Announcements

**DATABASE NORMALIZATION WEBINAR** - Interested in viewing a recording of a webinar that explores normalization concepts? If so, it is located here:  [Normalization Webinar](http://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=2826a49c-cd20-493a-ab1f-20684316792d).

​**DATA MODELING WEBINAR** - Interested in viewing a recording of a webinar that explores data modeling concepts. It is located here:  [Data Modeling Webinar](https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=bae03e4f-4a5f-44ca-ae59-ca2bf66ce2f1). While it talks about the Data Management Foundations concepts, its concepts are also very application to Data Management Applications as well.

**LINKED IN LEARNING VIDEO COURSES** - These [*Linked In Learning video courses*](https://www.linkedin.com/learning/mysql-essential-training-2/harness-the-power-of-mysql?u=2045532) that may help reinforce the concepts you are reading about in the course textbook. Remember, they are supplemental in nature and NOT intended to replace the course materials.

**IF YOU ARE REQUIRED TO COMPLETE EHN1 IN YOUR VERSION OF THE COURSE**, please keep in mind that since it is a performance assessment, instructors are limited in the amount of input they can provide.  However, while we are unable to provide direct feedback about what you have put together thus far, we can provide you with some general information about some of the frequent trouble spots that many students encounter and suggestions on how to avoid them.  To that end, we recommend that you take a look at the [**normalization case study**](https://srm--c.na13.visual.force.com/apex/coursearticle?Id=kA0a0000000PKmsCAG) , which provides an excellent example of the normalization process and what Evaluation will be looking for in Part A of the performance assessment. Watching the [Entity Relationship Modeling recording](https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=6f89341e-ad4c-48c7-b836-adff017c1834)may also be helpful as you work to complete Part A.

\*\*\*Please note that if you are required to take the ZLO1 assessment, this information does NOT apply to you.

**SUPPLEMENTAL READING:** Because portions of both assessments for this course will measure your knowledge of data modelling and normalization concepts (these concepts are discussed in the supplemental required reading for Data Management Foundations), if you are not as familiar with those concepts, please study chapters two and seven of the supplemental online textbook (and "The Index" section of chapter 8). Here is link to that resource: [Fundamentals of Database Management Systems](https://lrps.wgu.edu/provision/71492681).

If I didn't find this resource, I wouldn't have passed the exam. This site was the best SQL programming teacher and simulator that I found. Invaluable. I told WGU they should tell more students about it. screenshot from datacamp.com posted

This guide will work on assumption that you passed D426 and know a little SQL from it.

***Study Material -*** ZYBooks, ZYBooks, and more ZYBooks. I know I'm sorry, but it's going to give you exactly what you need to pass and nothing else. Plus the test itself is done in ZYBooks so you're going to have to get used to how it responds.

Here is how I used it, I slogged my way through chapter 1 and 2 to brush up from D426. Then I started actually studying for the test. I did the green labs in chapters 2, 7 , and 8 about four times then chapters 7 and 8 five more times. I basically did the labs until I could do them by heart and could correlate what they were asking from me and what SQL statements should use with proper syntax.

This is the only class I've taken that I would recommend taking the PA enough to memorize it as the OA is absurdly close.

**Reference Material** - I used W3 for my syntax reference material for MYSQL statements while doing the labs until I was comfortable without it. This helped a lot as they just put the exact material on the left, where with ZYBooks you have to bounce around multiple pages to try and find the exact statement you're trying to write.

**Dissecting The ERROR Code -** You will notice sometimes when you submit a statement in ZYBooks it'll say something like SYNTAX ERROR blah blah look at MYSQL something. Most of it is unhelpful, scroll to the far right in the box (most of the time) and you will see it says something like REFERENCES something(something) or ID INT, this almost always means your syntax error is right before what this says. It is usually something like you forgot a , or you forgot to enclose (something) or other little mistake, but its almost always right before what it says the error is.

**How To Check Your Answer -** When you're doing the PA get used to checking your answers before you submit as the OA usually wont tell you if you did it correctly other than ERROR codes. I will try to break this down as best as I can.

* CHECK YOU SPELLING - Make sure you spelling and CAPS are correct, make sure that you did leave out an s at the end of something make sure its FirstName not firstname, it will kick back as wrong. Type things out exactly how they want them written ZyBooks hates you and has no sympathy if you type Id instead of ID or FirstNames instead of FirstName.
* Writing a SELECT - When asked to write a SELECT statement you just need to be sure what they ask for is displayed in the order they wanted it displayed, these questions are the easiest to check.
* INSERT INTO - INSERT INTO and any other statements where you are adding data to the tables can be checked by putting a SELECT \* FROM TableName; after the close; of your first statement. Verify that the data was added, then delete the SELECT statement, run it again to make sure and then submit.
* CREATE TABLE - CREATE TABLE or adding columns without data are the hardest to check because you haven't been taught how in this class or the last class. A SELECT statement will return nothing, This is where you learn the DESCRIBE statement. After you close your statement ; write a second statement DESCRIBE TableName; You will see the columns you created and constraints assigned to them EX. INT, VARCHAR, CHAR.
* PRIMARY AND FOREIGN KEY - To make sure you've assigned the key correctly you can use the DESCRIBE statement just like before, I think it say PRI for PRIMARY and MUL for FOREIGN. This is to make sure you've assigned the key to the correct table/column I used it mostly for FOREIGN KEY as I had a tendency to reverse it somehow.

I started this class in earnest on Sunday morning and tested out Tuesday night using this method, I found the biggest help was when I discovered the DESCRIBE statement as I was able to verify my statements worked as intended before submitting. I sat on this class for weeks before actually giving it a go because D426 burnt me out so hard but this class is so much better as I feel like I actually did learn something from it.

1.1 Relational model

**Database models**

A ***database model*** is a conceptual framework for database systems, with three parts:

* **Data structures** that prescribe how data is organized.
* **Operations** that manipulate data structures.
* **Rules** that govern valid data.
* Table 1.1.1: Example database models.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Primary data structure | Initial product releases | Example database system | Strengths |
| **Hierarchical** | Tree | 1960s | IMS | Fast queries Efficient storage |
| **Network** | Linked list | 1970s | IDMS | Fast queries Efficient storage |
| **Relational** | Table | 1980s | Oracle Database | Productivity and simplicity Transactional applications |
| **Object** | Class | 1990s | ObjectStore | Integration with object-oriented programming languages |
| **Graph** | Vertex and edge | 2000s | Neo4j | Flexible schema Evolving business requirements |
| **Document** | XML JSON | 2010s | MongoDB | Flexible schema Unstructured and semi-structured data |

**Relational data structure**

The relational data structure is based on set theory. A ***set*** is an unordered collection of elements enclosed in braces. Ex: **{a, b, c}** and **{c, b, a}** are the same, since sets are not ordered. A ***tuple*** is an ordered collection of elements enclosed in parentheses. Ex: **(a, b, c)** and **(c, b, a)** are different, since tuples are ordered.

The data structure organizes data in tables:

* A ***table*** has a name, a fixed tuple of columns, and a varying set of rows.
* A ***column*** has a name and a data type.
* A ***row*** is an unnamed tuple of values. Each value corresponds to a column and belongs to the column's data type.
* A ***data type*** is a named set of values, from which column values are drawn.

Table 1.1.2: Similar data structure terms.

|  |  |  |
| --- | --- | --- |
| Databases | Mathematics | Files |
| Table | Relation | File |
| Column | Attribute | Field |
| Row | Tuple | Record |
| Data type | Domain | Data type |

**Relational operations**

Like the relational data structure, relational operations are based on set theory. Each operation generates a result table from one or two input tables:

* **Select** selects a subset of rows of a table.
* **Project** eliminates one or more columns of a table.
* **Product** lists all combinations of rows of two tables.
* **Join** combines two tables by comparing related columns.
* **Union** selects all rows of two tables.
* **Intersect** selects rows common to two tables.
* **Difference** selects rows that appear in one table but not another.
* **Rename** changes a table name.
* **Aggregate** computes functions over multiple table rows, such as sum and count.

These operations are collectively called ***relational algebra*** and are the theoretical foundation of the SQL language. Since the result of relational operations is always a table, the result of an SQL query is also a table.

**Relational rules**

Rules are logical constraints that ensure data is valid.

***Relational rules*** are part of the relational model and govern data in every relational database. Ex:

* **Unique primary key**. All tables have a primary key column, or group of columns, in which values may not repeat.
* **Unique column names**. Different columns of the same table have different names.
* **No duplicate rows**. No two rows of the same table have identical values in all columns.

***Business rules*** are based on business policy and specific to a particular database. Ex: All rows of the Employee table must have a valid entry in the DepartCode column. Ex: PassportNumber values may not repeat in different Employee rows.

Relational rules are implemented as SQL ***constraints*** and enforced by the database system. Business rules are discovered during database design and, like relational rules, often implemented as SQL constraints. However, some complex business rules must be enforced by applications running on the database.

The term relation, commonly used in academic literature, is equivalent to a database table. The term tuple, commonly used in academic literature, is equivalent to a database row. The term data type is used in both file and database processing. The term field, commonly used in file processing, is similar to a database column.

***Structured Query Language*** (***SQL***) is a high-level computer language for storing, manipulating, and retrieving data. SQL is the standard language for relational databases, and is commonly supported in non-relational databases. SQL is pronounced either 'S-Q-L' or 'seekwəl', but the preferred pronunciation is 'S-Q-L'.

**SQL syntax**

An SQL ***statement*** is a complete command composed of one or more clauses. A ***clause*** groups SQL keywords like SELECT, FROM, and WHERE with table names like City, column names like Name, and conditions like Population > 100000. *An SQL statement may be written on a single line, but good practice is to write each clause on a separate line.*

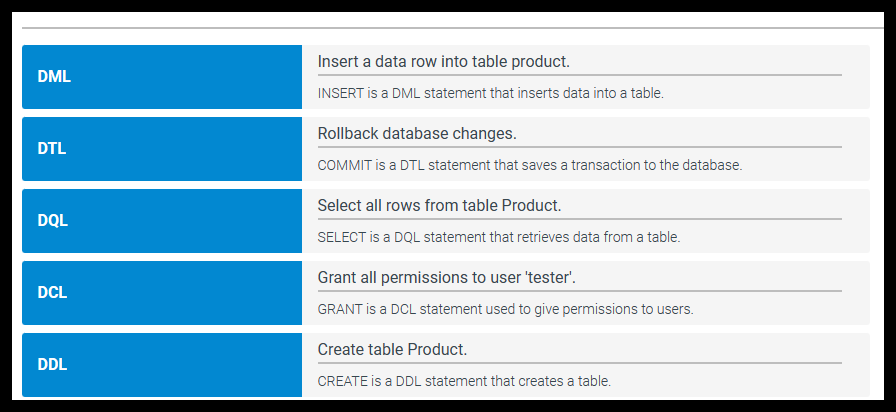
n MySQL, all SQL statements end with a semicolon. SQL keywords like SELECT, FROM, WHERE, etc. are not case sensitive. Ex: SELECT and select are equivalent. However, identifiers like column names and table names are case sensitive in many database systems. This material uses capital letters for SQL keywords so the keywords stand out from other syntactic parts. The table below summarizes various syntactic features of SQL.

|  |  |  |
| --- | --- | --- |
| Type | Description | Examples |
| Literals | Explicit values that are string, numeric, or binary. Strings must be surrounded by single quotes or double quotes. Binary values are represented with x'0' where the 0 is any hex value. | 'String'  "String"  123  x'0fa2' |
| Keywords | Words with special meaning. | SELECT, FROM, WHERE |
| Identifiers | Objects from the database like tables, columns, etc. | City, Name, Population |
| Comments | Statement intended only for humans and ignored by the database when parsing an SQL statement. | *-- single line comment*  */\* multi-line*  *Comment \*/* |

**SQL sublanguages**

The SQL language is divided into five sublanguages:

* ***Data Definition Language*** (DDL) defines the structure of the database.
* ***Data Query Language*** (DQL) retrieves data from the database.
* ***Data Manipulation Language*** (DML) manipulates data stored in a database.
* ***Data Control Language*** (DCL) controls database user access.
* ***Data Transaction Language*** (DTL) manages database transactions.



