```

CREATE Example

```
mysql> CREATE TABLE student
    -> (stud_num INT UNSIGNED,
    -> fname VARCHAR(30),
    -> lname VARCHAR(30),
    -> CONSTRAINT pk_student PRIMARY KEY (stud_num));
```

```
Query OK, 0 rows affected (0.03 sec)
```

CREATE Example

```
mysql> CREATE TABLE student
    -> (stud_num INT UNSIGNED,
    -> fname VARCHAR(30),
    -> lname VARCHAR(30),
    -> CONSTRAINT pk_student PRIMARY KEY (stud_num));
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> DESC student;
+----------+------------------+------+-----+---------+-------+
| Field    | Type             | Null | Key | Default | Extra |
+----------+------------------+------+-----+---------+-------+
| stud_num | int(10) unsigned | NO   | PRI | 0       |       |
| fname    | varchar(30)      | YES  |     | NULL    |       |
| lname    | varchar(30)      | YES  |     | NULL    |       |
+----------+------------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

Data Manipulation: ALTER

Definition

ALTER Statement: Used to change the characteristics of tables, views, and indices.

```
ALTER TABLE <table_name>
ADD [COLUMN] <col_name> <col_dataType>;

ALTER TABLE <table_name>
MODIFY [COLUMN] <col_name> <col_dataType> <
    modification>;
```

ALTER Example

```
mysql> ALTER TABLE student
    -> ADD major  VARCHAR(20);
```

ALTER Example

```
mysql> ALTER TABLE student
    -> ADD major  VARCHAR(20);
```

```
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

ALTER Example

```
mysql> ALTER TABLE student
    -> ADD major  VARCHAR(20);

Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DESC student;
+----------+------------------+------+-----+---------+-------+
| Field    | Type             | Null | Key | Default | Extra |
+----------+------------------+------+-----+---------+-------+
| stud_num | int(10) unsigned | NO   | PRI | 0       |       |
| fname    | varchar(30)      | YES  |     | NULL    |       |
| lname    | varchar(30)      | YES  |     | NULL    |       |
| major    | varchar(20)      | YES  |     | NULL    |       |
+----------+------------------+------+-----+---------+-------+
4 rows in set (0.01 sec)
```

Definition

DROP Statement: Delete/remove tables, indexes and databases.

```
DROP TABLE <table>;
DROP DATABASE <database>;
```

```
DROP INDEX <table>.<index_name>; /* SQL Server */
DROP INDEX <index_name>; /* DB2/Oracle */
ALTER TABLE <table> DROP INDEX <index>; /* MySQL */
```

Data Manipulation: DROP

Definition

DROP Statement: Delete/remove tables, indexes and databases.

```
DROP TABLE <table>;
DROP DATABASE <database>;
```

```
DROP INDEX <table>.<index_name>; /* SQL Server */
DROP INDEX <index_name>; /* DB2/Oracle */
ALTER TABLE <table> DROP INDEX <index>; /* MySQL */
```

Drop student table

```
mysql> DROP TABLE student;
Query OK, 0 rows affected (0.00 sec)
```

Data Control: GRANT

Definition

GRANT Statement: Grants privileges to users accounts on databases, tables, columns, or routines.

```
GRANT <priv_type> PRIVILEGES ON `<database>`.* TO '<
    user>'@'<host>'
```

GRANT Example

```
mysql> GRANT ALL PRIVILEGES ON bank.* TO 'darrell'@'localhost';
```

GRANT Example

```
mysql> GRANT ALL PRIVILEGES ON bank.* TO 'darrell'@'localhost';


Query OK, 0 rows affected (0.01 sec)
```

GRANT Example

```
mysql> GRANT ALL PRIVILEGES ON bank.* TO 'darrell'@'localhost';

Query OK, 0 rows affected (0.01 sec)

mysql> QUIT;
```

GRANT Example

```
mysql> GRANT ALL PRIVILEGES ON bank.* TO 'darrell'@'localhost';


Query OK, 0 rows affected (0.01 sec)


mysql> QUIT;


Bye
$ mysql -u darrell -p
```

GRANT Example

```
mysql> GRANT ALL PRIVILEGES ON bank.* TO 'darrell'@'localhost';

Query OK, 0 rows affected (0.01 sec)

mysql> QUIT;

Bye
$ mysql -u darrell -p

Enter password:
```

GRANT Example

```
mysql> GRANT ALL PRIVILEGES ON bank.* TO 'darrell'@'localhost';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> QUIT;
```

```
Bye
$ mysql -u darrell -p
```

```
Enter password:
```

```
mysql> SHOW GRANTS;
```

GRANT Example

```
mysql> GRANT ALL PRIVILEGES ON bank.* TO 'darrell'@'localhost';

Query OK, 0 rows affected (0.01 sec)

mysql> QUIT;

Bye
$ mysql -u darrell -p

Enter password:

mysql> SHOW GRANTS;

+------------------------------------------------------------------------------------------
| Grants for darrell@localhost
|
+------------------------------------------------------------------------------------------
| GRANT USAGE ON *.* TO 'darrell'@'localhost' IDENTIFIED BY PASSWORD '*KFAS8538DAB5
| GRANT ALL PRIVILEGES ON `bank`.* TO 'darrell'@'localhost'
+------------------------------------------------------------------------------------------
2 rows in set (0.00 sec)
```

Data Control: REVOKE

Definition

REVOKE Statement: Revokes privileges to users accounts on databases, tables, columns, or routines.

```
REVOKE <priv_type> PRIVILEGES ON `<database>`.* FROM
    '<user>'@'<host>'
```

REVOKE Example

```
mysql> REVOKE ALL PRIVILEGES ON bank.* FROM 'darrell'@'localhost';
```

REVOKE Example

```
mysql> REVOKE ALL PRIVILEGES ON bank.* FROM 'darrell'@'localhost';


Query OK, 0 rows affected (0.00 sec)
```

REVOKE Example

```
mysql> REVOKE ALL PRIVILEGES ON bank.* FROM 'darrell'@'localhost';

Query OK, 0 rows affected (0.00 sec)

mysql> QUIT;
```

REVOKE Example

```
mysql> REVOKE ALL PRIVILEGES ON bank.* FROM 'darrell'@'localhost';

Query OK, 0 rows affected (0.00 sec)

mysql> QUIT;

Bye
$ mysql -u darrell -p
```

REVOKE Example

```
mysql> REVOKE ALL PRIVILEGES ON bank.* FROM 'darrell'@'localhost';

Query OK, 0 rows affected (0.00 sec)

mysql> QUIT;

Bye
$ mysql -u darrell -p

Enter password:
```

REVOKE Example

```
mysql> REVOKE ALL PRIVILEGES ON bank.* FROM 'darrell'@'localhost';

Query OK, 0 rows affected (0.00 sec)

mysql> QUIT;

Bye
$ mysql -u darrell -p

Enter password:

mysql> SHOW GRANTS;
```

REVOKE Example

```
mysql> REVOKE ALL PRIVILEGES ON bank.* FROM 'darrell'@'localhost';

Query OK, 0 rows affected (0.00 sec)

mysql> QUIT;

Bye
$ mysql -u darrell -p

Enter password:

mysql> SHOW GRANTS;

+----------------------------------------------------------------------------
| Grants for darrell@localhost
|
+----------------------------------------------------------------------------
| GRANT USAGE ON *.* TO 'darrell'@'localhost' IDENTIFIED BY PASSWORD '*KFAS8538DAB5
+----------------------------------------------------------------------------
1 row in set (0.00 sec)
```

Built-in Data Types

There 3 main data types

1. Numeric

Built-in Data Types

There 3 main data types

1. Numeric
2. Strings

Built-in Data Types

There 3 main data types

1. Numeric
2. Strings
3. Temporal (Date, Times)

Numeric: Integer

Integer Type	Bytes	Min Value (Signed/Unsigned)	Max Value (Signed/Unsigned)
TINYINT	1	-2^7	$2^7 - 1$
		0	$2^8 - 1$
SMALLINT	2	-2^{15}	$2^{15} - 1$
		0	$2^{16} - 1$
MEDIUMINT	3	-2^{23}	$2^{23} - 1$
		0	$2^{24} - 1$
INT, **INTEGER**	4	-2^{31}	$2^{31} - 1$
		0	$2^{31} - 1$
BIGINT	8	-2^{39}	$2^{39} - 1$
		0	$2^{39} - 1$

Numeric: Fixed-Point

Fixed-Point data types have two components usually supplied by the user (Precision and Scale).

```
<variableName> DECIMAL(<integer-Precision>,
<integer-Scale>)
```

Numeric: Fixed-Point

Fixed-Point data types have two components usually supplied by the user (Precision and Scale).

```
<variableName> DECIMAL(<integer-Precision>,
<integer-Scale>)
```

Precision: How many significant digits there are (before and after the decimal)

Numeric: Fixed-Point

Fixed-Point data types have two components usually supplied by the user (Precision and Scale).

```
<variableName> DECIMAL(<integer-Precision>,
<integer-Scale>)
```

Precision: How many significant digits there are (before and after the decimal)

Scale: How many digits are after the decimal point

Numeric: Fixed-Point

Fixed-Point data types have two components usually supplied by the user (Precision and Scale).

```
<variableName> DECIMAL(<integer-Precision>,
<integer-Scale>)
```

Precision: How many significant digits there are (before and after the decimal)

Scale: How many digits are after the decimal point

- The max value for *Precision* is 65, with a default value of 10

Numeric: Fixed-Point

Fixed-Point data types have two components usually supplied by the user (Precision and Scale).

