

Introduction to SQL

Phil Spector

Statistical Computing Facility
University of California, Berkeley

What is SQL?

What is SQL?

- ▶ Structured Query Language

What is SQL?

- ▶ Structured Query Language
- ▶ Usually “talk” to a database server

What is SQL?

- ▶ Structured Query Language
- ▶ Usually “talk” to a database server
- ▶ Used as front end to many databases (mysql, postgresql, oracle, sybase)

What is SQL?

- ▶ Structured Query Language
- ▶ Usually “talk” to a database server
- ▶ Used as front end to many databases (mysql, postgresql, oracle, sybase)
- ▶ Three Subsystems: data description, data access and privileges

What is SQL?

- ▶ Structured Query Language
- ▶ Usually “talk” to a database server
- ▶ Used as front end to many databases (mysql, postgresql, oracle, sybase)
- ▶ Three Subsystems: data description, data access and privileges
- ▶ Optimized for certain data arrangements

What is SQL?

- ▶ Structured Query Language
- ▶ Usually “talk” to a database server
- ▶ Used as front end to many databases (mysql, postgresql, oracle, sybase)
- ▶ Three Subsystems: data description, data access and privileges
- ▶ Optimized for certain data arrangements
- ▶ The language is case-sensitive, but I use upper case for keywords.

When do you need a Database?

When do you need a Database?

- ▶ Multiple simultaneous changes to data (concurrency)

When do you need a Database?

- ▶ Multiple simultaneous changes to data (concurrency)
- ▶ Data changes on a regular basis

When do you need a Database?

- ▶ Multiple simultaneous changes to data (concurrency)
- ▶ Data changes on a regular basis
- ▶ Large data sets where you only need some observations/variables

When do you need a Database?

- ▶ Multiple simultaneous changes to data (concurrency)
- ▶ Data changes on a regular basis
- ▶ Large data sets where you only need some observations/variables
- ▶ Share huge data set among many people

When do you need a Database?

- ▶ Multiple simultaneous changes to data (concurrency)
- ▶ Data changes on a regular basis
- ▶ Large data sets where you only need some observations/variables
- ▶ Share huge data set among many people
- ▶ Rapid queries with no analysis

When do you need a Database?

- ▶ Multiple simultaneous changes to data (concurrency)
- ▶ Data changes on a regular basis
- ▶ Large data sets where you only need some observations/variables
- ▶ Share huge data set among many people
- ▶ Rapid queries with no analysis
- ▶ Web interfaces to data, especially dynamic data

Uses of Databases

Traditional Uses:

Uses of Databases

Traditional Uses:

- ▶ Live Queries

Uses of Databases

Traditional Uses:

- ▶ Live Queries
- ▶ Report Generation

Uses of Databases

Traditional Uses:

- ▶ Live Queries
- ▶ Report Generation
- ▶ Normalization, foreign keys, joins, etc.

Uses of Databases

Traditional Uses:

- ▶ Live Queries
- ▶ Report Generation
- ▶ Normalization, foreign keys, joins, etc.

Newer uses:

Uses of Databases

Traditional Uses:

- ▶ Live Queries
- ▶ Report Generation
- ▶ Normalization, foreign keys, joins, etc.

Newer uses:

- ▶ Storage - data is extracted and analyzed in another application

Uses of Databases

Traditional Uses:

- ▶ Live Queries
- ▶ Report Generation
- ▶ Normalization, foreign keys, joins, etc.

Newer uses:

- ▶ Storage - data is extracted and analyzed in another application
- ▶ Backends to web sites

Uses of Databases

Traditional Uses:

- ▶ Live Queries
- ▶ Report Generation
- ▶ Normalization, foreign keys, joins, etc.

Newer uses:

- ▶ Storage - data is extracted and analyzed in another application
- ▶ Backends to web sites
- ▶ Traditional rules may not be as important

Ways to Use SQL

Ways to Use SQL

- ▶ console command (`mysql -u user -p dbname`)

Ways to Use SQL

- ▶ console command (`mysql -u user -p dbname`)
- ▶ GUI interfaces are often available

Ways to Use SQL

- ▶ console command (`mysql -u user -p dbname`)
- ▶ GUI interfaces are often available
- ▶ Interfaces to many programming languages: R, python, perl, PHP, etc.

Ways to Use SQL

- ▶ console command (`mysql -u user -p dbname`)
- ▶ GUI interfaces are often available
- ▶ Interfaces to many programming languages: R, python, perl, PHP, etc.
- ▶ SQLite - use SQL without a database server

Ways to Use SQL

- ▶ console command (`mysql -u user -p dbname`)
- ▶ GUI interfaces are often available
- ▶ Interfaces to many programming languages: R, python, perl, PHP, etc.
- ▶ SQLite - use SQL without a database server
- ▶ PROC SQL in SAS

Some Relational Database Concepts

Some Relational Database Concepts

- ▶ A database server can contain many databases

Some Relational Database Concepts

- ▶ A database server can contain many databases
- ▶ Databases are collections of tables

Some Relational Database Concepts

- ▶ A database server can contain many databases
- ▶ Databases are collections of tables
- ▶ Tables are two-dimensional with rows (observations) and columns (variables)

Some Relational Database Concepts

- ▶ A database server can contain many databases
- ▶ Databases are collections of tables
- ▶ Tables are two-dimensional with rows (observations) and columns (variables)
- ▶ Limited mathematical and summary operations available

Some Relational Database Concepts

- ▶ A database server can contain many databases
- ▶ Databases are collections of tables
- ▶ Tables are two-dimensional with rows (observations) and columns (variables)
- ▶ Limited mathematical and summary operations available
- ▶ Very good at combining information from several tables

Finding Your Way Around the Server

Since a single server can support many databases, each containing many tables, with each table having a variety of columns, it's easy to get lost when you're working with databases. These commands will help figure out what's available:

Finding Your Way Around the Server

Since a single server can support many databases, each containing many tables, with each table having a variety of columns, it's easy to get lost when you're working with databases. These commands will help figure out what's available:

- ▶ `SHOW DATABASES;`

Finding Your Way Around the Server

Since a single server can support many databases, each containing many tables, with each table having a variety of columns, it's easy to get lost when you're working with databases. These commands will help figure out what's available:

- ▶ `SHOW DATABASES;`
- ▶ `SHOW TABLES IN database;`

Finding Your Way Around the Server

Since a single server can support many databases, each containing many tables, with each table having a variety of columns, it's easy to get lost when you're working with databases. These commands will help figure out what's available:

- ▶ `SHOW DATABASES;`
- ▶ `SHOW TABLES IN database;`
- ▶ `SHOW COLUMNS IN table;`

Finding Your Way Around the Server

Since a single server can support many databases, each containing many tables, with each table having a variety of columns, it's easy to get lost when you're working with databases. These commands will help figure out what's available:

- ▶ `SHOW DATABASES;`
- ▶ `SHOW TABLES IN database;`
- ▶ `SHOW COLUMNS IN table;`
- ▶ `DESCRIBE table;` - shows the columns and their types

Variable Types

SQL supports a very large number of different formats for internal storage of information.