1.3 Managing databases

**CREATE DATABASE and DROP DATABASE statements**

A ***database system instance*** is a single executing copy of a database system. Personal computers usually run just one instance of a database system. Shared computers, such as computers used for cloud services, usually run multiple instances of a database system. Each instance usually contains multiple system and user databases.

Several SQL statements help database administrators, designers, and users manage the databases on an instance. ***CREATE DATABASE DatabaseName*** creates a new database. ***DROP DATABASE DatabaseName*** deletes a database, including all tables in the database.

**USE and SHOW statements**

***USE DatabaseName*** selects a default database for use in subsequent SQL statements.

Several SHOW statements provide information about databases, tables, and columns:

* ***SHOW DATABASES*** lists all databases in the database system instance.
* ***SHOW TABLES*** lists all tables in the default database.
* ***SHOW COLUMNS FROM TableName*** lists all columns in the TableName table of the default database.
* ***SHOW CREATE TABLE TableName*** shows the CREATE TABLE statement for the TableName table of the default database.

Additional SHOW statements generate information about system errors, configuration, privileges, logs, and so on

**Tables, columns, and rows**

All data in a relational database is structured in tables:

* A ***table*** has a name, a fixed sequence of columns, and a varying set of rows.
* A ***column*** has a name and a data type.
* A ***row*** is an unnamed sequence of values. Each value corresponds to a column and belongs to the column's data type.
* A ***cell*** is a single column of a single row.

A table must have at least one column but any number of rows. A table without rows is called an ***empty table***.

**Rules governing tables**

Tables must obey relational rules, including:

1. **Exactly one value per cell**. A cell may not contain multiple values. Unknown data is represented with a special NULL value.
2. **No duplicate column names**. Duplicate column names are allowed in different tables, but not in the same table.
3. **No duplicate rows**. No two rows may have identical values in all columns.
4. **No row order**. Rows are not ordered. The organization of rows on a storage device, such as a disk drive, never affects query results.

Rules 3 and 4 follow directly from the definition of a table. A table is a set of rows. Since a set's elements may not repeat and are not ordered, the same is true of a table's rows.

Rule 4 is called ***data independence***. Data independence allows database administrators to improve query performance by changing the organization of data on storage devices, without affecting query results.

**CREATE TABLE and DROP TABLE statements**

The ***CREATE TABLE*** statement creates a new table by specifying the table name, column names, and column data types. Example data types are:

* **INT or INTEGER** — integer values
* **VARCHAR(N)** — values with 0 to N characters
* **DATE** — date values
* **DECIMAL(M, D)** — numeric values with M digits, of which D digits follow the decimal point

The ***DROP TABLE*** statement deletes a table, along with all the table's rows, from a database.

Figure 1.4.1: CREATE TABLE and DROP TABLE statements.

**ALTER TABLE statement**

The ***ALTER TABLE*** statement adds, deletes, or modifies columns on an existing table. The ALTER TABLE statement specifies the table name followed by a clause that indicates what should be altered. The table below summarizes the three ALTER TABLE clauses.

Table 1.4.1: ALTER TABLE statement.

|  |  |  |
| --- | --- | --- |
| ALTER TABLE clause | Description | Syntax |
| ADD | Adds a column | ALTER TABLE TableName  ADD ColumnName DataType; |
| CHANGE | Modifies a column | ALTER TABLE TableName  CHANGE CurrentColumnName NewColumnName NewDataType; |
| DROP | Deletes a column | ALTER TABLE TableName  DROP ColumnName; |

1.5 Data types

**Data type categories**

A ***data type*** is a named set of values from which column values are drawn. In relational databases, most data types fall into one of the following categories:

* ***Integer*** data types represent positive and negative integers. Several integer data types exist, varying by the number of bytes allocated for each value. Common integer data types include INT, implemented as 4 bytes of storage, and SMALLINT, implemented as 2 bytes.
* ***Decimal*** data types represent numbers with fractional values. Decimal data types vary by number of digits after the decimal point and maximum size. Common decimal data types include FLOAT and DECIMAL.
* ***Character*** data types represent textual characters. Common character data types include CHAR, a fixed string of characters, and VARCHAR, a string of variable length up to a specified maximum size.
* ***Date and time*** data types represent date, time, or both. Some date and time data types include a time zone or specify a time interval. Some date and time data types represent an interval rather than a point in time. Common date and time data types include DATE, TIME, DATETIME, and TIMESTAMP.
* ***Binary*** data types store data exactly as the data appears in memory or computer files, bit for bit. The database manages binary data as a series of zeros and ones. Common binary data types include BLOB, BINARY, VARBINARY, and IMAGE.
* ***Spatial*** data types store geometric information, such as lines, polygons, and map coordinates. Examples include POLYGON, POINT, and GEOMETRY. Spatial data types are relatively new and consequently vary greatly across database systems.
* ***Document*** data types contain textual data in a structured format such as XML or JSON.

### MySQL data types

All relational databases support integer, decimal, date and time, and character data types. Most databases allow integer and decimal numbers to be signed or unsigned. A ***signed*** number may be negative. An ***unsigned*** number cannot be negative.

Data types vary in storage requirements. Ex:

* Character data types use one or two bytes per character.
* Integer data types use a fixed number of bytes per number.
* Unsigned data types can store larger numbers than the signed version of the same data type.

To minimize table size, the data type with the smallest storage requirements should be used. Ex: Any integer data type can store integers that range from -100 to 100. However, TINYINT requires only one byte per number and is the best choice for this range.

Common MySQL data types appear in the table below. For a complete list of MySQL data types, see the link in Exploring Further, below.

Table 1.6.1: Common operators.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Operator | Description | Example | Value |
| Arithmetic | + | Adds two numeric values | 4 + 3 | 7 |
| - (unary) | Reverses the sign of one numeric value | -(-2) | 2 |
| - (binary) | Subtracts one numeric value from another | 11 - 5 | 6 |
| \* | Multiplies two numeric values | 3 \* 5 | 15 |
| / | Divides one numeric value by another | 4 / 2 | 2 |
| % (modulo) | Divides one numeric value by another and returns the integer remainder | 5 % 2 | 1 |
| ^ | Raises one numeric value to the power of another | 5^2 | 25 |
| Comparison | = | Compares two values for equality | 1 = 2 | FALSE |
| != <> | Compares two values for inequality | 1 != 2 1 <> 2 | TRUE |
| < | Compares two values with < | 2 < 2 | FALSE |
| <= | Compares two values with ≤ | 2 <= 2 | TRUE |
| > | Compares two values with > | '2019-08-13' > '2021-08-13' | FALSE |
| >= | Compares two values with ≥ | 'apple' >= 'banana' | FALSE |
| Logical | AND | Returns TRUE only when both values are TRUE | TRUE AND FALSE | FALSE |
| OR | Returns FALSE only when both values are FALSE | TRUE OR FALSE | TRUE |
| NOT | Reverses a logical value | NOT FALSE | TRUE |

**Expressions**

An ***expression*** is a string of operators, operands, and parentheses that evaluates to a single value. Operands may be column names or fixed values. The value of an expression may be any data type. Ex: Salary > 34000 AND Department = 'Marketing' is an expression with a logical value.

A simple expression may consist of a single column name or a fixed value. Ex: The column EmployeeName and the fixed value 'Maria' are expressions with a character data type.

When an expression is evaluated, column names are replaced with column values for a specific row. Consequently, an expression containing column names may have different values for different rows.

The order of operator evaluation may affect the value of an expression. Operators in an expression are evaluated in the order of ***operator precedence***, shown in the table below. Operators of the same precedence are evaluated from left to right. Regardless of operator precedence, expressions enclosed in parentheses are evaluated before any operators outside the parentheses are applied.

Expressions may evaluate to NULL. NULL-valued expressions are discussed elsewhere in this material.

Table 1.6.2: Operator precedence.

|  |  |
| --- | --- |
| Precedence | Operators |
| 1 | - (unary) |
| 2 | ^ |
| 3 | \*     /     % |
| 4 | +     - (binary) |
| 5 | =     !=     <     >     <=     >= |
| 6 | NOT |
| 7 | AND |
| 8 | OR |

**SELECT statement**

The SELECT statement selects rows from a table. The statement has a ***SELECT*** clause and a ***FROM*** clause. The FROM clause specifies the table from which rows are selected. The SELECT clause specifies one or more expressions, separated by commas, that determine what values are returned for each row.

**WHERE clause**

An expression may return a value of any data type. A ***condition*** is an expression that evaluates to a logical value.

A SELECT statement has an optional ***WHERE*** clause that specifies a condition for selecting rows. A row is selected when the condition is TRUE for the row values. A row is omitted when the condition is either FALSE or NULL.

The WHERE clause follows the FROM clause. When a SELECT statement has no WHERE clause, all rows are selected.

1.7 Null values

**NULL**

***NULL*** is a special value that represents either unknown or inapplicable data. NULL is not the same as zero for numeric data types or blanks for character data types. Ex: A zero bonus indicates an employee can, but has not, earned a bonus. A zero bonus is known and applicable, and should not be represented as NULL.

**NOT NULL constraint**

By default, columns may contain NULL values. In some cases, however, columns should never contain NULL. Ex: If a business requires that a name is specified for all employees, the Name column of an Employee table should not contain NULL.

The ***NOT NULL*** constraint prevents a column from having a NULL value. Statements that insert NULL, or update a value to NULL, are automatically rejected. NOT NULL follows the column name and data type in a CREATE TABLE statement.

**NULL arithmetic and comparisons**

When arithmetic or comparison operators have one or more NULL operands, the result is NULL. When a WHERE clause evaluates to NULL for values in a row, the row is not selected.