```
<variableName> DECIMAL(<integer-Precision>,
<integer-Scale>)
```

Precision: How many significant digits there are (before and after the decimal)

Scale: How many digits are after the decimal point

- The max value for *Precision* is 65, with a default value of 10
- *Scale's* min value is 0, also the default value

Numeric: Fixed-Point

Fixed-Point data types have two components usually supplied by the user (Precision and Scale).

```
<variableName> DECIMAL(<integer-Precision>,
<integer-Scale>)
```

Precision: How many significant digits there are (before and after the decimal)

Scale: How many digits are after the decimal point

- The max value for *Precision* is 65, with a default value of 10
- *Scale's* min value is 0, also the default value

Example

For a variable column to take on values from -999.99 to 999.99:

```
<variableName> DECIMAL(5, 2)
```

Numeric: Fixed-Point

DECIMAL, DEC, FIXED, NUMERIC are alases for this data type

Numeric: Fixed-Point

DECIMAL, DEC, FIXED, NUMERIC are alases for this data type

- **DECIMAL** is equivalent to **DECIMAL(10), DECIMAL(10,0)**

Numeric: Fixed-Point

DECIMAL, DEC, FIXED, NUMERIC are alases for this data type

- **DECIMAL** is equivalent to **DECIMAL(10), DECIMAL(10,0)**
- This is the best data type for money and other numerical values needing exact precision

Strings

CHAR(<integer>) /* a string of fixed length */

Strings

CHAR(<integer>) /* a string of fixed length */

- Has a max length of 255 bytes (usually 1 byte == character, depending on the char type)

Strings

CHAR(<integer>) /* a string of fixed length */

- Has a max length of 255 bytes (usually 1 byte == character, depending on the char type)

VARCHAR(<integer>) /* a string of variable length with set max length */

Strings

CHAR(<integer>) /* a string of fixed length */

- Has a max length of 255 bytes (usually 1 byte == character, depending on the char type)

VARCHAR(<integer>) /* a string of variable length with set max length */

- Has a max length of 65,535 bytes (usually 1 byte == character, depending on the char type)

Text Data Used for when you might need to exceed the 64 KB limit for VARCHAR (specific for MySQL, other SQL servers use similar but not the same types)

Text Type	Max Bytes
TINYTEXT	$2^8 - 1 =$255
TEXT	$2^{16} - 1 =$65,535
MEDIUMTEXT	$2^{24} - 1 =$16,777,215
LONGTEXT	$2^{32} - 1 =$4,294,967,295

Strings

Text Data Used for when you might need to exceed the 64 KB limit for VARCHAR (specific for MySQL, other SQL servers use similar but not the same types)

Text Type	Max Bytes
TINYTEXT	$2^8 - 1 =$255
TEXT	$2^{16} - 1 =$65,535
MEDIUMTEXT	$2^{24} - 1 =$16,777,215
LONGTEXT	$2^{32} - 1 =$4,294,967,295

- Any text entered that is larger than the max size, the data will be truncated

Strings

Text Data Used for when you might need to exceed the 64 KB limit for VARCHAR (specific for MySQL, other SQL servers use similar but not the same types)

Text Type	Max Bytes
TINYTEXT	$2^8 - 1 =$255
TEXT	$2^{16} - 1 =$65,535
MEDIUMTEXT	$2^{24} - 1 =$16,777,215
LONGTEXT	$2^{32} - 1 =$4,294,967,295

- Any text entered that is larger than the max size, the data will be truncated
- When being sorted, only the first 1,024 bytes are used (MySQL)

Temporal Data

Data Type	Default format	Min Range	Max Range
DATE	'YYYY-MM-DD'	'1000-01-01'	'9999-12-31'
DATETIME	'YYYY-MM-DD HH:MM:SS'	'1000-01-01 00:00:00'	'9999-12-31 23:59:59'
TIMESTAMP	'YYYY-MM-DD HH:MM:SS'	'1970-01-01 00:00:01'	'2038-01-19 03:14:07'
YEAR	'YYYY'	'1901'	'2155'
TIME	'HHH:MM:SS'	'-838:59:59'	'838:59:59'

- Date parts MUST ALWAYS be given in year-month-day order

Temporal Data

Data Type	Default format	Min Range	Max Range
DATE	'YYYY-MM-DD'	'1000-01-01'	'9999-12-31'
DATETIME	'YYYY-MM-DD HH:MM:SS'	'1000-01-01 00:00:00'	'9999-12-31 23:59:59'
TIMESTAMP	'YYYY-MM-DD HH:MM:SS'	'1970-01-01 00:00:01'	'2038-01-19 03:14:07'
YEAR	'YYYY'	'1901'	'2155'
TIME	'HHH:MM:SS'	'-838:59:59'	'838:59:59'

- Date parts MUST ALWAYS be given in year-month-day order
- **DATETIME** and **TIMESTAMP** can include fractional seconds part in up to microseconds (6 digits) precision.

Temporal Data

Data Type	Default format	Min Range	Max Range
DATE	'YYYY-MM-DD'	'1000-01-01'	'9999-12-31'
DATETIME	'YYYY-MM-DD HH:MM:SS'	'1000-01-01 00:00:00'	'9999-12-31 23:59:59'
TIMESTAMP	'YYYY-MM-DD HH:MM:SS'	'1970-01-01 00:00:01'	'2038-01-19 03:14:07'
YEAR	'YYYY'	'1901'	'2155'
TIME	'HHH:MM:SS'	'-838:59:59'	'838:59:59'

- Date parts MUST ALWAYS be given in year-month-day order
- **DATETIME** and **TIMESTAMP** can include fractional seconds part in up to microseconds (6 digits) precision.

  DATETIME/TIMESTAMP Using 'YYYY-MM-DD HH:MM:SS[.fraction]' with range '1000-01-01 00:00:00.000000' to '9999-12-31 23:59:59.999999'

Temporal Data

Data Type	Default format	Min Range	Max Range
DATE	'YYYY-MM-DD'	'1000-01-01'	'9999-12-31'
DATETIME	'YYYY-MM-DD HH:MM:SS'	'1000-01-01 00:00:00'	'9999-12-31 23:59:59'
TIMESTAMP	'YYYY-MM-DD HH:MM:SS'	'1970-01-01 00:00:01'	'2038-01-19 03:14:07'
YEAR	'YYYY'	'1901'	'2155'
TIME	'HHH:MM:SS'	'-838:59:59'	'838:59:59'

- Date parts MUST ALWAYS be given in year-month-day order
- **DATETIME** and **TIMESTAMP** can include fractional seconds part in up to microseconds (6 digits) precision.

  DATETIME/TIMESTAMP Using 'YYYY-MM-DD HH:MM:SS[.fraction]' with range '1000-01-01 00:00:00.000000' to '9999-12-31 23:59:59.999999'

- **DATETIME** and **TIMESTAMP** data types can offer automatic initialization to the current date and time

Temporal Data

- SQL converts **TIMESTAMP** to UTC for storage, meaning it gives back different values depending on the current time zone, it's being accessed from.

Temporal Data

- SQL converts **TIMESTAMP** to UTC for storage, meaning it gives back different values depending on the current time zone, it's being accessed from.
  - This is useful as data timestamped in one location does not need to be changed when doing data analysis in another time zone.

Null Values

Numeric, strings and temporal data types can have NULL values.
A field can never be equal to NULL, but should be considered unknown or missing data.
For instance, if either x or y is NULL:

- $x + y \implies$ NULL

Null Values

Numeric, strings and temporal data types can have NULL values.
A field can never be equal to NULL, but should be considered unknown or missing data.
For instance, if either x or y is NULL:

- $x + y \implies$ NULL
- $x > y \implies$ NULL

Null Values

Numeric, strings and temporal data types can have NULL values.
A field can never be equal to NULL, but should be considered unknown or missing data.
For instance, if either x or y is NULL:

- $x + y \implies$ NULL
- $x > y \implies$ NULL

Null Values

Numeric, strings and temporal data types can have NULL values.
A field can never be equal to NULL, but should be considered unknown or missing data.
For instance, if either x or y is NULL:

- $x + y \implies$ NULL
- $x > y \implies$ NULL

To test for NULL values use