Variable Types

SQL supports a very large number of different formats for internal storage of information.

Numeric

Variable Types

SQL supports a very large number of different formats for internal storage of information.

Numeric

- ▶ INTEGER, SMALLINT, BIGINT

Variable Types

SQL supports a very large number of different formats for internal storage of information.

Numeric

- ▶ INTEGER, SMALLINT, BIGINT
- ▶ NUMERIC(w,d), DECIMAL(w,d) - numbers with width w and d decimal places

Variable Types

SQL supports a very large number of different formats for internal storage of information.

Numeric

- ▶ INTEGER, SMALLINT, BIGINT
- ▶ NUMERIC(w,d), DECIMAL(w,d) - numbers with width w and d decimal places
- ▶ REAL, DOUBLE PRECISION - machine and database dependent

Variable Types

SQL supports a very large number of different formats for internal storage of information.

Numeric

- ▶ INTEGER, SMALLINT, BIGINT
- ▶ NUMERIC(*w*,*d*), DECIMAL(*w*,*d*) - numbers with width *w* and *d* decimal places
- ▶ REAL, DOUBLE PRECISION - machine and database dependent
- ▶ FLOAT(*p*) - floating point number with *p* binary digits of precision

Variable Types (cont'd)

Character

Variable Types (cont'd)

Character

- ▶ `CHARACTER(L)` - a fixed-length character of length L

Variable Types (cont'd)

Character

- ▶ `CHARACTER(L)` - a fixed-length character of length L
- ▶ `CHARACTER VARYING(L)` or `VARCHAR(L)` - supports maximum length of L

Variable Types (cont'd)

Character

- ▶ `CHARACTER(L)` - a fixed-length character of length L
- ▶ `CHARACTER VARYING(L)` or `VARCHAR(L)` - supports maximum length of L

Binary

Variable Types (cont'd)

Character

- ▶ `CHARACTER(L)` - a fixed-length character of length L
- ▶ `CHARACTER VARYING(L)` or `VARCHAR(L)` - supports maximum length of L

Binary

- ▶ `BIT(L)`, `BIT VARYING(L)` - like corresponding characters

Variable Types (cont'd)

Character

- ▶ `CHARACTER(L)` - a fixed-length character of length L
- ▶ `CHARACTER VARYING(L)` or `VARCHAR(L)` - supports maximum length of L

Binary

- ▶ `BIT(L)`, `BIT VARYING(L)` - like corresponding characters
- ▶ `BINARY LARGE OBJECT(L)` or `BLOB(L)`

Variable Types (cont'd)

Character

- ▶ `CHARACTER(L)` - a fixed-length character of length L
- ▶ `CHARACTER VARYING(L)` or `VARCHAR(L)` - supports maximum length of L

Binary

- ▶ `BIT(L)`, `BIT VARYING(L)` - like corresponding characters
- ▶ `BINARY LARGE OBJECT(L)` or `BLOB(L)`

Temporal

Variable Types (cont'd)

Character

- ▶ CHARACTER(L) - a fixed-length character of length L
- ▶ CHARACTER VARYING(L) or VARCHAR(L) - supports maximum length of L

Binary

- ▶ BIT(L), BIT VARYING(L) - like corresponding characters
- ▶ BINARY LARGE OBJECT(L) or BLOB(L)

Temporal

- ▶ DATE

Variable Types (cont'd)

Character

- ▶ CHARACTER(L) - a fixed-length character of length L
- ▶ CHARACTER VARYING(L) or VARCHAR(L) - supports maximum length of L

Binary

- ▶ BIT(L), BIT VARYING(L) - like corresponding characters
- ▶ BINARY LARGE OBJECT(L) or BLOB(L)

Temporal

- ▶ DATE
- ▶ TIME

Variable Types (cont'd)

Character

- ▶ CHARACTER(L) - a fixed-length character of length L
- ▶ CHARACTER VARYING(L) or VARCHAR(L) - supports maximum length of L

Binary

- ▶ BIT(L), BIT VARYING(L) - like corresponding characters
- ▶ BINARY LARGE OBJECT(L) or BLOB(L)

Temporal

- ▶ DATE
- ▶ TIME
- ▶ TIMESTAMP

CREATE TABLE statement

Suppose we have data measured on the height and weight of children over a range of ages. The first step is deciding on the appropriate variable types, and creating the table with the `CREATE TABLE` command.

CREATE TABLE statement

Suppose we have data measured on the height and weight of children over a range of ages. The first step is deciding on the appropriate variable types, and creating the table with the `CREATE TABLE` command.

```
CREATE TABLE kids(id CHAR(6),  
                   race SMALLINT,  
                   age DECIMAL(6,3),  
                   height DECIMAL(7,3),  
                   weight DECIMAL(7,3),  
                   sex SMALLINT);
```

Entering observations into a table

We could now enter individual items with the `INSERT` command:

```
INSERT INTO kids VALUES(100011,2,10.346,  
                          148.5,38.95,1);
```

This quickly gets tedious. We can automate the process using the `LOAD DATA` command:

Entering observations into a table

We could now enter individual items with the `INSERT` command:

```
INSERT INTO kids VALUES(100011,2,10.346,  
                          148.5,38.95,1);
```

This quickly gets tedious. We can automate the process using the `LOAD DATA` command:

```
LOAD DATA INFILE 'kids.tab'  
            INTO TABLE kids  
            FIELDS TERMINATED BY '\t';
```

Entering observations into a table

We could now enter individual items with the `INSERT` command:

```
INSERT INTO kids VALUES(100011,2,10.346,  
                          148.5,38.95,1);
```

This quickly gets tedious. We can automate the process using the `LOAD DATA` command:

```
LOAD DATA INFILE 'kids.tab'  
          INTO TABLE kids  
          FIELDS TERMINATED BY '\t';
```

This will read an entire tab-separated file into the database in one command.