**IS NULL operator**

Since comparison operators return NULL when either operand is NULL, comparison operators cannot be used to select NULL values. Ex: SELECT \* FROM Employee WHERE Salary = NULL; never returns any rows, because the WHERE clause is always NULL.

Instead, the ***IS NULL*** and ***IS NOT NULL*** operators must be used to select NULL values. Value IS NULL returns TRUE when the value is NULL. Value IS NOT NULL returns TRUE when the value is not NULL.

### NULL logic

In traditional mathematical logic, expressions are always TRUE or FALSE. When NULL is present, however, a logical expression may be either TRUE, FALSE, or NULL. NULL indicates the value of a logical expression is uncertain. Ex:

* TRUE AND TRUE is TRUE.
* TRUE AND FALSE is FALSE.
* TRUE AND NULL is NULL.

The value of logical expressions containing NULL operands is defined in ***truth tables***.

Figure 1.7.1: MySQL truth tables.

|  |  |  |  |
| --- | --- | --- | --- |
| x | y | x AND y | x OR y |
| TRUE | NULL | NULL | TRUE |
| NULL | TRUE |
| FALSE | NULL | FALSE | NULL |
| NULL | FALSE |
| NULL | NULL | NULL | NULL |
| x | NOT x |
| NULL | NULL |

1.8 Inserting, updating, and deleting rows

**INSERT statement**

The ***INSERT*** statement adds rows to a table. The INSERT statement has two clauses:

* The ***INSERT INTO*** clause names the table and columns where data is to be added. The keyword INTO is optional.
* The ***VALUES*** clause specifies the column values to be added.

The VALUES clause may list any number of rows in parentheses to insert multiple rows.

**DEFAULT values**

Columns may be omitted from an INSERT statement. When omitted, a column is assigned a NULL value. If the NOT NULL constraint is specified on the column, the insert is rejected.

Alternatively, a default value may be specified for a column. The optional ***DEFAULT*** keyword and default value follow the column name and data type in a CREATE TABLE statement. The column is assigned the default value, rather than NULL, when omitted from an INSERT statement.

**DELETE statement**

The ***DELETE*** statement deletes existing rows in a table. The ***FROM*** keyword is followed by the table name whose rows are to be deleted. An optional WHERE clause specifies which rows should be deleted. Omitting the WHERE clause results in all rows in the table being deleted.

TRUNCATE

*The****TRUNCATE****statement deletes all rows from a table. TRUNCATE is nearly identical to a DELETE statement with no WHERE clause except for minor differences that depend on the database system.*

TRUNCATE TABLE TableName;

1.9 Primary keys

**Primary keys**

A ***primary key*** is a column, or group of columns, used to identify a row. The primary key is usually the table's first column and appears on the left of table diagrams, but the position is not significant to the database. In this material, a solid circle (●) precedes the primary key in table diagrams. Ex: ID is the primary key of the Employee table below.

The primary key is specified in SQL when the table is created. If a table contains several unique columns, any unique column, or group of columns, may be specified.

Primary keys must be:

* **Unique**. This rule ensures that each value identifies at most one row.
* **Not NULL**. This rule ensures that each value identifies at least one row.

Together, the two rules ensure that each primary key value identifies exactly one row.

**Composite primary keys**

Sometimes multiple columns are necessary to identify a row. A ***simple primary key*** consists of a single column. A ***composite primary key*** consists of multiple columns. Composite primary keys are denoted with parentheses. Ex: (ColumnA, ColumnB).

Composite primary keys must be:

* **Unique**. Values of primary key columns, when grouped together, must be unique. No group of values may repeat in multiple rows.
* **Not NULL**. No column of a composite primary key may contain a NULL value.
* ***Minimal***. All primary key columns are necessary for uniqueness. When any column is removed, the resulting simple or composite column is no longer unique.

Simple primary keys are necessarily minimal, since no column can be removed from a simple key.

**PRIMARY KEY constraint**

The ***PRIMARY KEY*** constraint in a CREATE TABLE statement names the table's primary key. The PRIMARY KEY constraint ensures that a column or group of columns is always unique and non-null.

**Auto-increment columns**

An ***auto-increment column*** is a numeric column that is assigned an automatically incrementing value when a new row is inserted. The ***AUTO\_INCREMENT*** keyword defines an auto-increment column. AUTO\_INCREMENT follows the column's data type in a CREATE TABLE statement.

Integer primary keys are commonly implemented as auto-increment columns. In MySQL, AUTO\_INCREMENT may be applied only to primary key columns.

Figure 1.9.2: Auto-increment primary key example.

CREATE TABLE Employee (

ID SMALLINT UNSIGNED AUTO\_INCREMENT,

Name VARCHAR(60),

BirthDate DATE,

Salary DECIMAL(7,2),

PRIMARY KEY (ID)

);

Database users occasionally make the following errors when inserting primary keys:

* Inserting values for auto-increment primary keys.
* Omitting values for primary keys that are not auto-increment columns.

MySQL allows insertion of a specific value to an auto-increment column. However, overriding auto-increment for a primary key is usually a mistake.

**Foreign keys**

A ***foreign key*** is a column, or group of columns, that refer to a primary key. The data types of the foreign and primary keys must be the same, but the names may be different. In this material, an empty circle (○) precedes foreign keys in table diagrams, and an arrow leads to the referenced primary key.

Foreign keys do not obey the same rules as primary keys. Foreign key values may be repeated and may be NULL. Ex: In the animation below, the Sales and Marketing departments have the same manager. The Technical Support department does not have a manager.

Foreign keys obey a relational rule called referential integrity. ***Referential integrity*** requires foreign key values must either be NULL or match some value of the referenced primary key.

**Special cases**

Multiple foreign keys may refer to the same primary key. In the DepartmentStaff table, the Manager and Assistant foreign keys both refer to ID, the primary key of Employee.

A foreign key may refer to a primary key in the same table. In the EmployeeManager table, the Manager foreign key refers to the ID primary key.

A foreign key that refers to a composite primary key must also be composite. All columns of a composite foreign key must either be NULL or match the corresponding primary key columns. Ex: In the HealthPlan table, (EmployeeID, DependentNumber) is a composite foreign key that refers to the Family table's composite primary key.

**FOREIGN KEY constraint**

A foreign key constraint is added to a CREATE TABLE statement with the ***FOREIGN KEY*** and ***REFERENCES*** keywords. When a foreign key constraint is specified, the database rejects insert, update, and delete statements that violate referential integrity.

1.11 Referential integrity

**Referential integrity rule**

A ***fully NULL*** foreign key is a simple or composite foreign key in which all columns are NULL. ***Referential integrity*** is a relational rule that requires foreign key values are either fully NULL or match some primary key value.

In a relational database, foreign keys must obey referential integrity at all times. Occasionally, data entry errors or incomplete data result in referential integrity violations. Violations must be corrected before data is stored in the database.

**Referential integrity violations**

Referential integrity can be violated in four ways:

1. A primary key is updated.
2. A foreign key is updated.
3. A row containing a primary key is deleted.
4. A row containing a foreign key is inserted.

Only these four operations can violate referential integrity. Primary key inserts and foreign key deletes never violate referential integrity.

**Referential integrity actions**

An insert, update, or delete that violates referential integrity can be corrected manually. However, manual corrections are time-consuming and error-prone. Instead, databases automatically correct referential integrity violations with any of four actions, specified as SQL constraints:

* ***RESTRICT***rejects an insert, update, or delete that violates referential integrity.
* ***SET NULL*** sets invalid foreign keys to NULL.
* ***SET DEFAULT*** sets invalid foreign keys to the foreign key default value.
* ***CASCADE*** propagates primary key changes to foreign keys.

CASCADE behaves differently for primary key updates and deletes. If a primary key is deleted, rows containing matching foreign keys are deleted. If a primary key is updated, matching foreign keys are updated to the same value.

RESTRICT, SET NULL, and SET DEFAULT apply to primary key update and delete, and foreign key insert and update. CASCADE applies to primary key update and delete only. The animation below illustrates these actions when a primary key is deleted.

**ON UPDATE and ON DELETE clauses**

For foreign key inserts and updates, MySQL supports only RESTRICT. Foreign key inserts and updates that violate referential integrity are automatically rejected.

For primary key updates and deletes, MySQL supports all four actions. Actions are specified in the optional ***ON UPDATE*** and ***ON DELETE*** clauses of the FOREIGN KEY constraint. ON UPDATE and ON DELETE are followed by either RESTRICT, SET NULL, SET DEFAULT, or CASCADE.

ON UPDATE and ON DELETE determine what happens to the foreign key when the referenced primary key is updated or deleted. When several foreign keys refer to the same primary key, different actions can be specified for each foreign key.

MySQL has several limitations on primary key updates and deletes:

* RESTRICT is applied when the ON UPDATE or ON DELETE clause is omitted.
* SET NULL cannot be used when a foreign key is not allowed NULL values.
* SET DEFAULT is not supported in some MySQL configurations.

ON UPDATE and ON DELETE are standard SQL. The clauses are supported by most relational databases, but details and limitations vary.

1.12 Constraints

**Column and table constraints**

A ***constraint*** is a rule that governs allowable values in a database. Constraints are based on relational and business rules, and implemented with special keywords in a CREATE TABLE statement. The database automatically rejects insert, update, and delete statements that violate a constraint.

The following constraints are described elsewhere in this material:

* NOT NULL
* DEFAULT
* PRIMARY KEY
* FOREIGN KEY

A ***column constraint*** appears after the column name and data type in a CREATE TABLE statement. Column constraints govern values in a single column. Ex: NOT NULL is a column constraint.

A ***table constraint*** appears in a separate clause of a CREATE TABLE statement and governs values in one or more columns. Ex: FOREIGN KEY is a table constraint.

Some constraint types can be defined as either column or table constraints. Ex: A PRIMARY KEY constraint on a single column can appear either in the column declaration or a separate CREATE TABLE clause. A PRIMARY KEY constraint on a composite column must be defined as a table constraint.

DEFAULT constraint

*The DEFAULT constraint does not actually limit allowable values in a column. Instead, DEFAULT specifies a value that is inserted when a column is omitted from an INSERT statement. For this reason, DEFAULT is not always considered a constraint.*

**UNIQUE constraint**

The ***UNIQUE*** constraint ensures that values in a column, or group of columns, are unique. When applied to a single column, UNIQUE may appear either in the column declaration or a separate clause. When applied to a group of columns, UNIQUE is a table constraint and must appear in a separate clause.

The UNIQUE constraint can be applied to primary key columns but is unnecessary, since primary keys are automatically unique.

MySQL creates an index for each UNIQUE constraint. The index stores the values of the unique column, or group of columns, in sorted order. When new values are inserted or updated, MySQL searches the index to quickly determine if the new value is unique.