```
<attribute> IS NULL        <attribute> IS NOT NULL
```

Null Values

SQL uses a three-value logic system: TRUE, FALSE, NULL:

Null Values

SQL uses a three-value logic system: TRUE, FALSE, NULL:

$\wedge$	TRUE	FALSE	NULL
TRUE	T	F	NULL
FALSE	F	F	F
NULL	NULL	NULL	NULL

Null Values

SQL uses a three-value logic system: TRUE, FALSE, NULL:

$\wedge$	TRUE	FALSE	NULL
TRUE	T	F	NULL
FALSE	F	F	F
NULL	NULL	NULL	NULL

$\vee$	TRUE	FALSE	NULL
TRUE	T	T	T
FALSE	T	F	NULL
NULL	T	NULL	NULL

Null Values

SQL uses a three-value logic system: TRUE, FALSE, NULL:

$\wedge$	TRUE	FALSE	NULL
TRUE	T	F	NULL
FALSE	F	F	F
NULL	NULL	NULL	NULL

$\vee$	TRUE	FALSE	NULL
TRUE	T	T	T
FALSE	T	F	NULL
NULL	T	NULL	NULL

NOT	TRUE	FALSE	NULL
	F	T	NULL

Constraints

Constraints limit what can be entered into fields in a table and help ensure encapsulation:

PRIMARY KEY constraints Uniquely identifies each record in a table (quickly referencing it)

Constraints

Constraints limit what can be entered into fields in a table and help ensure encapsulation:

PRIMARY KEY constraints Uniquely identifies each record in a table (quickly referencing it)

FOREIGN KEY constraints Points to a PRIMARY KEY of another table, enabling them to easily join them

Constraints

Constraints limit what can be entered into fields in a table and help ensure encapsulation:

PRIMARY KEY constraints Uniquely identifies each record in a table (quickly referencing it)

FOREIGN KEY constraints Points to a PRIMARY KEY of another table, enabling them to easily join them

CHECK constraints Limits the range of values that a field can take.

Constraints

Constraints limit what can be entered into fields in a table and help ensure encapsulation:

PRIMARY KEY constraints
: Uniquely identifies each record in a table (quickly referencing it)

FOREIGN KEY constraints
: Points to a PRIMARY KEY of another table, enabling them to easily join them

CHECK constraints
: Limits the range of values that a field can take.

UNIQUE constraints
: Enforces uniqueness on an field (column).

Constraints

Constraints limit what can be entered into fields in a table and help ensure encapsulation:

PRIMARY KEY constraints Uniquely identifies each record in a table (quickly referencing it)

FOREIGN KEY constraints Points to a PRIMARY KEY of another table, enabling them to easily join them

CHECK constraints Limits the range of values that a field can take.

UNIQUE constraints Enforces uniqueness on an field (column).

NOT NULL constraints Enforces a field to always contain a value.

Constraints

Constraints limit what can be entered into fields in a table and help ensure encapsulation:

PRIMARY KEY constraints Uniquely identifies each record in a table (quickly referencing it)

FOREIGN KEY constraints Points to a PRIMARY KEY of another table, enabling them to easily join them

CHECK constraints Limits the range of values that a field can take.

UNIQUE constraints Enforces uniqueness on an field (column).

NOT NULL constraints Enforces a field to always contain a value.

DROP constraints Drops a constraint from a table.

Constraints

Constraints limit what can be entered into fields in a table and help ensure encapsulation:

PRIMARY KEY constraints Uniquely identifies each record in a table (quickly referencing it)

FOREIGN KEY constraints Points to a PRIMARY KEY of another table, enabling them to easily join them

CHECK constraints Limits the range of values that a field can take.

UNIQUE constraints Enforces uniqueness on an field (column).

NOT NULL constraints Enforces a field to always contain a value.

DROP constraints Drops a constraint from a table.

Constraints

Constraints limit what can be entered into fields in a table and help ensure encapsulation:

PRIMARY KEY constraints Uniquely identifies each record in a table (quickly referencing it)

FOREIGN KEY constraints Points to a PRIMARY KEY of another table, enabling them to easily join them

CHECK constraints Limits the range of values that a field can take.

UNIQUE constraints Enforces uniqueness on an field (column).

NOT NULL constraints Enforces a field to always contain a value.

DROP constraints Drops a constraint from a table.

We can also create indexes for fields making them easily searchable

Filter Clauses

There are 2 filter clauses in SQL: WHERE and HAVING.
WHERE is performed before aggregation (GROUP BY) and HAVING is performed after, allowing aggregate functions such as COUNT(*), to be performed.

Note: Unless specified there is an implicit GROUP BY statement for the whole row set.

WHERE Clause

WHERE clauses filters the result set, removing rows the condition returns either FALSE or NULL.

```
mysql> SELECT *
    -> FROM individual
    -> WHERE lname > 'j';
```

WHERE Clause

WHERE clauses filters the result set, removing rows the condition returns either FALSE or NULL.

```
mysql> SELECT *
    -> FROM individual
    -> WHERE lname > 'j';
```

```
+---------+----------+---------+------------+
| cust_id | fname    | lname   | birth_date |
+---------+----------+---------+------------+
|       2 | Susan    | Tingley | 1968-08-15 |
|       3 | Frank    | Tucker  | 1958-02-06 |
|       6 | John     | Spencer | 1962-09-14 |
|       7 | Margaret | Young   | 1947-03-19 |
+---------+----------+---------+------------+
4 rows in set (0.00 sec)
```

HAVING Clause

HAVING clauses are very similar to WHERE clauses but can have aggregate function in their conditions.

You can have a WHERE and HAVING clause in the same statement.

```
mysql> SELECT title, COUNT(*)
    -> FROM employee
    -> GROUP BY title
    -> HAVING COUNT(*) > 3;
```

HAVING Clause

HAVING clauses are very similar to WHERE clauses but can have aggregate function in their conditions.
You can have a WHERE and HAVING clause in the same statement.

```
mysql> SELECT title, COUNT(*)
    -> FROM employee
    -> GROUP BY title
    -> HAVING COUNT(*) > 3;
```

```
+-------------+----------+
| title       | COUNT(*) |
+-------------+----------+
| Head Teller |        4 |
| Teller      |        9 |
+-------------+----------+
2 rows in set (0.01 sec)
```

Predicate Operators

Operator	Description	Example
=	Equal to	WHERE gender = 'M'
<>, !=	Not equal to	WHERE gender <> 'M'
>	Greater than	WHERE num > 5
<	Less than	WHERE num < 5
>=	Greater than or equal to	WHERE num >= 5
<=	Greater than or equal to	WHERE num <= 5
IS NULL	Value is NULL	WHERE num IS NULL
IS NOT NULL	Value is not NULL	WHERE num IS NOT NULL
BETWEEN	Between an inclusive range	WHERE num BETWEEN 3 and 5
IN	Value in a list of values	WHERE num IN (3, 5, 8)
LIKE	Search for a pattern	WHERE str LIKE 'F%'
EXISTS	Does subquery have any rows	WHERE EXISTS (<subquery>)
REGEXP, RLIKE	(MySQL) Search for a regular expression pattern	WHERE str RLIKE '^[FG]'

WHERE Clause Example

WHERE clauses filters the result set, removing rows the condition returns either FALSE or NULL.

```
mysql> SELECT fname, lname, title
    -> FROM employee
    -> WHERE title in ('Teller', 'Head Teller');
```

WHERE Clause Example

WHERE clauses filters the result set, removing rows the condition returns either FALSE or NULL.

```
mysql> SELECT fname, lname, title
    -> FROM employee
    -> WHERE title in ('Teller', 'Head Teller');

+----------+----------+-------------+
| fname    | lname    | title       |
+----------+----------+-------------+
| Helen    | Fleming  | Head Teller |
| Chris    | Tucker   | Teller      |
| Sarah    | Parker   | Teller      |
| Jane     | Grossman | Teller      |
| Paula    | Roberts  | Head Teller |
| Thomas   | Ziegler  | Teller      |
| Samantha | Jameson  | Teller      |
| John     | Blake    | Head Teller |
| Cindy    | Mason    | Teller      |
| Frank    | Portman  | Teller      |
| Theresa  | Markham  | Head Teller |
| Beth     | Fowler   | Teller      |
| Rick     | Tulman   | Teller      |
+----------+----------+-------------+
13 rows in set (0.00 sec)
```

Operators Modifiers

The operators =, <>, !=, >, <, >=, <= can be used with a list of values and the operators ALL or ANY/SOME.

The operators =, <>, !=, >, <, >=, <= can be used with a list of values and the operators ALL or ANY/SOME.

ANY, SOME Operator returns true, if operator is true for **any** value (E_i) in the set.

$$E_1 \vee E_2 \vee E_3 \vee \cdots \vee E_n$$

The operators =, <>, !=, >, <, >=, <= can be used with a list of values and the operators ALL or ANY/SOME.

ANY, SOME Operator returns true, if operator is true for **any** value (E_i) in the set.

$$E_1 \vee E_2 \vee E_3 \vee \cdots \vee E_n$$

ALL Operator returns true, if operator is true for **all** values (E_i) in the set.

$$E_1 \wedge E_2 \wedge E_3 \wedge \cdots \wedge E_n$$

AND, OR Operators

A group of filter conditions can be linked together with AND or OR operators.

WHERE (<condition1> AND <condition2>) OR <condition3>

AND, OR Operators Example

Example

All tellers starting from 2003

```
mysql> SELECT fname, lname, title, start_date
    -> FROM employee
    -> WHERE title in ('Teller', 'Head Teller')
    -> AND YEAR(start_date) >= 2003;
```

AND, OR Operators Example

Example

All tellers starting from 2003

```
mysql> SELECT fname, lname, title, start_date
    -> FROM employee
    -> WHERE title in ('Teller', 'Head Teller')
    -> AND YEAR(start_date) >= 2003;

+----------+---------+-------------+------------+
| fname    | lname   | title       | start_date |
+----------+---------+-------------+------------+
| Helen    | Fleming | Head Teller | 2004-03-17 |
| Chris    | Tucker  | Teller      | 2004-09-15 |
| Samantha | Jameson | Teller      | 2003-01-08 |
| Frank    | Portman | Teller      | 2003-04-01 |
+----------+---------+-------------+------------+
4 rows in set (0.00 sec)
```

GROUP BY Clause

The GROUP BY Clause groups the result set by distinct entries in the columns specified.