Comparison Operators

In SQL, the **WHERE** clause allows you to operate on subsets of a table. The following comparison operators are available:

Comparison Operators

In SQL, the **WHERE** clause allows you to operate on subsets of a table. The following comparison operators are available:

- Usual logical operators: < > <= >= = <>

Comparison Operators

In SQL, the **WHERE** clause allows you to operate on subsets of a table. The following comparison operators are available:

- ▶ Usual logical operators: `<` `>` `<=` `>=` `=` `<>`
- ▶ **BETWEEN** used to test for a range

Comparison Operators

In SQL, the **WHERE** clause allows you to operate on subsets of a table. The following comparison operators are available:

- ▶ Usual logical operators: < > <= >= = <>
- ▶ **BETWEEN** used to test for a range
- ▶ **IN** used to test group membership
- ▶ Keyword **NOT** used for negation

Comparison Operators

In SQL, the **WHERE** clause allows you to operate on subsets of a table. The following comparison operators are available:

- ▶ Usual logical operators: < > <= >= = <>
- ▶ **BETWEEN** used to test for a range
- ▶ **IN** used to test group membership
- ▶ Keyword **NOT** used for negation
- ▶ **LIKE** operator allows wildcards

Comparison Operators

In SQL, the **WHERE** clause allows you to operate on subsets of a table. The following comparison operators are available:

- ▶ Usual logical operators: < > <= >= = <>
- ▶ **BETWEEN** used to test for a range
- ▶ **IN** used to test group membership
- ▶ Keyword **NOT** used for negation
- ▶ **LIKE** operator allows wildcards
 - ▶ _ means single character, % means anything

Comparison Operators

In SQL, the **WHERE** clause allows you to operate on subsets of a table. The following comparison operators are available:

- ▶ Usual logical operators: `<` `>` `<=` `>=` `=` `<>`
- ▶ **BETWEEN** used to test for a range
- ▶ **IN** used to test group membership
- ▶ Keyword **NOT** used for negation
- ▶ **LIKE** operator allows wildcards
 - ▶ `_` means single character, `%` means anything
 - ▶ `SELECT salary WHERE name LIKE 'Fred %';`

Comparison Operators

In SQL, the **WHERE** clause allows you to operate on subsets of a table. The following comparison operators are available:

- ▶ Usual logical operators: `<` `>` `<=` `>=` `=` `<>`
- ▶ **BETWEEN** used to test for a range
- ▶ **IN** used to test group membership
- ▶ Keyword **NOT** used for negation
- ▶ **LIKE** operator allows wildcards
 - ▶ `_` means single character, `%` means anything
 - ▶ `SELECT salary WHERE name LIKE 'Fred %';`
- ▶ **RLIKE** operator allows regular expressions

Comparison Operators

In SQL, the **WHERE** clause allows you to operate on subsets of a table. The following comparison operators are available:

- ▶ Usual logical operators: `<` `>` `<=` `>=` `=` `<>`
- ▶ **BETWEEN** used to test for a range
- ▶ **IN** used to test group membership
- ▶ Keyword **NOT** used for negation
- ▶ **LIKE** operator allows wildcards
 - ▶ `_` means single character, `%` means anything
 - ▶ `SELECT salary WHERE name LIKE 'Fred %';`
- ▶ **RLIKE** operator allows regular expressions
- ▶ Use **AND**(`&&`) and **OR**(`||`) to combine conditions

Updating a Table

To change some of the values of columns of a table, you can use the **UPDATE** command. Changes are provided as a comma-separated list of column/value pairs.

Updating a Table

To change some of the values of columns of a table, you can use the **UPDATE** command. Changes are provided as a comma-separated list of column/value pairs.

For example, to add one to the weight of an observation in the **kids** table where **id** is 101311 and **age** is between 9 and 10, we could use:

Updating a Table

To change some of the values of columns of a table, you can use the `UPDATE` command. Changes are provided as a comma-separated list of column/value pairs.

For example, to add one to the weight of an observation in the `kids` table where `id` is 101311 and `age` is between 9 and 10, we could use:

```
UPDATE kids SET weight=weight + 1
              WHERE id='101311' AND
              age BETWEEN 9 and 10;
```

Updating a Table

To change some of the values of columns of a table, you can use the `UPDATE` command. Changes are provided as a comma-separated list of column/value pairs.

For example, to add one to the weight of an observation in the `kids` table where `id` is 101311 and `age` is between 9 and 10, we could use:

```
UPDATE kids SET weight=weight + 1
              WHERE id='101311' AND
              age BETWEEN 9 and 10;
```

Be careful with `UPDATE`, because if you don't provide a `WHERE` clause, all the rows of the table will be changed.

The `SELECT` statement

For many of the modern uses of databases, all you'll need to do with the database is to select some subset of the variables and/or observations from a table, and let some other program manipulate them. In SQL the `SELECT` statement is the workhorse for these operations.

The SELECT statement

For many of the modern uses of databases, all you'll need to do with the database is to select some subset of the variables and/or observations from a table, and let some other program manipulate them. In SQL the **SELECT** statement is the workhorse for these operations.

```
SELECT columns or computations
FROM table
WHERE condition
GROUP BY columns
HAVING condition
ORDER BY column [ASC | DESC]
LIMIT offset,count;
```

Examples of SELECT queries

Suppose we wish to simply see all of the data:

Examples of SELECT queries

Suppose we wish to simply see all of the data:

```
SELECT * FROM kids;
```

▶ View

Examples of SELECT queries

Suppose we wish to simply see all of the data:

```
SELECT * FROM kids;
```

[▶ View](#)

Find the age, race, height and weight for any observations with weight greater than 80kg and height less than 150cm:

Examples of SELECT queries

Suppose we wish to simply see all of the data:

```
SELECT * FROM kids;
```

[▶ View](#)

Find the age, race, height and weight for any observations with weight greater than 80kg and height less than 150cm:

```
SELECT age,race,height,weight FROM kids  
WHERE weight > 80 AND height < 150;
```

[▶ View](#)

Examples of SELECT queries

Suppose we wish to simply see all of the data:

```
SELECT * FROM kids;
```

[▶ View](#)

Find the age, race, height and weight for any observations with weight greater than 80kg and height less than 150cm:

```
SELECT age,race,height,weight FROM kids  
WHERE weight > 80 AND height < 150;
```

[▶ View](#)

Find all information about the 10 tallest observations:

Examples of SELECT queries

Suppose we wish to simply see all of the data:

```
SELECT * FROM kids;
```

[▶ View](#)

Find the age, race, height and weight for any observations with weight greater than 80kg and height less than 150cm:

```
SELECT age,race,height,weight FROM kids  
      WHERE weight > 80 AND height < 150;
```

[▶ View](#)

Find all information about the 10 tallest observations:

```
SELECT * FROM kids  
      ORDER BY height DESC limit 1,10;
```

[▶ View](#)

Examples of SELECT queries

Suppose we wish to simply see all of the data:

```
SELECT * FROM kids;
```

[▶ View](#)

Find the age, race, height and weight for any observations with weight greater than 80kg and height less than 150cm:

```
SELECT age,race,height,weight FROM kids  
WHERE weight > 80 AND height < 150;
```

[▶ View](#)