**Constraint names**

Table constraints may be named using the optional ***CONSTRAINT*** keyword, followed by the constraint name and declaration. If no name is provided, the database generates a default name. Constraint names appear in error messages when constraints are violated.

Most column constraints cannot be named. However, the CHECK column constraint is an exception and can be named with a CONSTRAINT clause in the column declaration.

**Adding and dropping constraints**

Constraints are added and dropped with the ALTER TABLE TableName followed by an ADD, DROP, or CHANGE clause.

Unnamed constraints such as NOT NULL and DEFAULT are added or dropped with a CHANGE clause:

* CHANGE CurrentColumnName NewColumnName NewDataType [ConstraintDeclaration]

Named constraints are added with an ADD clause:

* ADD [CONSTRAINT ConstraintName] PRIMARY KEY (Column1, Column2 ...)
* ADD [CONSTRAINT ConstraintName] FOREIGN KEY (Column1, Column2 ...) REFERENCES TableName (Column)
* ADD [CONSTRAINT ConstraintName] UNIQUE (Column1, Column2 ...)
* ADD [CONSTRAINT ConstraintName] CHECK (expression)

Adding a constraint fails when the table contains data that violates the constraint.

Named constraints are dropped with a DROP clause:

* DROP PRIMARY KEY
* DROP FOREIGN KEY ConstraintName
* DROP INDEX ConstraintName (drops UNIQUE constraints)
* DROP CHECK ConstraintName
* DROP CONSTRAINT ConstraintName (drops any named constraint)

Dropping a table fails when a foreign key constraint refers to the table's primary key. Before dropping the table, either the foreign key constraint or the foreign key table must be dropped.

2.1 Special operators and clauses

**IN operator**

The ***IN*** operator is used in a WHERE clause to determine if a value matches one of several values. The SELECT statement in the figure below uses the IN operator to select only rows where the Language column has a Dutch, Kongo, or Albanian value.

**BETWEEN operator**

The ***BETWEEN*** operator provides an alternative way to determine if a value is between two other values. The operator is written value BETWEEN minValue AND maxValue and is equivalent to value >= minValue AND value <= maxValue.

**LIKE operator**

The ***LIKE*** operator, when used in a WHERE clause, matches text against a pattern using the two wildcard characters % and \_.

* % matches any number of characters. Ex: LIKE 'L%t' matches "Lt", "Lot", "Lift", and "Lol cat".
* \_ matches exactly one character. Ex: LIKE 'L\_t' matches "Lot" and "Lit" but not "Lt" and "Loot".

The LIKE operator performs case-insensitive pattern matching by default or case-sensitive pattern matching if followed by the ***BINARY*** keyword. Ex: LIKE BINARY 'L%t' matches 'Left' but not 'left'.

To search for the wildcard characters % or \_, a backslash (\) must precede % or \_. Ex: LIKE 'a\%' matches "a%".

**DISTINCT clause**

The ***DISTINCT*** clause is used with a SELECT statement to return only unique or 'distinct' values. Ex: The first SELECT statement in the figure below results in two 'Spanish' rows, but the second SELECT statement returns only unique languages, resulting in only one 'Spanish' row.

**ORDER BY clause**

A SELECT statement selects rows from a table with no guarantee the data will come back in a certain order. The ***ORDER BY*** clause orders selected rows by one or more columns in ascending (alphabetic or increasing) order. The ***DESC*** keyword with the ORDER BY clause orders rows in descending order.

2.2 Simple functions

**Numeric functions**

A ***function*** operates on an expression enclosed in parentheses, called an ***argument***, and returns a value. Usually, the argument is a simple expression, such as a column name or fixed value. Some functions have several arguments, separated by commas, and a few have no arguments at all.

Each function operates on, and evaluates to, specific data types. Ex: The LOG() function operates on any numeric data type and returns a DOUBLE value. If the argument is invalid, the function returns NULL. Ex: The SQRT() function computes the square root of positive numbers only, so SQRT(-1) returns NULL.

Numeric functions operate on, and evaluate to, integer and decimal data types.

Table 2.2.1: Common numeric functions.

|  |  |  |
| --- | --- | --- |
| Function | Description | Example |
| ***ABS(n)*** | Returns the absolute value of *n* | SELECT ABS(-5);  returns 5 |
| ***LOG(n)*** | Returns the natural logarithm of *n* | SELECT LOG(10);  returns 2.302585092994046 |
| ***POW(x, y)*** | Returns *x* to the power of *y* | SELECT POW(2, 3);  returns 8 |
| ***RAND()*** | Returns a random number between 0 (inclusive) and 1 (exclusive) | SELECT RAND();  returns 0.11831825703225868 |
| ***ROUND(n, d)*** | Returns *n* rounded to *d* decimal places | SELECT ROUND(16.25, 1);  returns 16.3 |
| ***SQRT(n)*** | Returns the square root of *n* | SELECT SQRT(25);  returns 5 |

**String functions**

String functions manipulate string values. SQL string functions are similar to string functions in programming languages like Java and Python.

Table 2.2.2: Common string functions.

|  |  |  |
| --- | --- | --- |
| Function | Description | Example |
| ***CONCAT(s1, s2, ...)*** | Returns the string that results from concatenating the string arguments | SELECT CONCAT('Dis', 'en', 'gage');  returns 'Disengage' |
| ***LOWER(s)*** | Returns the lowercase *s* | SELECT LOWER('MySQL');  returns 'mysql' |
| ***REPLACE(s, from, to)*** | Returns the string *s* with all occurrences of *from* replaced with *to* | SELECT REPLACE('This and that', 'and', 'or');  returns 'This or that' |
| ***SUBSTRING(s, pos, len)*** | Returns the substring from *s* that starts at position *pos* and has length *len* | SELECT SUBSTRING('Boomerang', 1, 4);  returns 'Boom' |
| ***TRIM(s)*** | Returns the string *s* without leading and trailing spaces | SELECT TRIM(' test ');  returns 'test' |
| ***UPPER(s)*** | Returns the uppercase *s* | SELECT UPPER('mysql');  returns 'MYSQL' |

**Date and time functions**

Date and time functions operate on DATE, TIME, and DATETIME data types.

Table 2.2.3: Common date and time functions.

|  |  |  |
| --- | --- | --- |
| Function | Description | Example |
| ***CURDATE()*** ***CURTIME()*** ***NOW()*** | Returns the current date, time, or date and time in 'YYYY-MM-DD', 'HH:MM:SS', or 'YYYY-MM-DD HH:MM:SS' format | SELECT CURDATE();   returns '2019-01-25'  SELECT CURTIME();   returns '21:05:44'  SELECT NOW();   returns '2019-01-25 21:05:44' |
| ***DATE(expr)*** ***TIME(expr)*** | Extracts the date or time from a date or datetime expression *expr* | SELECT DATE('2013-03-25 22:11:45');  returns '2013-03-25'  SELECT TIME('2013-03-25 22:11:45');  returns '22:11:45' |
| ***DAY(d)*** ***MONTH(d)*** ***YEAR(d)*** | Returns the day, month, or year from date *d* | SELECT DAY('2016-10-25');  returns 25  SELECT MONTH('2016-10-25');  returns 10  SELECT YEAR('2016-10-25');  returns 2016 |
| ***HOUR(t)*** ***MINUTE(t)*** ***SECOND(t)*** | Returns the hour, minute, or second from time *t* | SELECT HOUR('22:11:45');  returns 22  SELECT MINUTE('22:11:45');  returns 11  SELECT SECOND('22:11:45');  returns 45 |
| ***DATEDIFF(expr1, expr2)*** ***TIMEDIFF(expr1, expr2)*** | Returns *expr1 - expr2* in number of days or time values, given *expr1* and *expr2* are date, time, or datetime values | SELECT DATEDIFF('2013-03-10', '2013-03-04');  returns 6  SELECT TIMEDIFF('10:00:00', '09:45:30');  returns 00:14:30 |

2.3 Aggregate functions

**Aggregate functions**

An ***aggregate function*** processes values from a set of rows and returns a summary value. Common aggregate functions are:

* ***COUNT()*** counts the number of rows in the set.
* ***MIN()*** finds the minimum value in the set.
* ***MAX()*** finds the maximum value in the set.
* ***SUM()*** sums all the values in the set.
* ***AVG()*** computes the arithmetic mean of all the values in the set.

Aggregate functions appear in a SELECT clause and process all rows that satisfy the WHERE clause condition. If a SELECT statement has no WHERE clause, the aggregate function processes all rows.

**GROUP BY clause**

Aggregate functions are commonly used with the GROUP BY clause.

The ***GROUP BY*** clause consists of the GROUP BY keyword and one or more columns. Each simple or composite value of the column(s) becomes a group. The query computes the aggregate function separately, and returns one row, for each group.

The GROUP BY clause appears between the WHERE clause, if any, and the ORDER BY clause. Aside from the aggregate function, the SELECT clause may contain only column(s) that appear in the GROUP BY clause.

**HAVING clause**

The ***HAVING*** clause is used with the GROUP BY clause to filter group results. The optional HAVING clause follows the GROUP BY clause and precedes the optional ORDER BY clause.

**Aggregate functions and NULL values**

Aggregate functions ignore NULL values. Ex: SUM(Salary) adds all non-NULL salaries and ignores rows containing a NULL salary.

Aggregate functions and arithmetic operators handle NULL differently. Arithmetic operators return NULL when either operand is NULL. As a result, aggregate functions may generate surprising results when NULL is present. Ex: In the animations below, SUM(Salary) + SUM(Bonus) is not equal to SUM(Salary + Bonus).

2.4 Join queries

**Joins**

In relational databases, reports are commonly generated from data in multiple tables. Multi-table reports are written with join statements.

A ***join*** is a SELECT statement that combines data from two tables, known as the ***left table*** and ***right table***, into a single result. The tables are combined by comparing columns from the left and right tables, usually with the = operator. The columns must have comparable data types.

Usually, a join compares a foreign key of one table to the primary key of another. However, a join can compare any columns with comparable data types.

**Prefixes and aliases**

Occasionally, join tables contain columns with the same name. When duplicate column names appear in a query, the names must be distinguished with a prefix. The prefix is the table name followed by a period.

Use of a prefix makes column names more complex. To simplify queries or result tables, a column name can be replaced with an alias. The alias follows the column name, separated by an optional ***AS*** keyword.

Ex: In the figure below, the Name column appears in both tables and thus must have a prefix in the join query. Department.Name and Employee.Name have aliases Group and Supervisor, which simplify the result column names.

**Inner and full joins**

A ***join clause*** determines how a join query handles unmatched rows. Two common join clauses are:

* ***INNER JOIN*** selects only matching left and right table rows.
* ***FULL JOIN*** selects all left and right table rows, regardless of match.

In a FULL JOIN result table, unmatched left table rows appear with NULL values in right table columns, and vice versa.

The join clause appears between a FROM clause and an ON clause:

* The FROM clause specifies the left table.
* The INNER JOIN or FULL JOIN clause specifies the right table.
* The ***ON*** clause specifies the join columns.