```
SELECT <column tuple>, <aggregate function>
FROM <table/join/view/subquery>
[WHERE <condition>]
GROUP BY <column tuple>
[HAVING <condition>]
```

GROUP BY Clause

The GROUP BY Clause groups the result set by distinct entries in the columns specified.

```
SELECT <column tuple>, <aggregate function>
FROM <table/join/view/subquery>
[WHERE <condition>]
GROUP BY <column tuple>
[HAVING <condition>]
```

- Only columns specified in the GROUP BY (and expressions of aggregate functions) can appear in the SELECT clause.

GROUP BY Clause

The GROUP BY Clause groups the result set by distinct entries in the columns specified.

```
SELECT <column tuple>, <aggregate function>
FROM <table/join/view/subquery>
[WHERE <condition>]
GROUP BY <column tuple>
[HAVING <condition>]
```

- Only columns specified in the GROUP BY (and expressions of aggregate functions) can appear in the SELECT clause.
- There is an implicit GROUP BY statement allowing aggregate functions to be in the SELECT clause without a GROUP BY clause

GROUP BY Clause

The GROUP BY Clause groups the result set by distinct entries in the columns specified.

```
SELECT <column tuple>, <aggregate function>
FROM <table/join/view/subquery>
[WHERE <condition>]
GROUP BY <column tuple>
[HAVING <condition>]
```

- Only columns specified in the GROUP BY (and expressions of aggregate functions) can appear in the SELECT clause.
- There is an implicit GROUP BY statement allowing aggregate functions to be in the SELECT clause without a GROUP BY clause

GROUP BY Clause

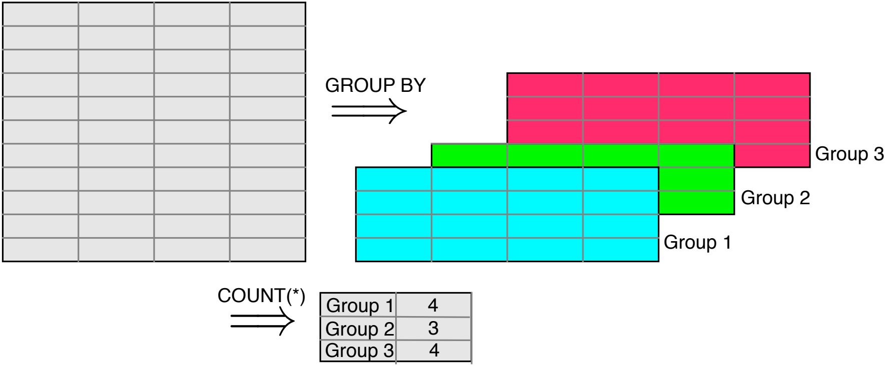
The GROUP BY Clause groups the result set by distinct entries in the columns specified.

```
SELECT <column tuple>, <aggregate function>
FROM <table/join/view/subquery>
[WHERE <condition>]
GROUP BY <column tuple>
[HAVING <condition>]
```

- Only columns specified in the GROUP BY (and expressions of aggregate functions) can appear in the SELECT clause.
- There is an implicit GROUP BY statement allowing aggregate functions to be in the SELECT clause without a GROUP BY clause

```
mysql> SELECT 'Total', SUM(avail_balance) FROM account;
+-------+--------------------+
| Total | SUM(avail_balance) |
+-------+--------------------+
| Total |          170754.46 |
+-------+--------------------+
1 row in set (0.00 sec)
```

GROUP BY Clause



GROUP BY Example

Example

Find the number of customers by type of customer and state

```
mysql> SELECT cust_type_cd, state, COUNT(*)
    -> FROM customer
    -> GROUP BY cust_type_cd,  state;
```

GROUP BY Example

Example

Find the number of customers by type of customer and state

```
mysql> SELECT cust_type_cd, state, COUNT(*)
    -> FROM customer
    -> GROUP BY cust_type_cd, state;

+--------------+-------+----------+
| cust_type_cd | state | COUNT(*) |
+--------------+-------+----------+
| I            | MA    |        7 |
| I            | NH    |        2 |
| B            | MA    |        2 |
| B            | NH    |        2 |
+--------------+-------+----------+
4 rows in set (0.00 sec)
```

GROUP BY Example

Example

Find the number of accounts for each customer and total available balance

```
mysql> SELECT cust_id, COUNT(*) AS Num_account,
    -> SUM(avail_balance) AS cust_avail_balance
    -> FROM account
    -> GROUP BY cust_id;
```

GROUP BY Example

Example

Find the number of accounts for each customer and total available balance

```
mysql> SELECT cust_id, COUNT(*) AS Num_account,
    -> SUM(avail_balance) AS cust_avail_balance
    -> FROM account
    -> GROUP BY cust_id;

+---------+-------------+--------------------+
| cust_id | Num_account | cust_avail_balance |
+---------+-------------+--------------------+
|       1 |           3 |            4557.75 |
|       2 |           2 |            2458.02 |
|       3 |           2 |            3270.25 |
|       4 |           3 |            6788.98 |
|       5 |           1 |            2237.97 |
|       6 |           2 |           10122.37 |
|       7 |           1 |            5000.00 |
|       8 |           2 |            3875.18 |
|       9 |           3 |           10971.22 |
|      10 |           2 |           23575.12 |
|      11 |           1 |            9345.55 |
|      12 |           1 |           38552.05 |
|      13 |           1 |           50000.00 |
```

Aggregate Functions

Function	Return value
AVG(<numeric col>)	Average of non-null values
COUNT(<col or *>)	Count of non-null values
MAX(<col>)	Maximum value of column
MIN(<col>)	Minimum value of column
SUM(<numeric col>)	Sum of column
STD(<numeric col>), STDDEV_POP(<numeric col>), STDDEV(<numeric col>)	Population standard deviation
STDDEV_SAMP(<numeric col>)	Sample standard deviation
VAR_POP(<numeric col>), VARIANCE(<numeric col>)	Population variance
VAR_SAMP(<numeric col>)	Sample variance estimate
GROUP_CONCAT(<string col>)	A concatenated string

Aggregate Functions Example

```
mysql> SELECT position, GROUP_CONCAT(duties)
    -> from exec_position
    -> GROUP BY position;
```

Aggregate Functions Example

```
mysql> SELECT position, GROUP_CONCAT(duties)
    -> from exec_position
    -> GROUP BY position;

+-----------------+----------------------------------------------
| Events          | To assist the president and other vice-presidents
| Finance         | To ensure membership fees are collected and mainta
| President       | To be aware of MathSocs Policies and Bylaws in rega
| Senior Advisor  | Have previous club management experience in order t
| Technology      | Maintain and update the club website.,Maintain any
+-----------------+----------------------------------------------
5 rows in set (0.01 sec)
```

Aggregate Functions with DISTINCT

We can specify that aggregate functions work on only distinct values in the set:

```
<aggregate fuction>(DISTINCT col)
```

Example

Find the number of distinct titles in employee

Aggregate Functions with DISTINCT

We can specify that aggregate functions work on only distinct values in the set:

```
<aggregate fuction>(DISTINCT col)
```

Example

Find the number of distinct titles in employee

```
mysql> SELECT COUNT(DISTINCT title)
    -> FROM employee;
+-----------------------+
| COUNT(DISTINCT title) |
+-----------------------+
|                     7 |
+-----------------------+
1 row in set (0.00 sec)
```

GROUP BY with ROLLUP

We can use aggregate functions with different levels of the GROUP BY clause by using WITH ROLLUP.

```
mysql> SELECT title, COUNT(*)
    -> FROM employee
    -> GROUP BY title WITH ROLLUP;
```

GROUP BY with ROLLUP

We can use aggregate functions with different levels of the GROUP BY clause by using WITH ROLLUP.

```
mysql> SELECT title, COUNT(*)
    -> FROM employee
    -> GROUP BY title WITH ROLLUP;

+--------------------+----------+
| title              | COUNT(*) |
+--------------------+----------+
| Head Teller        |        4 |
| Loan Manager       |        1 |
| Operations Manager |        1 |
| President          |        1 |
| Teller             |        9 |
| Treasurer          |        1 |
| Vice President     |        1 |
| NULL               |       18 |
+--------------------+----------+
8 rows in set (0.00 sec)
```

Example 2: GROUP BY with ROLLUP

```
mysql> SELECT title, YEAR(start_date), COUNT(*)
    -> FROM employee
    -> GROUP BY title, YEAR(start_date) WITH ROLLUP;
```

Example 2: GROUP BY with ROLLUP