Find all information about the 10 tallest observations:

```
SELECT * FROM kids  
ORDER BY height DESC limit 1,10;
```

[▶ View](#)

Find all information about observations where age is from 17 to 18 and weight is from 180 to 185:

Examples of SELECT queries

Suppose we wish to simply see all of the data:

```
SELECT * FROM kids;
```

[▶ View](#)

Find the age, race, height and weight for any observations with weight greater than 80kg and height less than 150cm:

```
SELECT age,race,height,weight FROM kids  
WHERE weight > 80 AND height < 150;
```

[▶ View](#)

Find all information about the 10 tallest observations:

```
SELECT * FROM kids  
ORDER BY height DESC limit 1,10;
```

[▶ View](#)

Find all information about observations where age is from 17 to 18 and weight is from 180 to 185:

```
SELECT * FROM kids WHERE age BETWEEN 17 AND 18  
AND weight BETWEEN 180 AND 185;
```

[▶ View](#)

Summaries and Computations

SQL supports basic arithmetic operations to create new columns, as well as some summarization functions which include

Summaries and Computations

SQL supports basic arithmetic operations to create new columns, as well as some summarization functions which include

- ▶ `COUNT()`

Summaries and Computations

SQL supports basic arithmetic operations to create new columns, as well as some summarization functions which include

- ▶ `COUNT()`
- ▶ `AVG()` (mean)

Summaries and Computations

SQL supports basic arithmetic operations to create new columns, as well as some summarization functions which include

- ▶ `COUNT()`
- ▶ `AVG()` (mean)
- ▶ `SUM()`

Summaries and Computations

SQL supports basic arithmetic operations to create new columns, as well as some summarization functions which include

- ▶ `COUNT()`
- ▶ `AVG()` (mean)
- ▶ `SUM()`
- ▶ `MIN()`

Summaries and Computations

SQL supports basic arithmetic operations to create new columns, as well as some summarization functions which include

- ▶ `COUNT()`
- ▶ `AVG()` (mean)
- ▶ `SUM()`
- ▶ `MIN()`
- ▶ `MAX()`

Summaries and Computations

SQL supports basic arithmetic operations to create new columns, as well as some summarization functions which include

- ▶ `COUNT()`
- ▶ `AVG()` (mean)
- ▶ `SUM()`
- ▶ `MIN()`
- ▶ `MAX()`

Since the `COUNT` for all columns is the same, the form `COUNT(*)` is often used.

Summaries and Computations

SQL supports basic arithmetic operations to create new columns, as well as some summarization functions which include

- ▶ `COUNT()`
- ▶ `AVG()` (mean)
- ▶ `SUM()`
- ▶ `MIN()`
- ▶ `MAX()`

Since the `COUNT` for all columns is the same, the form `COUNT(*)` is often used.

Other functions (`ABS()`, `FLOOR()`, `ROUND()`, `SQRT()`, etc.) may also be available.

Summary and Computation examples

Summary and Computation examples

Find max. height for age between 10 and 11 and race=1:

Summary and Computation examples

Find max. height for age between 10 and 11 and race=1:

```
SELECT MAX(height) FROM kids
```

[▶ View](#)

```
WHERE age BETWEEN 10 AND 11 AND race = 1 ;
```


Summary and Computation examples

Find max. height for age between 10 and 11 and race=1:

```
SELECT MAX(height) FROM kids
```

[▶ View](#)

```
WHERE age BETWEEN 10 AND 11 AND race = 1 ;
```

By combining with the **GROUP BY** command, useful summaries can be obtained.

Summary and Computation examples

Find max. height for age between 10 and 11 and race=1:

```
SELECT MAX(height) FROM kids
```

[▶ View](#)

```
WHERE age BETWEEN 10 AND 11 AND race = 1 ;
```

By combining with the **GROUP BY** command, useful summaries can be obtained.

Find the average BMI ($\text{weight}/\text{height}^2 * 10000$) by sex and race:

Summary and Computation examples

Find max. height for age between 10 and 11 and race=1:

```
SELECT MAX(height) FROM kids
```

[▶ View](#)

```
WHERE age BETWEEN 10 AND 11 AND race = 1 ;
```

By combining with the GROUP BY command, useful summaries can be obtained.

Find the average BMI ($\text{weight}/\text{height}^2 * 10000$) by sex and race:

```
SELECT sex,race,count(*) AS n,
```

[▶ View](#)

```
AVG(weight/(height*height)*10000) AS bmi
```

```
FROM kids GROUP BY sex,race;
```

Summary and Computation examples

Find max. height for age between 10 and 11 and race=1:

```
SELECT MAX(height) FROM kids
```

[▶ View](#)

```
WHERE age BETWEEN 10 AND 11 AND race = 1 ;
```

By combining with the GROUP BY command, useful summaries can be obtained.

Find the average BMI ($\text{weight}/\text{height}^2 * 10000$) by sex and race:

```
SELECT sex,race,count(*) AS n,
```

[▶ View](#)

```
    AVG(weight/(height*height)*10000) AS bmi
```

```
FROM kids GROUP BY sex,race;
```

The SUM function can count logical expressions:

Summary and Computation examples

Find max. height for age between 10 and 11 and race=1:

```
SELECT MAX(height) FROM kids
```

[▶ View](#)

```
WHERE age BETWEEN 10 AND 11 AND race = 1 ;
```

By combining with the GROUP BY command, useful summaries can be obtained.

Find the average BMI ($\text{weight}/\text{height}^2 * 10000$) by sex and race:

```
SELECT sex,race,count(*) AS n,
```

[▶ View](#)

```
    AVG(weight/(height*height)*10000) AS bmi
```

```
FROM kids GROUP BY sex,race;
```

The SUM function can count logical expressions:

```
SELECT race,SUM(height > 150)/COUNT(*)
```

[▶ View](#)

```
FROM kids GROUP BY race;
```

Selecting based on Summaries

Summaries can't be used in the **WHERE** clause, but they can be used in the **HAVING** clause. For example, suppose we wanted to find all the **IDs** in the **kids** database for which there were less than 2 observations:

Selecting based on Summaries

Summaries can't be used in the **WHERE** clause, but they can be used in the **HAVING** clause. For example, suppose we wanted to find all the IDs in the **kids** database for which there were less than 2 observations:

```
SELECT id FROM kids
      GROUP BY id HAVING COUNT(*) < 2;
```

► View

Selecting based on Summaries

Summaries can't be used in the **WHERE** clause, but they can be used in the **HAVING** clause. For example, suppose we wanted to find all the IDs in the **kids** database for which there were less than 2 observations:

```
SELECT id FROM kids
        GROUP BY id HAVING COUNT(*) < 2;
```

[▶ View](#)

Get all information about ids that have exactly ten observations:

Selecting based on Summaries

Summaries can't be used in the **WHERE** clause, but they can be used in the **HAVING** clause. For example, suppose we wanted to find all the IDs in the **kids** database for which there were less than 2 observations:

```
SELECT id FROM kids
        GROUP BY id HAVING COUNT(*) < 2;
```

► View

Get all information about ids that have exactly ten observations:

```
SELECT * FROM kids
        GROUP BY id HAVING COUNT(*) = 10;
```

► View

Selecting based on Summaries

Summaries can't be used in the **WHERE** clause, but they can be used in the **HAVING** clause. For example, suppose we wanted to find all the IDs in the **kids** database for which there were less than 2 observations:

```
SELECT id FROM kids
        GROUP BY id HAVING COUNT(*) < 2;
```

[▶ View](#)

Get all information about ids that have exactly ten observations:

```
SELECT * FROM kids
        GROUP BY id HAVING COUNT(*) = 10;
```

[▶ View](#)

This doesn't work - it only gives the first observation for each id.