An optional WHERE clause follows the ON clause.

**Left and right joins**

In some cases, the database user wants to see unmatched rows from either the left or right table, but not both. To enable these cases, relational databases support left and right joins:

* ***LEFT JOIN*** selects all left table rows, but only matching right table rows.
* ***RIGHT JOIN*** selects all right table rows, but only matching left table rows.

An ***outer join*** is any join that selects unmatched rows, including left, right, and full joins.

Join clauses are standard SQL syntax and supported by most relational databases. MySQL supports INNER, LEFT, and RIGHT JOIN but not FULL JOIN. For details of MySQL join syntax, see the link in 'Exploring further' below.

**Alternative join queries**

Inner and outer joins can be written without a JOIN clause. However, the JOIN clause clarifies join behavior and simplifies queries. *Thus, use of the JOIN clause is good practice.*

Ex: The queries in the figure below are equivalent. In the query on the right, the first SELECT returns matching rows and the second SELECT returns unmatched Department rows. The ***UNION*** keyword combines the two results into one table.

Figure 2.4.2: Equivalent left join queries.

SELECT Department.Name, Employee.Name

FROM Department

LEFT JOIN Employee

ON Manager = ID;

SELECT Department.Name, Employee.Name

FROM Department, Employee

WHERE Manager = ID

UNION

SELECT Department.Name, NULL

FROM Department

WHERE Manager NOT IN (SELECT ID FROM Employee)

OR Manager IS NULL;

2.5 Equijoins, self-joins, and cross-joins

**Equijoins**

An ***equijoin*** compares columns of two tables with the = operator. Most joins are equijoins. A ***non-equijoin*** compares columns with an operator other than =, such as < and >.

In the figure below, a non-equijoin selects all buyers along with properties priced below the buyer's maximum price.

**Self-joins**

A ***self-join*** joins a table to itself. A self-join can compare any columns of a table, as long as the columns have comparable data types. If a foreign key and the referenced primary key are in the same table, a self-join commonly compares those key columns. In a self-join, aliases are necessary to distinguish left and right tables.

In the figure below, A is the left table's alias, and B is the right table's alias. A.Name is the Name column of the left table, representing the employee. B.Name is the Name column of the right table, representing the employee's manager. The result shows employees along with each employee's manager.

**Cross-joins**

A ***cross-join*** combines two tables without comparing columns. A cross-join uses a ***CROSS JOIN*** clause without an ON clause. As a result, all possible combinations of rows from both tables appear in the result.

In the figure below, all configurations of iPhone models and storage appear, along with total price.

2.6 Subqueries

**Subqueries**

A ***subquery***, sometimes called a ***nested query*** or ***inner query***, is a query within another SQL query. The subquery is typically used in a SELECT statement's WHERE clause to return data to the outer query and restrict the selected results. The subquery is placed inside parentheses ().

**Correlated subqueries**

A subquery is ***correlated*** when the subquery's WHERE clause references a column from the outer query. In a correlated subquery, the rows selected depend on what row is currently being examined by the outer query.

If a column name in the correlated subquery is identical to a column name in the outer query, the TableName.ColumnName differentiates the columns. Ex: City.CountryCode refers to the City table's CountryCode column .

An alias can also help differentiate the columns. An ***alias*** is a temporary name assigned to a column or table. The ***AS*** keyword follows a column or table name to create an alias. Ex: SELECT Name AS N FROM Country AS C creates the alias N for the Name column and alias C for the Country table. The AS keyword is optional and may be omitted. Ex: SELECT Name N FROM Country C.

In the example below, the outer SELECT statement uses a correlated subquery to find cities with a population larger than the country's average city population.

**EXISTS operator**

Correlated subqueries commonly use the ***EXISTS*** operator, which returns TRUE if a subquery selects at least one row and FALSE if no rows are selected. The ***NOT EXISTS*** operator returns TRUE if a subquery selects no rows and FALSE if at least one row is selected.

**Flattening subqueries**

Many subqueries can be rewritten as a join. Most databases optimize a subquery and outer query separately, whereas joins are optimized in one pass. So joins are usually faster and preferred when performance is a concern.

Replacing a subquery with an equivalent join is called ***flattening*** a query. The criteria for flattening subqueries are complex and depend on the SQL implementation in each database system. Most subqueries that follow IN or EXISTS, or return a single value, can be flattened. Most subqueries that follow NOT EXISTS or contain a GROUP BY clause cannot be flattened.

The following steps are a first pass at flattening a query:

1. Retain the outer query SELECT, FROM, GROUP BY, HAVING, and ORDER BY clauses.
2. Add INNER JOIN clauses for each subquery table.
3. Move comparisons between subquery and outer query columns to ON clauses.
4. Add a WHERE clause with the remaining expressions in the subquery and outer query WHERE clauses.
5. If necessary, remove duplicate rows with SELECT DISTINCT.

After this first pass, test the flattened query and adjust to achieve the correct result. Verify that the original and flattened queries are equivalent against a variety of data.

2.7 Complex query example

**Writing a complex query**

Database users frequently create complex SQL queries that join data from multiple tables to answer business questions. Ex: A bookstore might ask, "Which books are selling best in summer?" and "What types of books do customers from the West Coast purchase?"

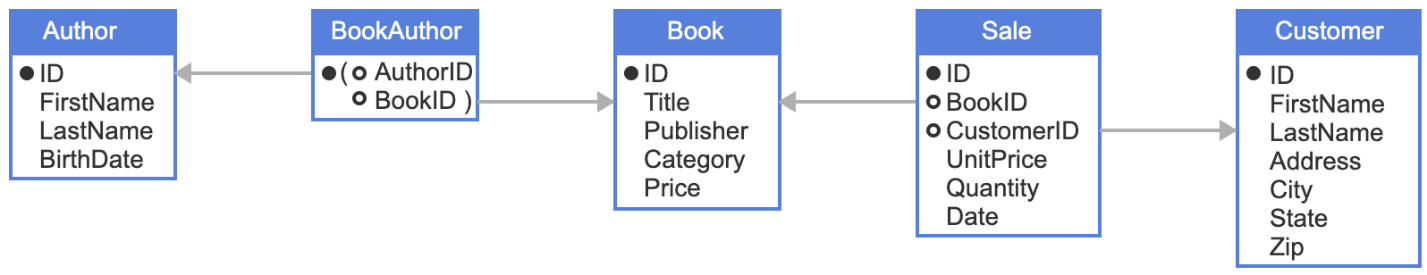
To create a complex query, a database user can employ the following strategy:

1. Examine a table diagram or other database summary to understand the tables and relationships.
2. Identify the tables containing the necessary data to answer the question.
3. Determine which columns should appear in the result table.
4. Write a query that joins the tables using the table's primary and foreign keys.
5. Break the problem into simple queries, writing one part of the query at a time.

The Zion Bookstore wants to know which books, written by a single author, generated the most sales to customers from Colorado or Oklahoma in February 2020. The information required to answer this question is spread across several tables, requiring a complex query to answer the question.

The table diagram in the figure below describes the Zion Bookstore database, which tracks books, customers, and sales.

Figure 2.7.1: Table diagram for Zion Bookstore database.



**Joining tables**

To answer the Zion Bookstore question, the result table must contain columns from the previously identified tables or columns that can be computed from the tables. The result table should contain the following: Customer state (Customer.State), Book ID (Sale.BookID), book title (Book.Title), number of books purchased (Sale.Quantity), and total price (Sale.Quantity × Sale.UnitPrice).

2.8 View tables

**Creating views**

Table design is optimized for a variety of reasons, such as minimal storage, fast query execution, and support for relational and business rules. Occasionally, the design is not ideal for database users and programmers. Ex: The Employee table may contain personal information, such as name, marital status, and birth date. A separate Address table may contain employee addresses. This design allows several employees to share the same address. Human resources staff, however, may prefer to see personal and address information in one table rather than two.

View tables solve this problem. Views restructure table columns and data types without changes to the underlying database design.

A ***view table*** is a table name associated with a SELECT statement, called the ***view query***. The ***CREATE VIEW*** statement creates a view table and specifies the view name, query, and, optionally, column names. If column names are not specified, column names are the same as in the view query result table.

Figure 2.8.1: CREATE VIEW statement.

CREATE VIEW ViewName [ ( Column1, Column2, ... ) ]

AS SelectStatement;

**Querying views**

A table specified in the view query's FROM clause is called a ***base table***. Unlike base table data, view table data is not normally stored. Instead, when a view table appears in an SQL statement, the view query is merged with the SQL query. The database executes the merged query against base tables.

In some databases, view data can be stored. A ***materialized view*** is a view for which data is stored at all times. Whenever a base table changes, the corresponding view tables can also change, so materialized views must be refreshed. To avoid the overhead of refreshing views, MySQL and many other databases do not support materialized views.

Terminology

*A view can be defined on other view tables when the view query FROM clause includes additional view tables. In this case, the additional view tables are not base tables.****Base tables****are always source tables, created as tables rather than as views.*

**Advantages of views**

View tables have several advantages:

* **Protect sensitive data**. A table may contain sensitive data. Ex: The Employee table contains compensation columns such as Salary and Bonus. A view can exclude sensitive columns but include all other columns. Authorizing users and programmers access to the view but not the underlying table protects the sensitive data.
* **Save complex queries**. Complex SELECT statements can be saved as a view. Database users can reference the view without writing the SELECT statement.
* **Save optimized queries**. Often, the same result table can be generated with equivalent SELECT statements. Although the results of equivalent statements are the same, performance may vary. To ensure fast execution, the optimal statement can be saved as a view and distributed to database users.

For the above reasons, views are supported in all relational databases and are frequently created by database administrators. Database users need not be aware of the difference between view and base tables.

**Inserting, updating, and deleting views**

View tables are commonly used in SELECT statements. Using views in INSERT, UPDATE, and DELETE statements is problematic:

* **Primary keys**. If a base table primary key does not appear in a view, an insert to the view generates a NULL primary key value. Since primary keys may not be NULL, the insert is not allowed.
* **Aggregate values**. A view query may contain aggregate functions such as AVG() or SUM(). One aggregate value corresponds to many base table values. An update or insert to the view may create a new aggregate value, which must be converted to many base table values. The conversion is undefined, so the insert or update is not allowed.
* **Join views**. In a join view, foreign keys of one base table may match primary keys of another. A delete from a view might delete foreign key rows only, or primary key rows only, or both the primary and foreign key rows. The effect of the join view delete is undefined and therefore not allowed.

The above examples illustrate just a few of many potential problems of changing data in view tables. As a result, relational databases either disallow or severely limit view table inserts, updates, and deletes. Regardless of specific database limitations, inserts, updates, and deletes to views should be avoided. Views are best for reading data.

**WITH CHECK OPTION clause**

Databases that allow view updates face one particularly bothersome behavior. A view insert or update may create a row that does not satisfy the view query WHERE clause. In this case, the inserted or updated row does not appear in the view table. From the perspective of the database user, the insert or update appears to fail even though the base tables have changed.