```
mysql> SELECT title, YEAR(start_date), COUNT(*)
    -> FROM employee
    -> GROUP BY title, YEAR(start_date) WITH ROLLUP;
```

title	YEAR(start_date)	COUNT(*)
Head Teller	2000	1
Head Teller	2001	1
Head Teller	2002	1
Head Teller	2004	1
Head Teller	NULL	4
Loan Manager	2003	1
Loan Manager	NULL	1
Operations Manager	2002	1
Operations Manager	NULL	1
President	2001	1
President	NULL	1
Teller	2000	1
Teller	2002	5
Teller	2003	2
Teller	2004	1
Teller	NULL	9
Treasurer	2000	1
Treasurer	NULL	1
Vice President	2002	1
Vice President	NULL	1
NULL	NULL	18

21 rows in set (0.00 sec)

Joins

At times, we need information from multiple tables, to do this we need to join tables together. We can do this several ways:

At times, we need information from multiple tables, to do this we need to join tables together. We can do this several ways:

1. **CROSS JOIN**: The cartesian product of rows from each table.

Joins

At times, we need information from multiple tables, to do this we need to join tables together. We can do this several ways:

1. **CROSS JOIN**: The cartesian product of rows from each table.
2. **INNER JOIN**: Join two tables on a join-predicate, losing rows when evaluated false/null.

Joins

At times, we need information from multiple tables, to do this we need to join tables together. We can do this several ways:

1. **CROSS JOIN**: The cartesian product of rows from each table.
2. **INNER JOIN**: Join two tables on a join-predicate, losing rows when evaluated false/null.
3. **OUTER JOIN**: Retains each record for the table(s) even when it has no matching rows from the other table. The returning table has null values for missing records.

Joins

At times, we need information from multiple tables, to do this we need to join tables together. We can do this several ways:

1. **CROSS JOIN**: The cartesian product of rows from each table.
2. **INNER JOIN**: Join two tables on a join-predicate, losing rows when evaluated false/null.
3. **OUTER JOIN**: Retains each record for the table(s) even when it has no matching rows from the other table. The returning table has null values for missing records.
    1. **LEFT OUTER JOIN**: Keep each record for first table but not the table it's joining with.

Joins

At times, we need information from multiple tables, to do this we need to join tables together. We can do this several ways:

1. **CROSS JOIN**: The cartesian product of rows from each table.
2. **INNER JOIN**: Join two tables on a join-predicate, losing rows when evaluated false/null.
3. **OUTER JOIN**: Retains each record for the table(s) even when it has no matching rows from the other table. The returning table has null values for missing records.
    1. **LEFT OUTER JOIN**: Keep each record for first table but not the table it's joining with.
    2. **RIGHT OUTER JOIN**: Keep each record for second table but not the table it's joining with.

Joins

At times, we need information from multiple tables, to do this we need to join tables together. We can do this several ways:

1. **CROSS JOIN**: The cartesian product of rows from each table.
2. **INNER JOIN**: Join two tables on a join-predicate, losing rows when evaluated false/null.
3. **OUTER JOIN**: Retains each record for the table(s) even when it has no matching rows from the other table. The returning table has null values for missing records.
    1. **LEFT OUTER JOIN**: Keep each record for first table but not the table it's joining with.
    2. **RIGHT OUTER JOIN**: Keep each record for second table but not the table it's joining with.
    3. **FULL OUTER JOIN**: Keep all record for all tables.

Joins

At times, we need information from multiple tables, to do this we need to join tables together. We can do this several ways:

1. **CROSS JOIN**: The cartesian product of rows from each table.
2. **INNER JOIN**: Join two tables on a join-predicate, losing rows when evaluated false/null.
3. **OUTER JOIN**: Retains each record for the table(s) even when it has no matching rows from the other table. The returning table has null values for missing records.
    1. **LEFT OUTER JOIN**: Keep each record for first table but not the table it's joining with.
    2. **RIGHT OUTER JOIN**: Keep each record for second table but not the table it's joining with.
    3. **FULL OUTER JOIN**: Keep all record for all tables.
4. **NATURAL JOIN**: Tables with the exact same column name and datatype are joined along that column.

CROSS JOIN

CROSS JOIN is the cartesian product of two tables

```
SELECT <columns>
FROM <table1> CROSS JOIN <table2>
...
```

CROSS JOIN

Two tables t1, t2 with values 1, 2 for t1 and 'one', 'two', 'three' for t2.

CROSS JOIN

CROSS JOIN is the cartesian product of two tables

```
SELECT <columns>
FROM <table1> CROSS JOIN <table2>
...
```

CROSS JOIN

Two tables t1, t2 with values 1, 2 for t1 and 'one', 'two', 'three' for t2.

```
mysql> SELECT t1.num, t2.num
    -> FROM t1 CROSS JOIN t2;
```

CROSS JOIN

CROSS JOIN is the cartesian product of two tables

```
SELECT <columns>
FROM <table1> CROSS JOIN <table2>
...
```

CROSS JOIN

Two tables t1, t2 with values 1, 2 for t1 and 'one', 'two', 'three' for t2.

```
mysql> SELECT t1.num, t2.num
    -> FROM t1 CROSS JOIN t2;

+-------+-------+
| num   | num   |
+-------+-------+
|     1 | one   |
|     2 | one   |
|     1 | two   |
|     2 | two   |
|     1 | three |
|     2 | three |
+-------+-------+
6 rows in set (0.00 sec)
```

INNER JOIN

INNER JOIN Join two tables where the join condition returns true. Discard when returning false or NULL.

```
SELECT <columns>
FROM <table1> INNER JOIN <table2>
ON <join condition>
...
```

INNER JOIN

```
mysql> SELECT e.name AS Name, e.position , p.duties
    -> FROM exec AS e JOIN exec_position AS p
    -> ON e.position = p.position;
```

INNER JOIN

```
mysql> SELECT e.name AS Name, e.position, p.duties
    -> FROM exec AS e JOIN exec_position AS p
    -> ON e.position = p.position;
```

Name	position	duties
Ajanthan Thavaraja (Aj)	Events	To assist the president and other vice-
Ajanthan Thavaraja (Aj)	Events	To chair the organization and promotion
Darrell Aucoin	President	To be aware of MathSocs Policies and By
Darrell Aucoin	President	To call and preside over general meetin
Darrell Aucoin	President	To manage the executive team and the st
Darrell Aucoin	President	To post announcements of all club meeti
JinCheng Wong	Events	To assist the president and other vice-
JinCheng Wong	Events	To chair the organization and promotion
Massey Cashore	Events	To assist the president and other vice-
Massey Cashore	Events	To chair the organization and promotion
Jacob Burns	President	To be aware of MathSocs Policies and By
Jacob Burns	President	To call and preside over general meetin
Jacob Burns	President	To manage the executive team and the st
Jacob Burns	President	To post announcements of all club meeti
Ming Pan	Technology	Maintain and update the club website.
Ming Pan	Technology	Maintain any hardware, software, or tec
Ming Pan	Technology	Perform the duties of a Vice President
Zixin Nie	Events	To assist the president and other vice-
Zixin Nie	Events	To chair the organization and promotion
Simon wang	Senior Advisor	Have previous club management experienc
Simon wang	Senior Advisor	To be aware of MathSoc's␣Policies␣and␣B
Alice Wang	Finance	To ensure membership fees are collected
Alice Wang	Finance	To keep an up-to-date record of financi
Alice Wang	Finance	To prepare a summary of the financial r
Alice Wang	Finance	To prepare the budget at the beginning
Alice Wang	Finance	To volunteer as president in the absenc

ON Clause

The ON clause specifies the join condition:

- The ON clause can use a multiple set of conditions connected by AND, OR

ON Clause

The ON clause specifies the join condition:

- The ON clause can use a multiple set of conditions connected by AND, OR
- USING(<join col>) can also be used if both tables have the same column name and type

ON Clause

The ON clause specifies the join condition:

- The ON clause can use a multiple set of conditions connected by AND, OR
- USING(<join col>) can also be used if both tables have the same column name and type
- Some SQL implementations constructs the ON clause from the WHERE clause (DB2)

INNER JOIN

```
mysql> SELECT e.name AS Name, e.position, p.duties
    -> FROM exec AS e JOIN exec_position AS p
    ->   USING(position);
```

INNER JOIN

```
mysql> SELECT e.name AS Name, e.position , p.duties
    -> FROM exec AS e JOIN exec_position AS p
    ->   USING(position);

+---------------------------+------------------+----------------------------------------------
| Name                      | position         | duties
+---------------------------+------------------+----------------------------------------------
| Ajanthan Thavaraja (Aj)   | Events           | To assist the president and other vice-
| Ajanthan Thavaraja (Aj)   | Events           | To chair the organization and promotion
| Darrell Aucoin            | President        | To be aware of MathSocs Policies and By
| Darrell Aucoin            | President        | To call and preside over general meetin
| Darrell Aucoin            | President        | To manage the executive team and the st
| Darrell Aucoin            | President        | To post announcements of all club meeti
| JinCheng Wong             | Events           | To assist the president and other vice-
| JinCheng Wong             | Events           | To chair the organization and promotion
| Massey Cashore            | Events           | To assist the president and other vice-
| Massey Cashore            | Events           | To chair the organization and promotion
| Jacob Burns               | President        | To be aware of MathSocs Policies and By
| Jacob Burns               | President        | To call and preside over general meetin
| Jacob Burns               | President        | To manage the executive team and the st
| Jacob Burns               | President        | To post announcements of all club meeti
| Ming Pan                  | Technology       | Maintain and update the club website.
| Ming Pan                  | Technology       | Maintain any hardware, software, or tec
| Ming Pan                  | Technology       | Perform the duties of a Vice President
| Zixin Nie                 | Events           | To assist the president and other vice-
| Zixin Nie                 | Events           | To chair the organization and promotion
| Simon wang                | Senior Advisor   | Have previous club management experienc
| Simon wang                | Senior Advisor   | To be aware of MathSoc's␣Policies␣and␣B
| Alice Wang                | Finance          | To ensure membership fees are collected
| Alice Wang                | Finance          | To keep an up-to-date record of financi
| Alice Wang                | Finance          | To prepare a summary of the financial r
| Alice Wang                | Finance          | To prepare the budget at the beginning
| Alice Wang                | Finance          | To volunteer as president in the absenc
+---------------------------+------------------+----------------------------------------------
```

NATURAL JOIN

NATURAL JOIN A join condition that lets the server decide on the join conditions based on:

- Same column names and types across columns for join

NATURAL JOIN Example