Subqueries

By putting a **SELECT** statement in parentheses, you can use it in other **SELECT** statements as if it were another table.

Subqueries

By putting a **SELECT** statement in parentheses, you can use it in other **SELECT** statements as if it were another table.

```
SELECT * FROM kids
      WHERE id IN
      (SELECT id FROM kids
      GROUP BY id
      HAVING COUNT(*) = 10);
```

► View

Subqueries

By putting a **SELECT** statement in parentheses, you can use it in other **SELECT** statements as if it were another table.

```
SELECT * FROM kids
  WHERE id IN
    (SELECT id FROM kids
     GROUP BY id
     HAVING COUNT(*) = 10);
```

► View

This may be slow if the number of ids is large.

Subqueries

By putting a **SELECT** statement in parentheses, you can use it in other **SELECT** statements as if it were another table.

```
SELECT * FROM kids
  WHERE id IN
    (SELECT id FROM kids
     GROUP BY id
     HAVING COUNT(*) = 10);
```

► View

This may be slow if the number of ids is large.

A more efficient way is to use the subquery in an inner join (discussed later):

Subqueries

By putting a **SELECT** statement in parentheses, you can use it in other **SELECT** statements as if it were another table.

```
SELECT * FROM kids
      WHERE id IN
      (SELECT id FROM kids
       GROUP BY id
       HAVING COUNT(*) = 10);
```

[► View](#)

This may be slow if the number of ids is large.

A more efficient way is to use the subquery in an inner join (discussed later):

```
SELECT * FROM kids
      INNER JOIN
      (SELECT id FROM kids
       GROUP BY id
       HAVING COUNT(*) = 10) AS t USING(id);
```

[► View](#)

Subqueries

By putting a **SELECT** statement in parentheses, you can use it in other **SELECT** statements as if it were another table.

```
SELECT * FROM kids
      WHERE id IN
      (SELECT id FROM kids
       GROUP BY id
       HAVING COUNT(*) = 10);
```

► View

This may be slow if the number of ids is large.

A more efficient way is to use the subquery in an inner join (discussed later):

```
SELECT * FROM kids
      INNER JOIN
      (SELECT id FROM kids
       GROUP BY id
       HAVING COUNT(*) = 10) AS t USING(id);
```

► View

This is considerably faster than the previous query.

Subqueries (cont'd)

Suppose we want to find all information about the observation with maximum weight:

Subqueries (cont'd)

Suppose we want to find all information about the observation with maximum weight:

```
SELECT * FROM kids  
        HAVING weight = MAX(weight);
```

► View

Subqueries (cont'd)

Suppose we want to find all information about the observation with maximum weight:

```
SELECT * FROM kids  
        HAVING weight = MAX(weight);
```

It returns an empty set!

► View

Subqueries (cont'd)

Suppose we want to find all information about the observation with maximum weight:

```
SELECT * FROM kids  
        HAVING weight = MAX(weight);
```

► View

It returns an empty set!

Subqueries can be used to find the correct information:

Subqueries (cont'd)

Suppose we want to find all information about the observation with maximum weight:

```
SELECT * FROM kids  
        HAVING weight = MAX(weight);
```

[▶ View](#)

It returns an empty set!

Subqueries can be used to find the correct information:

```
SELECT * FROM kids  
        WHERE weight =  
        (SELECT MAX(weight) FROM kids);
```

[▶ View](#)

Subqueries (cont'd)

Suppose we want to find all information about the observation with maximum weight:

```
SELECT * FROM kids  
      HAVING weight = MAX(weight);
```

[▶ View](#)

It returns an empty set!

Subqueries can be used to find the correct information:

```
SELECT * FROM kids  
      WHERE weight =  
            (SELECT MAX(weight) FROM kids);
```

[▶ View](#)

A similar thing can be done when there are grouping variables:

Subqueries (cont'd)

Suppose we want to find all information about the observation with maximum weight:

```
SELECT * FROM kids  
      HAVING weight = MAX(weight);
```

[▶ View](#)

It returns an empty set!

Subqueries can be used to find the correct information:

```
SELECT * FROM kids  
      WHERE weight =  
      (SELECT MAX(weight) FROM kids);
```

[▶ View](#)

A similar thing can be done when there are grouping variables:

```
SELECT k.id,k.sex,k.race,k.age,  
       k.weight,k.height FROM kids AS k,  
       (SELECT sex,race,max(weight) AS weight from  
        kids) AS m WHERE k.sex=m.sex AND  
       k.race=m.race AND k.weight=m.weight;
```

[▶ View](#)

Making Tables from Queries

Sometimes it is useful to store a table which results from a query.

Making Tables from Queries

Sometimes it is useful to store a table which results from a query.

Suppose we want to create a table with only observations with age less than 15.

Making Tables from Queries

Sometimes it is useful to store a table which results from a query.

Suppose we want to create a table with only observations with age less than 15.

```
CREATE TABLE young LIKE kids;  
INSERT INTO young SELECT * FROM kids  
WHERE age < 15;
```

Making Tables from Queries

Sometimes it is useful to store a table which results from a query.

Suppose we want to create a table with only observations with age less than 15.

```
CREATE TABLE young LIKE kids;  
INSERT INTO young SELECT * FROM kids  
WHERE age < 15;
```

Such a table will stay on the database – to create a temporary one:

Making Tables from Queries

Sometimes it is useful to store a table which results from a query.

Suppose we want to create a table with only observations with age less than 15.

```
CREATE TABLE young LIKE kids;  
INSERT INTO young SELECT * FROM kids  
    WHERE age < 15;
```

Such a table will stay on the database – to create a temporary one:

```
CREATE TEMPORARY TABLE young LIKE kids;
```

Making Tables from Queries

Sometimes it is useful to store a table which results from a query.