To prevent inserts or updates that appear to fail, databases that support view updates have an optional WITH CHECK OPTION clause. When ***WITH CHECK OPTION*** is specified, the database rejects inserts and updates that do not satisfy the view query WHERE clause. Instead, the database generates an error message that explains the violation.

Figure 2.8.2: WITH CHECK OPTION clause.

CREATE VIEW ViewName [ ( Column1, Column2, ... ) ]

AS SelectStatement

[ WITH CHECK OPTION ];

3.1 MySQL

**MySQL**

This material uses MySQL as a reference relational database system. Although the material is relevant to all relational databases, SQL syntax and many activities are based on MySQL.

***MySQL*** is a leading relational database system sponsored by Oracle. MySQL is relatively easy to install and use, yet has many advanced capabilities. MySQL runs on all major operating systems, including Linux, Unix, Mac OS, and Windows. For these reasons, MySQL is one of the most popular database systems.

MySQL is available in two editions:

* ***MySQL Community***, commonly called ***MySQL Server***, is a free edition. MySQL Server includes a complete set of database services and tools, and is suitable for non-commercial applications such as education.
* ***MySQL Enterprise*** is a paid edition for managing commercial databases. MySQL Enterprise includes MySQL Server and additional administrative applications.

This book is based on MySQL Server release 8.0. Forthcoming versions of this book will be upgraded to MySQL Server 8.1, released in July 2023. Complete documentation for MySQL Server 8.0 is available online.

**MySQL Command-Line Client**

The ***MySQL Command-Line Client*** is a text interface included in the MySQL Server download. The Command-Line Client allows developers to connect to the database server, perform administrative functions, and execute SQL statements.

To run the Command-Line Client, a user must first open a Command Prompt on Windows or a Terminal on a Mac:

* Windows: Click the Start button in the Taskbar, type "cmd", then click Command Prompt.
* Mac: Click on the Terminal application, usually found in the Applications > Utilities folder.

When MySQL Command-Line Client is started with the root account, the user is prompted to enter the root account password. Then Command-Line Client attempts to connect to the database server running on the local machine.

MySQL Server returns an ***error code*** and description when an SQL statement is syntactically incorrect or the database cannot execute the statement.

Figure 3.1.2: MySQL error codes.

mysql> SELECT FROM city;

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to

your MySQL Server version for the right syntax to use near 'FROM city' at line 1

mysql> INSERT INTO city VALUES (123, 'Amsterdam', 'NLD', 'Noord-Holland', 731200);

ERROR 1062 (23000): Duplicate entry '123' for key 'PRIMARY'

**MySQL Workbench**

Some developers prefer to interact with MySQL Server via a graphical user interface. ***MySQL Workbench*** is installed with MySQL Server and allows developers to execute SQL commands using an editor. When MySQL Workbench is started, the user can connect to MySQL Server running on the local machine or on the network.

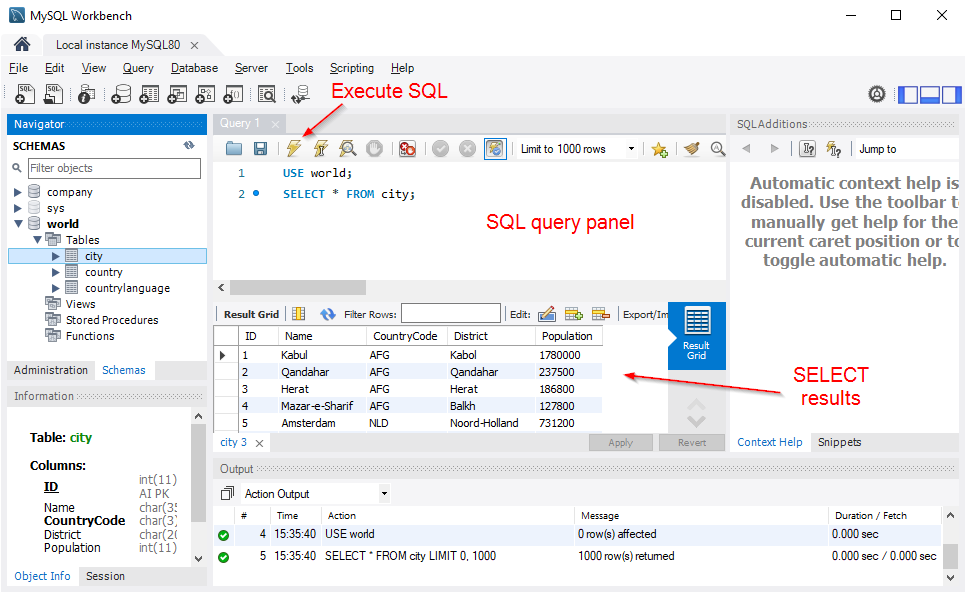
The figure below shows the MySQL Workbench home screen on Windows. The Mac version has some minor differences. Clicking on the box labeled **Local Instance MySQL80** connects to MySQL Server running on the same computer as MySQL Workbench.

After connecting to MySQL server, Workbench shows the **Navigator** sidebar on the left with two tabs:

1. The **Administration** tab shows various administrative options, like checking the server's status, importing/exporting data, and starting/stopping the MySQL server.
2. The **Schemas** tab shows a list of available databases. A database can be expanded to show the database's tables.

The figure below shows the world database's three tables: city, country, and countrylanguage. The query panel is where the user enters SQL statements. Pressing the lightning bolt icon executes the SQL statements and shows the results below the query panel. A summary of the executed statements is shown at the bottom of the window.

Figure 3.1.4: MySQL Workbench executing a SELECT statement.



3.2 MySQL architecture

**Layers**

***Architecture*** describes the components of a computer system and the relationships between components. This section describes MySQL architecture. Other relational databases have similar components, but component details and relationships vary greatly.

MySQL components are organized in four layers:

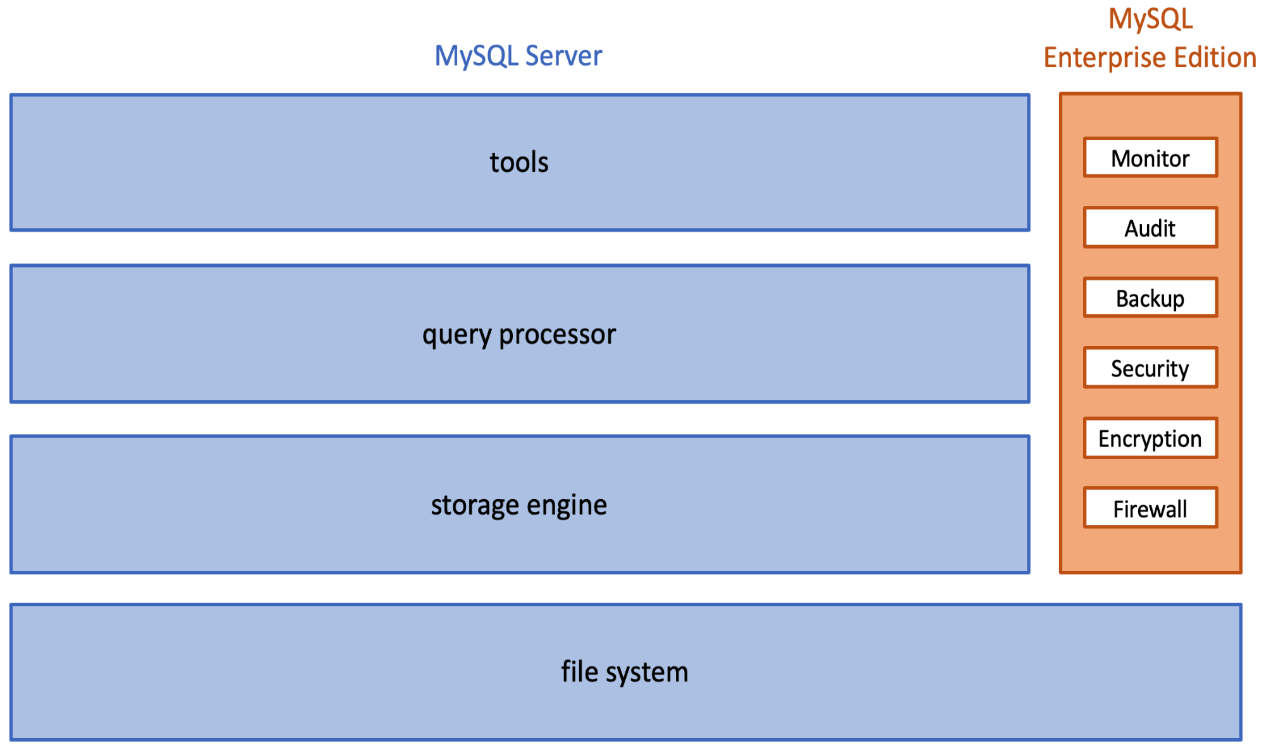
* ***Tools*** interact directly with database users and administrators, and send queries to the query processor.
* The ***query processor*** manages connections from multiple users and compiles queries into low-level instructions for the storage engine.
* The ***storage engine***, also called a ***storage manager***, executes instructions, manages indexes, and interacts with the file system. Some storage engines support database transactions, described elsewhere in this material.
* The ***file system*** accesses data on storage media. The file system contains both system and user data, such as log files, tables, and indexes.

MySQL is available in a free version, called ***MySQL Server***, and a paid version, called ***MySQL Enterprise Edition***. The Enterprise Edition includes MySQL Server and components for high-end commercial installations, such as:

* ***Monitor*** collects and displays information on CPU, memory, and index utilization, as well as queries and results. Database administrators use Enterprise Monitor to manage and tune large databases with many users.
* ***Audit*** keeps track of all database changes. For each change, Audit tracks the time of change and who made the change. Audit supports government and business audit requirements for sensitive databases such as financial, medical, and defense.

Additional Enterprise Edition components provide advanced support for backup, security, encryption, and firewall. Enterprise Edition components are intended for database administrators, not users.

Figure 3.2.1: MySQL layers.



**Tools**

The tools layer includes Connectors and APIs, Workbench, and utility programs.

Connectors and APIs are groups of application programming interfaces, linking applications to the query processor layer. Connectors are newer and developed by Oracle, which sponsors MySQL. APIs are older and, with the exception of the C API, developed by other organizations. Most programmers use Connectors, but system programmers may write specialized utilities in C with the C API.

Workbench is a desktop application to manage and use databases. Workbench is designed for both database administrators and users.