```
mysql> SELECT a.account_id, a.cust_id, c.cust_type_cd, c.fed_id
    -> FROM account a NATURAL JOIN customer c;
```

NATURAL JOIN Example

```
mysql> SELECT a.account_id, a.cust_id, c.cust_type_cd, c.fed_id
    -> FROM account a NATURAL JOIN customer c;

+------------+---------+--------------+-------------+
| account_id | cust_id | cust_type_cd | fed_id      |
+------------+---------+--------------+-------------+
|          1 |       1 | I            | 111-11-1111 |
|          2 |       1 | I            | 111-11-1111 |
|          3 |       1 | I            | 111-11-1111 |
|          4 |       2 | I            | 222-22-2222 |
|          5 |       2 | I            | 222-22-2222 |
|          6 |       3 | I            | 333-33-3333 |
|          7 |       3 | I            | 333-33-3333 |
|          8 |       4 | I            | 444-44-4444 |
|          9 |       4 | I            | 444-44-4444 |
|         10 |       4 | I            | 444-44-4444 |
|         11 |       5 | I            | 555-55-5555 |
|         12 |       6 | I            | 666-66-6666 |
|         13 |       6 | I            | 666-66-6666 |
|         14 |       7 | I            | 777-77-7777 |
|         15 |       8 | I            | 888-88-8888 |
|         16 |       8 | I            | 888-88-8888 |
|         17 |       9 | I            | 999-99-9999 |
|         18 |       9 | I            | 999-99-9999 |
|         19 |       9 | I            | 999-99-9999 |
|         20 |      10 | B            | 04-1111111  |
|         21 |      10 | B            | 04-1111111  |
|         22 |      11 | B            | 04-2222222  |
|         23 |      12 | B            | 04-3333333  |
|         24 |      13 | B            | 04-4444444  |
+------------+---------+--------------+-------------+
24 rows in set (0.02 sec)
```

OUTER JOIN

OUTER JOIN A join that returns all rows for 1 or 2 tables, even when there is no corresponding value. In these cases, NULL values are entered for these corresponding rows.

OUTER JOIN

OUTER JOIN A join that returns all rows for 1 or 2 tables, even when there is no corresponding value. In these cases, NULL values are entered for these corresponding rows.

OUTER JOIN

OUTER JOIN A join that returns all rows for 1 or 2 tables, even when there is no corresponding value. In these cases, NULL values are entered for these corresponding rows.

There are 3 types of OUTER JOINs:

1. **LEFT OUTER JOIN**: An OUTER JOIN returning all rows of the table first mentioned.

OUTER JOIN

OUTER JOIN A join that returns all rows for 1 or 2 tables, even when there is no corresponding value. In these cases, NULL values are entered for these corresponding rows.

There are 3 types of OUTER JOINs:

1. **LEFT OUTER JOIN**: An OUTER JOIN returning all rows of the table first mentioned.
2. **RIGHT OUTER JOIN**: An OUTER JOIN returning all rows of the table second mentioned.

OUTER JOIN

OUTER JOIN A join that returns all rows for 1 or 2 tables, even when there is no corresponding value. In these cases, NULL values are entered for these corresponding rows.

There are 3 types of OUTER JOINs:

1. **LEFT OUTER JOIN**: An OUTER JOIN returning all rows of the table first mentioned.
2. **RIGHT OUTER JOIN**: An OUTER JOIN returning all rows of the table second mentioned.
3. **FULL OUTER JOIN**: An OUTER JOIN returning all rows of both tables.

LEFT OUTER JOIN Example

```
mysql> SELECT c.cust_id, b.name
    -> FROM customer AS c LEFT OUTER JOIN business AS b
    ->     ON c.cust_id = b.cust_id;
```

LEFT OUTER JOIN Example

```
mysql> SELECT c.cust_id, b.name
    -> FROM customer AS c LEFT OUTER JOIN business AS b
    ->     ON c.cust_id = b.cust_id;

+---------+-------------------------+
| cust_id | name                    |
+---------+-------------------------+
|      10 | Chilton Engineering     |
|      11 | Northeast Cooling Inc.  |
|      12 | Superior Auto Body      |
|      13 | AAA Insurance Inc.      |
|       1 | NULL                    |
|       2 | NULL                    |
|       3 | NULL                    |
|       4 | NULL                    |
|       5 | NULL                    |
|       6 | NULL                    |
|       7 | NULL                    |
|       8 | NULL                    |
|       9 | NULL                    |
+---------+-------------------------+
13 rows in set (0.00 sec)
```

RIGHT OUTER JOIN Example

Now, if we change to a RIGHT OUTER JOIN:

```
mysql> SELECT c.cust_id, b.name
    -> FROM customer AS c RIGHT OUTER JOIN business AS b
    ->     ON c.cust_id = b.cust_id;
```

RIGHT OUTER JOIN Example

Now, if we change to a RIGHT OUTER JOIN:

```
mysql> SELECT c.cust_id, b.name
    -> FROM customer AS c RIGHT OUTER JOIN business AS b
    ->     ON c.cust_id = b.cust_id;

+---------+------------------------+
| cust_id | name                   |
+---------+------------------------+
|      10 | Chilton Engineering    |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body     |
|      13 | AAA Insurance Inc.     |
+---------+------------------------+
4 rows in set (0.00 sec)
```

RIGHT OUTER JOIN Example

Switching the tables in the FROM clause:

```
mysql> SELECT c.cust_id, b.name
    -> FROM business AS b RIGHT OUTER JOIN customer AS c
    ->     ON c.cust_id = b.cust_id;
```

RIGHT OUTER JOIN Example

Switching the tables in the FROM clause:

```
mysql> SELECT c.cust_id, b.name
    -> FROM business AS b RIGHT OUTER JOIN customer AS c
    ->     ON c.cust_id = b.cust_id;

+---------+------------------------+
| cust_id | name                   |
+---------+------------------------+
|      10 | Chilton Engineering    |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body     |
|      13 | AAA Insurance Inc.     |
|       1 | NULL                   |
|       2 | NULL                   |
|       3 | NULL                   |
|       4 | NULL                   |
|       5 | NULL                   |
|       6 | NULL                   |
|       7 | NULL                   |
|       8 | NULL                   |
|       9 | NULL                   |
+---------+------------------------+
13 rows in set (0.00 sec)
```

RIGHT OUTER JOIN Example

Switching the tables in the FROM clause:

```
mysql> SELECT c.cust_id, b.name
    -> FROM business AS b RIGHT OUTER JOIN customer AS c
    ->      ON c.cust_id = b.cust_id;

+---------+------------------------+
| cust_id | name                   |
+---------+------------------------+
|      10 | Chilton Engineering    |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body     |
|      13 | AAA Insurance Inc.     |
|       1 | NULL                   |
|       2 | NULL                   |
|       3 | NULL                   |
|       4 | NULL                   |
|       5 | NULL                   |
|       6 | NULL                   |
|       7 | NULL                   |
|       8 | NULL                   |
|       9 | NULL                   |
+---------+------------------------+
13 rows in set (0.00 sec)
```

We get the original table back with NULL values.

FULL OUTER JOIN Example

FULL OUTER JOIN is **not** implemented in MySQL, but in SQL versions that does, it will return all rows for both tables.

```
some sql> SELECT c.cust_id, b.name
       -> FROM customer AS c FULL OUTER JOIN business AS b
       ->     ON c.cust_id = b.cust_id;
```

FULL OUTER JOIN Example

FULL OUTER JOIN is **not** implemented in MySQL, but in SQL versions that does, it will return all rows for both tables.

```
some sql> SELECT c.cust_id, b.name
       -> FROM customer AS c FULL OUTER JOIN business AS b
       ->    ON c.cust_id = b.cust_id;

+---------+--------------------------+
| cust_id | name                     |
+---------+--------------------------+
|      10 | Chilton Engineering      |
|      11 | Northeast Cooling Inc.   |
|      12 | Superior Auto Body       |
|      13 | AAA Insurance Inc.       |
|       1 | NULL                     |
|       2 | NULL                     |
|       3 | NULL                     |
|       4 | NULL                     |
|       5 | NULL                     |
|       6 | NULL                     |
|       7 | NULL                     |
|       8 | NULL                     |
|       9 | NULL                     |
+---------+--------------------------+
13 rows in set (0.00 sec)
```

Subqueries

Subqueries are queries contained into queries. These subqueries are contained in '(', ')'

There are two types of subqueries:

1. **Non-Correlated Subqueries**: Can be run **independently** of the larger query.

Subqueries

Subqueries are queries contained into queries. These subqueries are contained in '(', ')'

There are two types of subqueries:

1. **Non-Correlated Subqueries**: Can be run **independently** of the larger query.
2. **Correlated Subqueries**: Must be run concurrently with the outer query. They are **dependent** on the outer query.

Non-Correlated Subqueries

Non-Correlated Subquery Any valid query within query that if executed by itself will produce a result (including empty set). These are enclosed in '(', ')' in **FROM**, **WHERE**, or **HAVING** clause.

Non-Correlated Subqueries

Non-Correlated Subquery Any valid query within query that if executed by itself will produce a result (including empty set). These are enclosed in '(', ')' in **FROM**, **WHERE**, or **HAVING** clause.

What Stat Club Exec is in charge of posters?

Non-Correlated Subqueries

Non-Correlated Subquery Any valid query within query that if executed by itself will produce a result (including empty set). These are enclosed in '(', ')' in **FROM**, **WHERE**, or **HAVING** clause.

What Stat Club Exec is in charge of posters?

Non-Correlated Subquery Any valid query within query that if executed by itself will produce a result (including empty set). These are enclosed in '(', ')' in **FROM**, **WHERE**, or **HAVING** clause.

What Stat Club Exec is in charge of posters?