Suppose we want to create a table with only observations with age less than 15.

```
CREATE TABLE young LIKE kids;  
INSERT INTO young SELECT * FROM kids  
WHERE age < 15;
```

Such a table will stay on the database – to create a temporary one:

```
CREATE TEMPORARY TABLE young LIKE kids;
```

Alternatively, you can DROP the table when you're done:

Making Tables from Queries

Sometimes it is useful to store a table which results from a query.

Suppose we want to create a table with only observations with age less than 15.

```
CREATE TABLE young LIKE kids;  
INSERT INTO young SELECT * FROM kids  
WHERE age < 15;
```

Such a table will stay on the database – to create a temporary one:

```
CREATE TEMPORARY TABLE young LIKE kids;
```

Alternatively, you can DROP the table when you're done:

```
DROP TABLE young;
```

Music Collection Example

Traditionally, redundancy is the enemy of database design, because it wastes storage space and increase data entry errors. For this reason, may traditional databases have a separate table for each attribute of importance. For example, suppose we have a collection of songs, organized into albums. Rather than store each song as a row with the album title and artist, we would create three tables: one for songs(tracks), one for albums, and one for artists.

Music Collection Example

Traditionally, redundancy is the enemy of database design, because it wastes storage space and increase data entry errors. For this reason, may traditional databases have a separate table for each attribute of importance. For example, suppose we have a collection of songs, organized into albums. Rather than store each song as a row with the album title and artist, we would create three tables: one for songs(tracks), one for albums, and one for artists.

Album		Artist		Track	
alid	INT	aid	INT	tid	INT
aid	INT	name	VARCHAR(40)	alid	INT
title	VARCHAR(60)			time	INT
				title	VARCHAR(40)
				filename	VARCHAR(14)

A Look at the Tables

```
mysql> select * from album limit 1,5;
```

+-----+-----+-----+-----+			
	alid		aid title
+-----+-----+-----+-----+			
	140		102 Ugetsu
	150		109 Born To Be Blue
	151		109 Connecticut Jazz Party
	152		109 Easy Does It
	153		109 In Person
+-----+-----+-----+-----+			

5 rows in set (0.03 sec)

```
mysql> select * from artist limit 1,5;
```

+-----+-----+-----+		
	aid	name
+-----+-----+-----+		
	109	Bobby Timmons
	134	Dizzy Gillespie
	140	Elmo Hope
	146	Erroll Garner
	159	Horace Silver
+-----+-----+-----+		

5 rows in set (0.03 sec)

```
mysql> select * from track limit 1,5;
```

+-----+-----+-----+-----+-----+-----+					
	tid		alid		time title filename
+-----+-----+-----+-----+-----+-----+					
	1713		139		413 Sincerely Diane (alternate take) 1077698286.mp3
	1714		139		384 Yama 1077698288.mp3
	1715		139		404 When your lover has gone 1077698290.mp3
	2276		139		398 So tired 1077699502.mp3
	3669		139		408 Sincerely Diana 1077702347.mp3
+-----+-----+-----+-----+-----+-----+					

5 rows in set (0.03 sec)

SELECT with multiple tables

Produce a list of album titles along with artist:

SELECT with multiple tables

Produce a list of album titles along with artist:

```
SELECT a.title,r.name  
      FROM album AS a, artist AS r  
      WHERE a.aid = r.aid;
```

► View

SELECT with multiple tables

Produce a list of album titles along with artist:

```
SELECT a.title,r.name  
      FROM album AS a, artist AS r  
      WHERE a.aid = r.aid;
```

► View

This is a common operation, known as an *inner join*:

SELECT with multiple tables

Produce a list of album titles along with artist:

```
SELECT a.title,r.name  
      FROM album AS a, artist AS r  
      WHERE a.aid = r.aid;
```

► View

This is a common operation, known as an *inner join*:

```
SELECT a.title,r.name FROM album AS a  
      INNER JOIN artist AS r USING(aid);
```

SELECT with multiple tables

Produce a list of album titles along with artist:

```
SELECT a.title,r.name
      FROM album AS a, artist AS r
      WHERE a.aid = r.aid;
```

► View

This is a common operation, known as an *inner join*:

```
SELECT a.title,r.name FROM album AS a
      INNER JOIN artist AS r USING(aid);
```

This produces the same result as the previous query.

SELECT with multiple tables

Produce a list of album titles along with artist:

```
SELECT a.title,r.name
      FROM album AS a, artist AS r
      WHERE a.aid = r.aid;
```

► View

This is a common operation, known as an *inner join*:

```
SELECT a.title,r.name FROM album AS a
      INNER JOIN artist AS r USING(aid);
```

This produces the same result as the previous query.

Find the sum of the times on each album:

```
SELECT SUM(time) as duration
      FROM track GROUP BY alid
      ORDER BY duration DESC;
```

► View

SELECT with multiple tables

Produce a list of album titles along with artist:

```
SELECT a.title,r.name
      FROM album AS a, artist AS r
      WHERE a.aid = r.aid;
```

► View

This is a common operation, known as an *inner join*:

```
SELECT a.title,r.name FROM album AS a
      INNER JOIN artist AS r USING(aid);
```

This produces the same result as the previous query.

Find the sum of the times on each album:

```
SELECT SUM(time) as duration
      FROM track GROUP BY alid
      ORDER BY duration DESC;
```

► View

Unfortunately, all we have are the album ids, not the names

SELECT with multiple tables(cont'd)

To improve our previous example, we need to combine the track information with album and artist information. Suppose we want to find the 10 longest albums in the collection:

SELECT with multiple tables(cont'd)

To improve our previous example, we need to combine the track information with album and artist information. Suppose we want to find the 10 longest albums in the collection:

```
SELECT a.title,r.name,  
       SUM(time) AS duration  
FROM track AS t, album as a, artist as r  
WHERE t.alid = a.alid AND a.aid = r.aid  
GROUP BY t.alid ORDER BY duration DESC  
LIMIT 1,10;
```

► View

SELECT with multiple tables(cont'd)

To improve our previous example, we need to combine the track information with album and artist information. Suppose we want to find the 10 longest albums in the collection:

```
SELECT a.title,r.name,  
       SUM(time) AS duration  
FROM track AS t, album as a, artist as r  
WHERE t.alid = a.alid AND a.aid = r.aid  
GROUP BY t.alid ORDER BY duration DESC  
LIMIT 1,10;
```

► View

The Rules Have Changed

As powerful as SQL is, we can use it as a data store without having to use all of the SQL features.

The Rules Have Changed

As powerful as SQL is, we can use it as a data store without having to use all of the SQL features.

- ▶ Don't hesitate to use familiar programs to do the hard work

The Rules Have Changed

As powerful as SQL is, we can use it as a data store without having to use all of the SQL features.