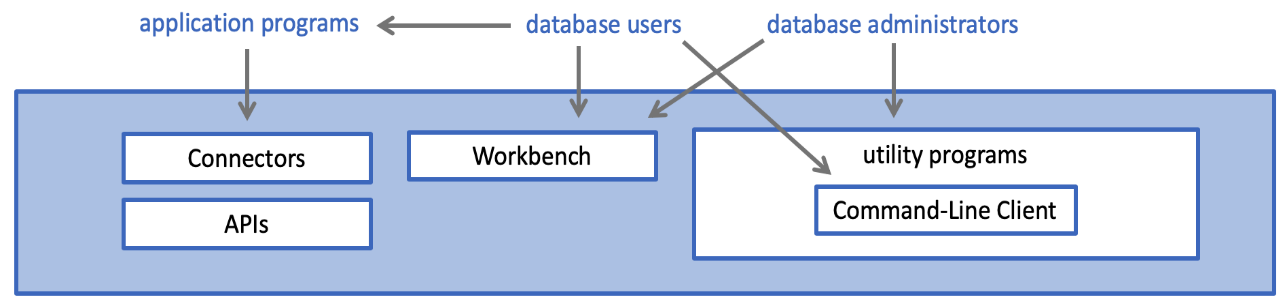
***Utility programs*** include approximately 30 tools, grouped in five categories: installation, client, administrative, developer, and miscellaneous tools. Most utility programs are intended for database administrators or programmers. Example functions include:

* Upgrade existing databases to a new MySQL release
* Backup databases
* Import data to databases
* Inspect log files
* Administer database servers

The Command-Line Client is a particularly important utility program, commonly used by both database administrators and users. The Command-Line Client displays the mysql> prompt and processes individual SQL queries interactively.

Connectors, Workbench, and the Command-Line Client are described elsewhere in this material.

Figure 3.2.2: Tools.



**Query processor**

The query processor layer has two main functions: manage connections and compile queries.

A ***connection*** is a link between tools and the query processor. Each connection specifies a database name, server address, logon name, and password. The connection manager creates connections and manages communications between tools and the query parser.

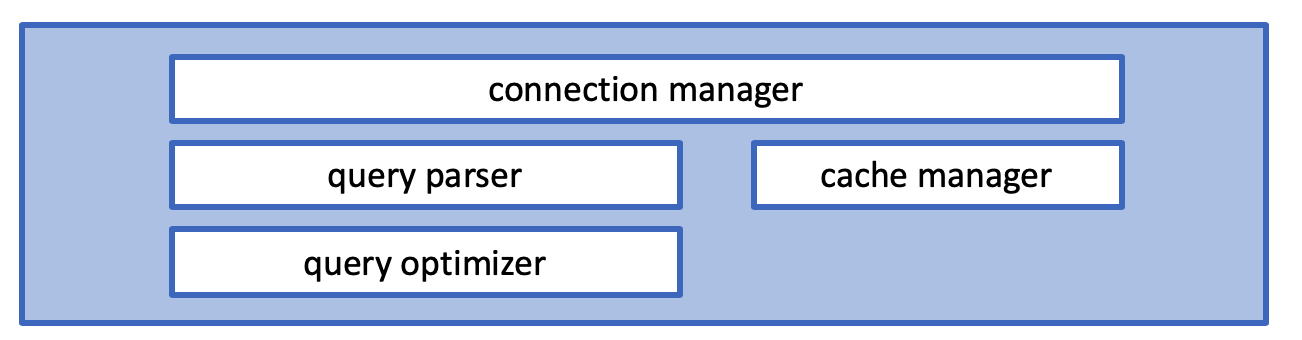
Query compilation generates a query execution plan. An ***execution plan*** is a detailed, low-level sequence of steps that specify exactly how to process a query.

The query processor generates an execution plan in two steps:

1. The ***query parser*** checks each query for syntax errors and converts valid queries to an internal representation.
2. The ***query optimizer*** reads the internal representation, generates alternative execution plans, estimates execution times, and selects the fastest plan. Estimates are based on heuristics and statistics about data, like the number of rows in each table and the number of values in each column. These statistics are maintained in the data dictionary, described below.

For optimal performance, the query processor layer has a ***cache manager*** that stores reusable information in main memory. Ex: The cache manager retains execution plans for queries that are submitted multiple times. If data used in repeated queries does not change, the cache manager may also save query results.

Figure 3.2.3: Query processor.



**Storage engine**

The storage engine layer has two main functions: transaction management and data access.

Transaction management includes the concurrency system, recovery system, and lock manager. These components ensure all transactions are atomic, consistent, isolated, and durable, as explained elsewhere in this material.

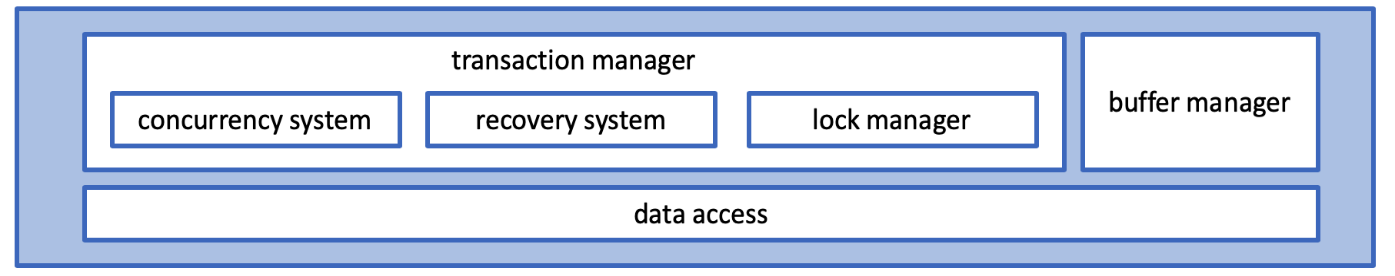
The data access component communicates with the file system and translates table, column, and index reads into block addresses.

To reduce data access time, the ***buffer manager*** retains data blocks from the file system for possible reuse. The data blocks are retained in an area of main memory called the ***buffer***. Ex: If queries frequently access department data, the buffer manager may retain some or all blocks of the Department table. The buffer manager is similar to the cache manager of the query processor layer.

The buffer manager has a fixed amount of memory. As the database processes queries and reads blocks, an algorithm determines which blocks to retain and which to discard. The InnoDB buffer manager uses a ***least recently used*** or ***LRU*** algorithm. The LRU algorithm tracks the time each block was last accessed and, when space is needed, discards 'stale' blocks. If data in a block has been updated, discarded blocks are first saved on disk.

MySQL supports nine storage engines, including InnoDB, MyISAM, CSV, and MEMORY. Each storage engine is optimized for a specific application, such as transaction management or analytics. The database administrator can assign a different storage engine to each table in a database. InnoDB is the default and most commonly used storage engine. InnoDB supports transactions, but many other storage engines do not.

Figure 3.2.4: Storage engine.



**File system**

The file system layer consists of data stored on storage media and organized in files. The file system contains three types of data for each database: user data, log files, and a data dictionary.

User data includes tables and indexes. Specific storage structures for tables and indexes are described elsewhere in this material.

Log files contain a detailed, sequential record of each change applied to a database. The recovery system uses log files to restore data in the event of a transaction, system, or storage media failure.

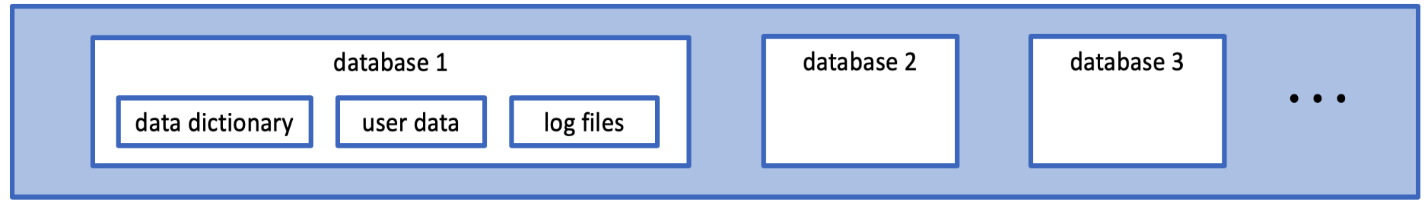
A ***catalog***, also known as a ***data dictionary***, is a directory of tables, columns, keys, indexes, and other objects in a relational database. All relational databases contain a catalog. Query processors and storage managers use catalog information when queries are processed and executed.

MySQL uses the term 'data dictionary'. The MySQL data dictionary contains roughly 30 tables, including:

* tables describes all tables
* table\_stats contains table statistics, such as the number of rows in each table
* columns describes all columns
* foreign\_keys describes all foreign keys
* indexes describes all indexes
* routines describes all stored procedures and stored functions
* triggers describes all triggers

Data dictionary tables cannot be accessed directly with SELECT, INSERT, UPDATE, and DELETE queries. However, the table contents can be accessed indirectly. The SHOW query is compiled as a SELECT query against dictionary tables. Ex: SHOW COLUMNS generates a SELECT query against the columns table. CREATE generates an INSERT, ALTER generates an UPDATE, and DROP generates a DELETE against dictionary tables.

Figure 3.2.5: File system.



3.3 MySQL Workbench: Import and export

**Import a database**

MySQL Workbench can import an entire database from an SQL file. The SQL file normally contains SQL statements to create a database, create tables for the database, and insert the data into the tables.

The figure below shows the contents of an SQL file called company.sql. The SQL statements create a database called company with Employee and Department tables. Five employees and four departments are inserted into the tables.

4.1 Entities, relationships, and attributes

**The entity-relationship model**

Database design begins with verbal or written requirements for the database. Requirements are formalized as an entity-relationship model and then implemented in SQL.

An ***entity-relationship model*** is a high-level representation of data requirements, ignoring implementation details. An entity-relationship model guides implementation in a particular database system, such as MySQL.

An entity-relationship model includes three kinds of objects:

* An ***entity*** is a person, place, product, concept, or activity.
* A ***relationship*** is a statement about two entities.
* An ***attribute*** is a descript­ive property of an entity.

A relationship is usually a statement about two different entities, but the two entities may be the same. A ***reflexive relationship*** relates an entity to itself.

When the model is implemented in SQL, entities typically become tables. Relationships and attributes typically become foreign keys and columns, respectively. However, some relationships and attributes become tables. Since the conversion is indirect, requirements are documented as entities, relationships, and attributes rather than tables, keys, and columns.

Terminology

***Attribute****is used in both entity-relationship and relational models. In the relational model, attribute is a formal term for column. Since entity-relationship attributes typically become relational columns, the meaning of attribute is similar in both models.*

**Entity-relationship diagram and glossary**

An ***entity-relationship diagram***, commonly called an ***ER diagram***, is a schematic picture of entities, relationships, and attributes. Entities are drawn as rectangles. Relationships are drawn as lines connecting rectangles. Attributes appear as additional text within an entity rectangle, under the entity name.

Entities and relationships always appear in ER diagrams. Attributes are optional and only appear when additional detail is needed.

A ***glossary***, also known as a ***data dictionary*** or ***repository***, documents additional detail in text format. A glossary includes names, synonyms, and descriptions of entities, relationships, and attributes. For simple databases with few users, a database designer may record the glossary with a text editor. For more complex databases, the designer may use a database or software tool specifically designed for glossaries.