```
mysql> SELECT e.name, e.position
    -> FROM exec AS e
    -> WHERE e.questid in (SELECT poster FROM event);
```

Non-Correlated Subqueries

Non-Correlated Subquery Any valid query within query that if executed by itself will produce a result (including empty set). These are enclosed in '(', ')' in **FROM**, **WHERE**, or **HAVING** clause.

What Stat Club Exec is in charge of posters?

```
mysql> SELECT e.name, e.position
    -> FROM exec AS e
    -> WHERE e.questid in (SELECT poster FROM event);

+-------------------------+-----------+
| name                    | position  |
+-------------------------+-----------+
| Ajanthan Thavaraja (Aj) | Events    |
| Darrell Aucoin          | President |
| Jacob Burns             | President |
+-------------------------+-----------+
```

WITH Clause

WITH clause: Makes a non-correlated subquery look like a table in the executed statement:

```
WITH <subquery_name> [(colname1, ...)] AS
(SELECT ...),
<subquery_name2> [(colname1, ...)] AS
(SELECT ...)
/* which then is used in the query */
SELECT ...
```

WITH Clause

WITH clause Makes a non-correlated subquery look like a table in the executed statement:

- Increases readability of the query as well as ensure that if it is used in several different places, it will only be executed once

```
WITH <subquery_name> [(colname1, ...)] AS
(SELECT ...),
<subquery_name2> [(colname1, ...)] AS
(SELECT ...)
/* which then is used in the query */
SELECT ...
```

WITH Clause

WITH clause Makes a non-correlated subquery look like a table in the executed statement:

- Increases readability of the query as well as ensure that if it is used in several different places, it will only be executed once
- This clause is **NOT** implemented in MySQL

```
WITH <subquery_name> [(colname1, ...)] AS
(SELECT ...),
<subquery_name2> [(colname1, ...)] AS
(SELECT ...)
/* which then is used in the query */
SELECT ...
```

Correlated Subqueries

Correlated Subquery makes references it's containing query, executing it for every candidate row referenced.

Correlated Subqueries

Correlated Subquery makes references it's containing query, executing it for every candidate row referenced.

Correlated Subqueries

Correlated Subquery makes references it's containing query, executing it for every candidate row referenced.

```
mysql> SELECT c.cust_id, c.cust_type_cd, c.city
    -> FROM customer AS c
    -> WHERE (SELECT SUM(a.avail_balance)
    ->     FROM account AS a
    ->     WHERE a.cust_id = c.cust_id)
    ->  BETWEEN 5000 AND 10000;
```

Correlated Subqueries

Correlated Subquery makes references it's containing query, executing it for every candidate row referenced.

```
mysql> SELECT c.cust_id, c.cust_type_cd, c.city
    -> FROM customer AS c
    -> WHERE (SELECT SUM(a.avail_balance)
    ->     FROM account AS a
    ->     WHERE a.cust_id = c.cust_id)
    ->   BETWEEN 5000 AND 10000;

+---------+--------------+------------+
| cust_id | cust_type_cd | city       |
+---------+--------------+------------+
|       4 | I            | Waltham    |
|       7 | I            | Wilmington |
|      11 | B            | Wilmington |
+---------+--------------+------------+
3 rows in set (0.00 sec)
```

Correlated Subqueries in SELECT Clause

Correlated subqueries can be used in the **SELECT**, as well as the **WHERE**, and **HAVING** clauses.

Correlated Subqueries in SELECT Clause

Correlated subqueries can be used in the **SELECT**, as well as the **WHERE**, and **HAVING** clauses.

```
mysql> SELECT c.cust_id, c.cust_type_cd, c.city,
    ->      (SELECT SUM(a.avail_balance)
    ->      FROM account a
    ->      WHERE a.cust_id = c.cust_id) AS balance
    -> FROM customer AS c;
```

Correlated Subqueries in SELECT Clause

Correlated subqueries can be used in the **SELECT**, as well as the **WHERE**, and **HAVING** clauses.

```
mysql> SELECT c.cust_id, c.cust_type_cd, c.city,
    ->        (SELECT SUM(a.avail_balance)
    ->        FROM account a
    ->        WHERE a.cust_id = c.cust_id) AS balance
    -> FROM customer AS c;
```

cust_id	cust_type_cd	city	balance
1	I	Lynnfield	4557.75
2	I	Woburn	2458.02
3	I	Quincy	3270.25
4	I	Waltham	6788.98
5	I	Salem	2237.97
6	I	Waltham	10122.37
7	I	Wilmington	5000.00
8	I	Salem	3875.18
9	I	Newton	10971.22
10	B	Salem	23575.12

Correlated vs Non-Correlated

1. Correlated subquery is **dependent** on outer query, non-correlated is **independent**.

Correlated vs Non-Correlated

1. Correlated subquery is **dependent** on outer query, non-correlated is **independent**.
2. Correlated subquery is executed concurrently with outer query, non-correlated is executed before.

Correlated vs Non-Correlated

1. Correlated subquery is **dependent** on outer query, non-correlated is **independent**.
2. Correlated subquery is executed concurrently with outer query, non-correlated is executed before.
3. In general, for speed of execution:

$$\text{Correlated subquery} < \text{Non-Correlated subquery} < \text{Joins}$$

CASE Expressions

CASE expressions: Similar to a series of if else statements executed for every entry in a table. A new value is returned for every row in the table.

```
CASE [<column>]
WHEN <condition1> THEN <result1>
WHEN <condition2> THEN <result2>
...
WHEN <condition n> THEN <result n>
[ELSE <result>]
END
```

CASE Expressions

CASE expressions Similar to a series of if else statements executed for every entry in a table. A new value is returned for every row in the table.

```
CASE [<column>]
WHEN <condition1> THEN <result1>
WHEN <condition2> THEN <result2>
...
WHEN <condition n> THEN <result n>
[ELSE <result>]
END
```

- The result can be of any datatype or the result of a correlated or non-correlated subquery (if the result is a single)

CASE Expressions

CASE expressions Similar to a series of if else statements executed for every entry in a table. A new value is returned for every row in the table.

```
CASE [<column>]
WHEN <condition1> THEN <result1>
WHEN <condition2> THEN <result2>
...
WHEN <condition n> THEN <result n>
[ELSE <result>]
END
```

- The result can be of any datatype or the result of a correlated or non-correlated subquery (if the result is a single)
- CASE expressions are performed by themselves in the SELECT clause or within a function or aggregate function

CASE Expressions

CASE expressions Similar to a series of if else statements executed for every entry in a table. A new value is returned for every row in the table.

```
CASE [<column>]
WHEN <condition1> THEN <result1>
WHEN <condition2> THEN <result2>
...
WHEN <condition n> THEN <result n>
[ELSE <result>]
END
```

- The result can be of any datatype or the result of a correlated or non-correlated subquery (if the result is a single)
- CASE expressions are performed by themselves in the SELECT clause or within a function or aggregate function
  - CASE expressions within aggregate functions allow us to do counts, sums, averages, etc. of particular occurrences

CASE Expression Example

```
mysql> SELECT
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2000 THEN 1
    ->         ELSE 0
    ->       END) year_2000,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2001 THEN 1
    ->         ELSE 0
    ->       END) year_2001,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2002 THEN 1
    ->         ELSE 0
    ->       END) year_2002,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2003 THEN 1
    ->         ELSE 0
    ->       END) year_2003,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2004 THEN 1
    ->         ELSE 0
    ->       END) year_2004,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2005 THEN 1
    ->         ELSE 0
    ->       END) year_2005
    -> FROM account
    -> WHERE open_date > '1999-12-31' AND open_date < '2006-01-01';
```

CASE Expression Example

year_2000	year_2001	year_2002	year_2003	year_2004	year_2005
3	4	5	3	9	0

1 row in set (0.01 sec)

UNION Operator

UNION operator $(A \cup B)$: Addition of one result set to another result set with the same number of attributes and types.

```
SELECT ... FROM ...
UNION [ALL]
SELECT ... FROM ...
```

UNION Operator

UNION operator ($A \cup B$): Addition of one result set to another result set with the same number of attributes and types.

```
SELECT ... FROM ...
UNION [ALL]
SELECT ... FROM ...
```

- Just UNION removes duplicates, while UNION ALL keeps all rows from both result sets.

UNION Operator Example

```
mysql> SELECT 'IND' type_cd, cust_id, lname name
    -> FROM individual
    -> UNION ALL
    -> SELECT 'BUS' type_cd, cust_id, name
    -> FROM business;
```

UNION Operator Example

```
mysql> SELECT 'IND' type_cd, cust_id, lname name
    -> FROM individual
    -> UNION ALL
    -> SELECT 'BUS' type_cd, cust_id, name
    -> FROM business;

+---------+---------+------------------------+
| type_cd | cust_id | name                   |
+---------+---------+------------------------+
| IND     |       1 | Hadley                 |
| IND     |       2 | Tingley                |
| IND     |       3 | Tucker                 |
| IND     |       4 | Hayward                |
| IND     |       5 | Frasier                |
| IND     |       6 | Spencer                |
| IND     |       7 | Young                  |
| IND     |       8 | Blake                  |
| IND     |       9 | Farley                 |
| BUS     |      10 | Chilton Engineering    |
| BUS     |      11 | Northeast Cooling Inc. |
| BUS     |      12 | Superior Auto Body     |
| BUS     |      13 | AAA Insurance Inc.     |
+---------+---------+------------------------+
13 rows in set (0.04 sec)
```

INTERSECT Operator

INTERSECT operator ($A \cap B$): Returns only tuples that are in common between two result sets. Result sets must be equal in number and type of attributes.