- ▶ Don't hesitate to use familiar programs to do the hard work
- ▶ Repeated **SELECT** queries in loops can do wonders

The Rules Have Changed

As powerful as SQL is, we can use it as a data store without having to use all of the SQL features.

- ▶ Don't hesitate to use familiar programs to do the hard work
- ▶ Repeated **SELECT** queries in loops can do wonders
- ▶ Load up data structures with entire tables

The Rules Have Changed

As powerful as SQL is, we can use it as a data store without having to use all of the SQL features.

- ▶ Don't hesitate to use familiar programs to do the hard work
- ▶ Repeated **SELECT** queries in loops can do wonders
- ▶ Load up data structures with entire tables
- ▶ Use as little or as much pure SQL as you like

The Rules Have Changed

As powerful as SQL is, we can use it as a data store without having to use all of the SQL features.

- ▶ Don't hesitate to use familiar programs to do the hard work
- ▶ Repeated **SELECT** queries in loops can do wonders
- ▶ Load up data structures with entire tables
- ▶ Use as little or as much pure SQL as you like

These ideas are illustrated using the music collection data, R, python, and perl

Using SQL in R

```
library(RMySQL)
drv = dbDriver("MySQL")
con = dbConnect(drv,dbname="dbname",user="user",pass="pass")
rs = dbSendQuery(con,statement="select * from album")
album = fetch(rs,n=-1)
rs = dbSendQuery(con,statement="select * from track")
track = fetch(rs,n=-1)
rs = dbSendQuery(con,statement="select * from artist")
artist = fetch(rs,n=-1)

tracks = data.frame(
  album = factor(track$alid,levels=album$alid,
                 labels=album$title),
  artist = factor(merge(track[, "alid", drop=FALSE],
                        album[, c("alid", "aid")], by="alid")$aid,
                 levels=artist$aid,
                 labels=artist$name),
  time = track$time)

res = aggregate(tracks$time,
                list(album=tracks$album,artist=tracks$artist),sum)
res = res[order(res$x,decreasing=TRUE),]
print(res[1:10,])
```

Using SQL in python

```
#!/usr/bin/python

from MySQLdb import *

con = connect(user='user',passwd='pass',db='dbname')
cursor = con.cursor()
cursor.execute('select * from track')
tracks = cursor.fetchall()

durations = {}
for t in tracks:
    durations[t[1]] = durations.get(t[1],0) + t[2]

alids = durations.keys()
alids.sort(lambda x,y:cmp(durations[y],durations[x]))

for i in range(10):
    cursor.execute(
        'select title,aid from album where alid = %d' % alids[i])
    title,aid = cursor.fetchall()[0]
    cursor.execute('select name from artist where aid = %d' % aid)
    name = cursor.fetchall()[0][0]
    print '%s\t%s\t%d' % (title,name,durations[alids[i]])
```

Using SQL in perl

```
#!/usr/bin/perl
use DBI;
$dbh = DBI->connect('DBI:mysql:dbname:localhost','user','pass');

$sth = $dbh->prepare('select * from album');
$sth->execute();
while((@row) = $sth->fetchrow()){
    $album{$row[0]} = $row[2];
    $artist{$row[0]} = $row[1];
}

$sth = $dbh->prepare('select * from artist');
$sth->execute();
$artist{$row[0]} = $row[1] while((@row) = $sth->fetchrow());

$sth = $dbh->prepare('select * from track');
$sth->execute();
$duration{$row[1]} += $row[2] while((@row) = $sth->fetchrow());

@salbum = sort({$duration{$b} <=> $duration{$a}} keys(%duration));
foreach $i (0..9){
    print
        "$album{$salbum[$i]}\t$artist{$artist{$salbum[$i]}}\t",
        "$duration{$salbum[$i]}\n"
}
```

```
mysql> select * from kids;
```

id	race	age	height	weight	sex
100011	2	10.346	148.500	38.950	1
100011	2	11.282	157.100	44.100	1
100011	2	14.428	165.950	57.800	1
100011	2	15.321	167.050	59.650	1
100031	1	10.920	158.000	63.700	1
100031	1	11.917	161.000	68.500	1
100031	1	13.007	162.750	85.950	1
.					
308091	1	9.460	138.000	39.000	1
308091	1	10.740	147.500	53.100	1
308091	1	11.359	151.750	57.050	1
308101	1	9.800	152.350	38.500	2
308101	1	10.781	159.335	48.235	2
308101	1	11.701	164.285	51.700	2

20704 rows in set (0.18 sec)

```
mysql> select age,race,height,weight from kids
      -> where weight > 80 and height < 150;
```

+-----+-----+-----+-----+			
age	race	height	weight
+-----+-----+-----+-----+			
12.429	2	147.800	83.000
11.674	2	149.350	82.950
14.414	2	149.300	86.750
+-----+-----+-----+-----+			

3 rows in set (0.06 sec)

```
mysql> select * from kids order by height desc;
```

id	race	age	height	weight	sex
302941	2	19.657	201.905	83.820	2
300861	2	17.804	201.850	126.610	2
302941	2	16.572	201.795	76.670	2
300861	2	14.833	201.520	124.245	2
300861	2	18.781	201.520	123.310	2
302941	2	18.611	201.410	83.710	2
107061	2	17.626	201.300	82.005	2
302941	2	15.537	201.190	72.820	2
304441	1	17.946	201.190	67.430	2
116741	1	17.338	201.025	72.710	2

10 rows in set (0.10 sec)

```
mysql> select * from kids
      -> where age between 17 and 18
      -> and weight between 180 and 185;
```

id	race	age	height	weight	sex
304741	1	17.875	194.150	184.250	2

1 row in set (0.03 sec)


```
mysql> select max(height) from kids
      ->      where age between 10 and 11 and race = 1;
+-----+
| max(height) |
+-----+
|      178.750 |
+-----+
1 row in set (0.06 sec)
```

```
mysql> select sex,race,count(*) as n,  
        -> avg(weight/(height*height)*10000) as bmi  
        -> from kids group by sex,race;
```

+-----+-----+-----+-----+				
sex	race	n	bmi	
+-----+-----+-----+-----+				
1	1	4977	21.312670406	
1	2	5532	23.489962065	
2	1	4973	19.153469602	
2	2	5222	21.040500147	
+-----+-----+-----+-----+				