```
SELECT ... FROM ...
INTERSECT
SELECT ... FROM ...
```

INTERSECT Operator

INTERSECT operator ($A \cap B$): Returns only tuples that are in common between two result sets. Result sets must be equal in number and type of attributes.

```
SELECT ... FROM ...
INTERSECT
SELECT ... FROM ...
```

- INTERSECT operator is **not** implemented in MySQL

INTERSECT Operator Example

```
SELECT emp_id, fname, lname
FROM employee
INTERSECT
SELECT cust_id, fname, lname
FROM individual;
```

INTERSECT Operator Example

```
SELECT emp_id, fname, lname
FROM employee
INTERSECT
SELECT cust_id, fname, lname
FROM individual;
```

```
Empty set (0.04 sec
```

EXCEPT Operator

EXCEPT operator ($A \backslash B$): Returns the first result set minus anything it has in common with the second result set.

```
SELECT ... FROM ...
EXCEPT [ALL]
SELECT ... FROM ...
```

EXCEPT Operator

EXCEPT operator ($A \backslash B$): Returns the first result set minus anything it has in common with the second result set.

```
SELECT ... FROM ...
EXCEPT [ALL]
SELECT ... FROM ...
```

- EXCEPT operator is **not** implemented in MySQL

EXCEPT Operator

EXCEPT operator ($A \setminus B$): Returns the first result set minus anything it has in common with the second result set.

```
SELECT ... FROM ...
EXCEPT [ALL]
SELECT ... FROM ...
```

- EXCEPT operator is **not** implemented in MySQL
- Oracle uses a non-ANSI-compliant `minus` operator

EXCEPT Operator

EXCEPT operator ($A \setminus B$): Returns the first result set minus anything it has in common with the second result set.

```
SELECT ... FROM ...
EXCEPT [ALL]
SELECT ... FROM ...
```

- EXCEPT operator is **not** implemented in MySQL
- Oracle uses a non-ANSI-compliant `minus` operator
- Just EXCEPT uses set theory version of minus.

EXCEPT Operator

EXCEPT operator ($A \setminus B$): Returns the first result set minus anything it has in common with the second result set.

```
SELECT ... FROM ...
EXCEPT [ALL]
SELECT ... FROM ...
```

- EXCEPT operator is **not** implemented in MySQL
- Oracle uses a non-ANSI-compliant `minus` operator
- Just EXCEPT uses set theory version of minus.
  - If B has a row in common with A then all rows matching that row is removed

EXCEPT Operator

EXCEPT operator ($A \backslash B$): Returns the first result set minus anything it has in common with the second result set.

```
SELECT ... FROM ...
EXCEPT [ALL]
SELECT ... FROM ...
```

- EXCEPT operator is **not** implemented in MySQL
- Oracle uses a non-ANSI-compliant `minus` operator
- Just EXCEPT uses set theory version of minus.
    - If B has a row in common with A then all rows matching that row is removed
- The optional ALL uses the bag semantics version.

EXCEPT Operator

EXCEPT operator ($A \backslash B$): Returns the first result set minus anything it has in common with the second result set.

```
SELECT ... FROM ...
EXCEPT [ALL]
SELECT ... FROM ...
```

- EXCEPT operator is **not** implemented in MySQL
- Oracle uses a non-ANSI-compliant `minus` operator
- Just EXCEPT uses set theory version of minus.
  - If B has a row in common with A then all rows matching that row is removed
- The optional ALL uses the bag semantics version.
  - If B has a row in common with A then only the number of common rows in B rows matching that row is removed

EXCEPT Operator Example

```
SELECT emp_id
FROM employee
WHERE assigned_branch_id = 2
  AND (title = 'Teller' OR title = 'Head Teller')
EXCEPT
SELECT DISTINCT open_emp_id
FROM account
WHERE open_branch_id = 2;
```

EXCEPT Operator Example

```
SELECT emp_id
FROM employee
WHERE assigned_branch_id = 2
  AND (title = 'Teller' OR title = 'Head Teller')
EXCEPT
SELECT DISTINCT open_emp_id
FROM account
WHERE open_branch_id = 2;
```

```
+--------+
| emp_id |
+--------+
|     11 |
|     12 |
+--------+
2 rows in set (0.01 sec)
```

Functions and Operators

There are too many functions in SQL and each implementation of SQL to list here. Here are some random functions in MySQL:

CURRENT_TIMESTAMP() Returns the datetime of being executed.

YEAR(d) Return the year for date d.

MONTH(d) Return the month for date d.

DAY(d) Return the day of the month for date d.

DAYNAME(d) Return the name of the weekday of date d.

FLOOR(n) Floor the numeric value n.

CONCAT(str1, str2,...) Concatenate the strings to one string.

LOWER(s) Return the lower case version of the string s.

LOG(n) Return the natural log of number n.

CURTIME() Return the current time.

RAND() Return a random floating-point number.

Using SQL within R

For SQL within R, we usually need two packages:

1. **DBI:** R Database Interface
2. Specific package for the individual SQL implementation (MySQL, SQLite, Oracle, etc.).

RMySQL for MySQL

DBI Functions

DBI contains various virtual classes and functions in connecting and querying a database:

dbDriver Driver specifying the operations for creating connections to SQL Servers

```
m = dbDriver("MySQL")
# equivalent to MySQL()
```

dbConnect Connect to a DBMS.

```
conn = dbConnect(m, user="darrell", db="bank", host
   ="localhost", password="pass")
```

dbDisconnect Disconnect from a DBMS. You should always disconnect after you no longer need it

```
dbDisconnect(conn)
```

DBI Functions

dbSendQuery Submits and executes SQL statement (information retrieved using fetch)

```
query = dbSendQuery(conn, "SELECT * FROM account;")
```

dbGetQuery Submits, executes SQL statement and retrieves information

```
res = dbGetQuery(conn, "SELECT * FROM account;")
```

fetch Get records from a dbSendQuery

```
max.num.row = 100
res = fetch(query, n=max.num.row)
```

dbListTables List tables in database connection

```
tables = dbListTables(conn)
```

dbGetInfo Get meta-data for DBIObjects

```
meta.data = dbGetInfo(query)
```

DBI Functions

dbReadTable Fetch the data from a table.

```
res = dbReadTable(conn, "table")
```

dbListFields Return the column names for a given table

```
columns = dbListFields(conn, "table")
```

Connecting to Local FileSystem

```
library("RMySQL")

## Loading required package:  methods
## Loading required package:  DBI

m = dbDriver("MySQL")
conn = dbConnect(m, user="darrell", db="bank",
                                host="localhost",
                                password="pass")
query = dbSendQuery(conn,
                    "select *
                    from account;")
result = fetch(query)
dbDisconnect(conn)

## [1] TRUE
```

Connecting to Local FileSystem

```
head(result)

##   account_id product_cd cust_id  open_date close_date last_a
## 1          1        CHK       1 2000-01-15       <NA>
## 2          2        SAV       1 2000-01-15       <NA>
## 3          3         CD       1 2004-06-30       <NA>
## 4          4        CHK       2 2001-03-12       <NA>
## 5          5        SAV       2 2001-03-12       <NA>
## 6          7        CHK       3 2002-11-23       <NA>
##   status open_branch_id open_emp_id avail_balance pending_ba
## 1 ACTIVE              2          10          1058
## 2 ACTIVE              2          10           500
## 3 ACTIVE              2          10          3000
## 4 ACTIVE              2          10          2258
## 5 ACTIVE              2          10           200
## 6 ACTIVE              3          13          1058
```

Connecting to Remote Server

```
hg19 = dbConnect(MySQL(), user="genome", db="hg19",
        host="genome-mysql.cse.ucsc.edu")
result = dbGetQuery(hg19,
                "SELECT COUNT(*) FROM affyU133Plus2")
dbDisconnect(hg19)

## [1] TRUE

print(result)

##   COUNT(*)
## 1    58463
```

Books:

Learning SQL, Second Edition, by Alan Beaulieu. Copyright 2009 O'Reilly Media, Inc., 978-0-596-52083-0.

- Many code examples are borrowed from this book

CS Database Courses:

CS 338: Computer Applications in Business: Databases (CS Minors)

CS 348: Introduction to Database Management (CS Majors)

Coursera Courses:

Introduction to Databases: by Jennifer Widom from Stanford

Getting and Cleaning Data: by Jeff Leek, PhD, Roger D. Peng, PhD, Brian Caffo, PhD from Johns Hopkins University

Thanks

Thanks to the Stats Club Execs for helping pull off this event and Grant Weddell for letting me listen in on his CS 348 Database class.

Topics not mentioned:

- Views
- Transactions
- How to setup a database: insert data, create constraints, indexes
    - Application interfaces
- How to import/export data: csv, tsv, XML, etc.