4 rows in set (0.12 sec)

```
mysql> select race,sum(height > 150)/count(*)
      -> from kids group by race;
```

race	sum(height > 150)/count(*)
1	0.85
2	0.89

2 rows in set (0.05 sec)

```
mysql> select id from kids
      ->      group by id having count(*) < 2;
+-----+
| id      |
+-----+
| 101051  |
| 103181  |
| 103191  |
| 107231  |
| 109001  |
| . . .  |
| 207291  |
| 207961  |
| 302241  |
| 304561  |
| 307081  |
+-----+
22 rows in set (0.10 sec)
```

```
mysql> select * from kids group by id having count(*)=10;
+-----+-----+-----+-----+-----+-----+
| id      | race | age      | height  | weight  | sex |
+-----+-----+-----+-----+-----+-----+
| 100031 | 1 | 10.920 | 158.000 | 63.700 | 1 |
| 100041 | 1 | 10.070 | 159.500 | 51.700 | 2 |
| 100071 | 2 | 10.630 | 139.700 | 37.500 | 1 |
| 100081 | 2 | 9.110 | 152.130 | 36.795 | 2 |
| 100091 | 2 | 9.200 | 148.250 | 54.150 | 1 |
|      |      |      |      |      |      |
| 308021 | 1 | 9.330 | 157.850 | 41.470 | 2 |
| 308041 | 1 | 10.810 | 157.025 | 38.060 | 2 |
| 308061 | 1 | 10.120 | 156.200 | 32.780 | 2 |
| 308071 | 1 | 10.990 | 138.500 | 29.450 | 1 |
| 308081 | 1 | 9.920 | 152.900 | 31.130 | 2 |
+-----+-----+-----+-----+-----+-----+
1303 rows in set (0.11 sec)
```

```
mysql> select * from kids where id in
->      (select id from kids group by id
->      having count(*)=10);
```

+-----+-----+-----+-----+-----+-----+						
id	race	age	height	weight	sex	
+-----+-----+-----+-----+-----+-----+						
100011	2	10.346	148.500	38.950	1	
100011	2	11.282	157.100	44.100	1	
100011	2	12.336	163.900	51.150	1	
100011	2	13.388	166.450	57.400	1	
100011	2	14.428	165.950	57.800	1	
.						
308081	1	14.803	183.700	55.935	2	
308081	1	15.780	183.590	54.780	2	
308081	1	16.865	184.195	58.905	2	
308081	1	17.864	184.580	56.320	2	
308081	1	18.631	184.195	56.100	2	
+-----+-----+-----+-----+-----+-----+						
13030 rows in set (35 min 33.96 sec)						

```
mysql> select * from kids inner join
-> (select id from kids group by id having count(*)=10)
-> as a using(id);
```

id	race	age	height	weight	sex
100011	2	10.346	148.500	38.950	1
100011	2	11.282	157.100	44.100	1
100011	2	12.336	163.900	51.150	1
100011	2	13.388	166.450	57.400	1
100011	2	14.428	165.950	57.800	1
.					
308081	1	14.803	183.700	55.935	2
308081	1	15.780	183.590	54.780	2
308081	1	16.865	184.195	58.905	2
308081	1	17.864	184.580	56.320	2
308081	1	18.631	184.195	56.100	2

13030 rows in set (11.89 sec)

```
mysql> select * from kids  
      -> having weight = max(weight);  
Empty set (0.00 sec)
```

◀ Return


```
mysql> select * from kids
      -> where weight = (select max(weight) from kids);
+-----+-----+-----+-----+-----+-----+
| id      | race | age      | height | weight | sex |
+-----+-----+-----+-----+-----+-----+
| 304741  | 1    | 18.680   | 192.940 | 189.695 | 2   |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.03 sec)
```

```
mysql> select k.id,k.sex,k.race,k.age,k.weight,k.height
-> from kids as k, (select sex,race,max(weight) as weight
-> from kids group by sex,race) as m
-> where k.sex = m.sex and k.race = m.race and
-> k.weight = m.weight;
```

id	sex	race	age	weight	height
207201	2	2	19.405	173.360	191.565
207931	1	2	19.674	151.200	164.900
208171	1	1	18.633	128.500	168.100
304741	2	1	18.680	189.695	192.940

4 rows in set (0.34 sec)

```
mysql> select a.title,r.name from album as a,artist as r where a.aid = r.aid;
```

title	name
A Night in Tunisia	Art Blakey & Jazz Messengers
Ugetsu	Art Blakey & Jazz Messengers
Born To Be Blue	Bobby Timmons
Connecticut Jazz Party	Bobby Timmons
Easy Does It	Bobby Timmons
In Person	Bobby Timmons
Moanin' Blues	Bobby Timmons
The Prestige Trio Sessions	Bobby Timmons
Soul Man Soul Food	Bobby Timmons
Soul Time	Bobby Timmons
Workin' Out	Bobby Timmons
1945-1950 Small Groups	Dizzy Gillespie
.	
Live at the Circle Room and Mo	Nat King Cole
Birth of the Cole 1938-1939	Nat King Cole
Rockin' Boppin' & Blues	Nat King Cole
WWII Transcriptions	Nat King Cole
Oscar Peterson And Clark Terry	Oscar Peterson
A Tribute To My Friends	Oscar Peterson
The Oscar Peterson Trio Live At Zardi's - Disc One	Oscar Peterson
The Oscar Peterson Trio Live At Zardi's - Disc Two	Oscar Peterson
Skol	Oscar Peterson
Oscar Peterson and Dizzy Gillespie	Oscar Peterson
Overseas	Tommy Flanagan
The Tommy Flanagan Trio	Tommy Flanagan
Trio & Sextet	Tommy Flanagan

72 rows in set (0.02 sec)

```
mysql> select alid,sum(time) as duration
      -> from track group by alid order by duration desc;
+-----+-----+
| alid | duration |
+-----+-----+
| 150  | 6057    |
| 286  | 5664    |
| 264  | 5028    |
| 156  | 4764    |
| 158  | 4674    |
|      | . . .   |
| 343  | 2031    |
| 263  | 1865    |
| 281  | 1749    |
| 280  | 1611    |
| 287  | 1519    |
| 203  | 1061    |
+-----+-----+
72 rows in set (0.04 sec)
```

```
mysql> select a.title,r.name,sum(time) as duration
-> from track as t,album as a,artist as r
-> where t.alid=a.alid and a.aid = r.aid
-> group by t.alid
-> order by duration desc limit 1,10;
```

title	name	duration
My Funny Valentine	Miles Davis	5664
Trio	Kenny Drew	5028
Soul Man Soul Food	Bobby Timmons	4764
Workin' Out	Bobby Timmons	4674
The All-Stars Sessions	Elmo Hope	4636
The Oscar Peterson Trio Live At Zardi's - Disc Two	Oscar Peterson	4567
Memories Of You	Erroll Garner	4538
Elmo Hope	Elmo Hope	4536
WWII Transcriptions	Nat King Cole	4456
The Oscar Peterson Trio Live At Zardi's - Disc One	Oscar Peterson	4355

10 rows in set (0.10 sec)