The ER diagram and glossary are complementary and, together, completely describe an entity-relationship model.

**Types and instances**

In entity-relationship modeling, a type is a set:

* An ***entity type*** is a set of things. Ex: All employees in a company.
* A ***relationship type*** is a set of related things. Ex: Employee-Manages-Department is a set of (employee, department) pairs, where the employee manages the department.
* An ***attribute type*** is a set of values. Ex: All employee salaries.

Entity, relationship, and attribute types usually become tables, foreign keys, and columns, respectively.

An instance is an element of a set:

* An ***entity instance*** is an individual thing. Ex: The employee Sam Snead.
* A ***relationship instance*** is a statement about entity instances. Ex: "Maria Rodriguez manages Sales."
* An ***attribute instance*** is an individual value. Ex: The salary $35,000.

Entity, relationship, and attribute instances usually become rows, foreign key values, and column values, respectively

**Database design**

Complex databases are developed in three phases:

1. ***Analysis*** develops an entity-relationship model, capturing data requirements while ignoring implementation details.
2. ***Logical design*** converts the entity-relationship model into tables, columns, and keys for a particular database system.
3. ***Physical design*** adds indexes and specifies how tables are organized on storage media.

Analysis is particularly important for complex databases with many users when documenting requirements is challenging. For small databases with just a few tables and users, analysis is less important and often omitted.

Analysis and logical design steps are summarized in the table below. Although these steps are presented in sequence, in practice execution is not always sequential. Often an early step is revisited after a later step is completed.

Physical design is dependent on specific index and table structures, which vary greatly across relational databases. Physical design is discussed elsewhere in this material.

Table 4.1.2: Analysis steps.

|  |  |
| --- | --- |
| Step | Name |
| 1 | Discover entities, relationships, and attributes |
| 2 | Determine cardinality |
| 3 | Distinguish strong and weak entities |
| 4 | Create supertype and subtype entities |

Feedback?

Table 4.1.3: Logical design steps.

|  |  |
| --- | --- |
| Step | Name |
| 5 | Implement entities |
| 6 | Implement relationships |
| 7 | Implement attributes |
| 8 | Apply normal form |

4.2 Discovery

**Discovery**

Entities, relationships, and attributes are discovered in interviews with database users and managers. Users and managers are usually familiar with data requirements from an old database, or perhaps a manual process with paper records. When users are difficult to reach, a database designer may communicate with surrogates. Ex: A sales representative might communicate on behalf of prospective customers.

In addition to interviews, written documents are a good source of data requirements. Ex: The user manual for an older version of the database is a good source of requirements.

In interviews and documents, entities, relationships, and attributes surface as nouns and verbs:

* Entities usually appear as nouns, but not all nouns are entities. Designers should ignore nouns that denote specific data or are not relevant to the database.
* Relationships are often expressed as verbs. Designers should ignore statements that are not about entities, not relevant to the database, or redundant to other relationships. Designers should look for relationships that are not explicitly stated, since users may overlook important information.
* Attributes are usually nouns that denote specific data, such as names, dates, quantities, and monetary values.

### Names

Entity names are a singular noun. Ex: Employee rather than Employees. The best names are commonly used and easily understood by database users.

Relationships names have the form Entity-Verb-Entity, such as Division-Contains-Department. When the related entities are obvious, in ER diagrams or informal conversation, Verb is sufficient and entity names can be omitted. The verb should be active rather than passive. Ex: Manages rather than IsManagedBy. Occasionally, the same verb relates different entity pairs. Ex: Order-Contains-LineItem and Division-Contains-Department.

Attribute names have the form EntityQualifierType, such as EmployeeFirstName:

* Entity is the name of the entity that the attribute describes. When the entity is obvious, in ER diagrams or informal conversation, QualifierType is sufficient and the entity name can be omitted.
* Qualifier describes the meaning of the attribute. Ex: First, Last, and Alternate. Sometimes a qualifier is unnecessary and can be omitted. Ex: StudentNumber.
* Type is chosen from a list of standard attribute types such as Name, Number, and Count. Attribute types are not identical to SQL data types. Ex: "Amount" might be an attribute type representing monetary values, implemented as the MONEY data type in SQL. "Count" might be an attribute type representing quantity, implemented as NUMBER in SQL.

Standard attribute types are documented in the glossary and applied uniformly to all attribute names.

Table 4.2.1: Example names.

|  |  |  |
| --- | --- | --- |
|  | Formal name | Informal name |
| Entity | Vehicle | Vehicle |
| Relationship | Vehicle-BelongsTo-Person | BelongsTo |
| Attribute | VehicleLicenseNumber | LicenseNumber |

**Synonyms and descriptions**

Often, entity, relationship, and attribute names have synonyms. Ex: Representative may be a synonym for SalesAgent. Synonyms are common in informal communications. To avoid confusion, one official name is selected for each entity, relationship, and attribute. Other names are documented in the glossary as synonyms.

The glossary also contains complete descriptions of entities, relationships, and attributes. The description states the meaning of each entity, relationship, or attribute in complete sentences. The description begins with the name and includes examples and counterexamples to illustrate usage.

**Database design**

The first step of the analysis phase is discovery of entities, relationships, and attributes in interviews and document review. As discovery proceeds, the designer draws an ER diagram, determines standard attributed types, and documents names, synonyms, and descriptions in the glossary.

Although the step numbers suggest a sequence, database designers commonly move back and forth between steps. As names, synonyms, and descriptions are documented, additional entities, relationships, and attributes are discovered. The ER diagram and glossary are usually developed in parallel.

4.3 Cardinality

**Relationship maximum**

In entity-relationship modeling, ***cardinality*** refers to maxima and minima of relationships and attributes.

***Relationship maximum*** is the greatest number of instances of one entity that can relate to a single instance of another entity. A relationship has two maxima, one for each of the related entities. Maxima are usually specified as one or many. A related entity is ***singular*** when the maximum is one and ***plural*** when the maximum is many.

On ER diagrams, maxima are shown as 1 or M.

**Relationship minimum**

***Relationship minimum*** is the least number of instances of one entity that can relate to a single instance of another entity. A relationship has two minima, one for each of the related entities. Minima are usually specified as zero or one. A related entity is ***optional*** when the minimum is zero and ***required*** when the minimum is one.

On ER diagrams, minima are shown after maxima in parentheses. Ex: M(1) or M(0).

**Attribute maximum and minimum**

***Attribute maximum*** is the greatest number of attribute values that can describe each entity instance. Attribute maximum is usually specified as one (singular) or many (plural).

***Attribute minimum*** is the least number of attribute values that can describe each entity instance. Attribute minimum is usually specified as zero (optional) or one (required).

In ER diagrams, attribute maximum and minimum follow the attribute name. The minimum appears in parentheses.

Occasionally, attribute maximum and minimum are specified as a number rather than zero, one, or many. Ex: If each person submits exactly two forms of identification, the IDNumber attribute of Person has a maximum and minimum of two.

In some cases, individual attributes are not unique, but a composite of several attributes is unique. Ex: Different airlines use the same flight number for different flights. AirlineCode and FlightNumber are not unique attributes of Flight, but the composite (AirlineCode, FlightNumber) is unique.

Entity-Has-Attribute relationship

*Entities have an implicit relationship with their attributes, called Entity-Has-Attribute. Attribute maximum and minimum are the cardinality of the attribute in Entity-Has-Attribute. Ex: Each vehicle has exactly one vehicle identification number, or VIN. So VehicleNumber is singular and required in Vehicle-Has-VehicleNumber.*

*An attribute is unique when the entity in Entity-Has-Attribute is singular. Ex: Each VIN describes at most one vehicle. So VehicleNumber is unique and Vehicle is singular in Vehicle-Has-VehicleNumber.*

**Database design**

Relationship and attribute cardinality depends on business rules. Ex: Usually Employee-WorksIn-Department maxima are many-one. If a company assigns employees to multiple departments, however, the maxima are many-many.

During the analysis phase, the designer looks for cardinality business rules in interviews and document review. The designer then converts business rules into 0, 1, and M specifications, and documents specifications in the ER diagram and glossary.

Depending on the desired level of detail, cardinality does not always appear on ER diagrams. Usually, ER diagrams are drawn with software tools that can automatically show or hide cardinality.

4.4 Strong and weak entities

**Strong entities**

An ***identifying attribute*** is unique, singular, and required. Identifying attribute values correspond one-to-one to, or ***identify***, entity instances.

A ***strong entity*** has one or more identifying attributes. When a strong entity is implemented as a table, one of the identifying attributes may become the primary key.

**Weak entities**

A ***weak entity*** does not have an identifying attribute. Instead, a weak entity usually has a relationship, called an ***identifying relationship***, to another entity, called an ***identifying entity***. Cardinality of the identifying entity is 1(1).

In an ER diagram, an identifying relationship has a diamond next to the identifying entity. Cardinality of the identifying entity is always 1(1), so the diamond replaces the cardinality symbol.

For weak entities, identifying relationships replace identifying attributes. Ex: In the animation above, If each project has at most one task, ProjectNumber identifies Task. If each project has many tasks, (ProjectNumber, TaskName) identifies Task. The second attribute, TaskName, must be singular, required, and unique within each project.

**Identifying entities**

A weak entity is usually identified by a strong entity. However, a weak entity can be identified by another weak entity or by several entities.

When a weak entity is identified by a weak entity or multiple entities, the identifying attribute may be complex. Ex: In the animation above:

* If each task has many subtasks, a composite attribute such as (ProjectNumber, TaskName, SubtaskCode) identifies Subtask.
* If each person and each flight has many bookings, a composite attribute such as (FlightNumber, PassengerID, BookingDate) identifies Booking.

In these cases, the identifying attribute depends on business rules and may not be apparent in the ER diagram.

**Database design**

Relationship and attribute cardinality depends on business rules. Ex: Usually Employee-WorksIn-Department maxima are many-one. If a company assigns employees to multiple departments, however, the maxima are many-many.

During the analysis phase, the designer looks for cardinality business rules in interviews and document review. The designer then converts business rules into 0, 1, and M specifications, and documents specifications in the ER diagram and glossary.

Depending on the desired level of detail, cardinality does not always appear on ER diagrams. Usually, ER diagrams are drawn with software tools that can automatically show or hide cardinality.

4.4 Strong and weak entities

**Strong entities**

An ***identifying attribute*** is unique, singular, and required. Identifying attribute values correspond one-to-one to, or ***identify***, entity instances.

A ***strong entity*** has one or more identifying attributes. When a strong entity is implemented as a table, one of the identifying attributes may become the primary key.

**Weak entities**

A ***weak entity*** does not have an identifying attribute. Instead, a weak entity usually has a relationship, called an ***identifying relationship***, to another entity, called an ***identifying entity***. Cardinality of the identifying entity is 1(1).

In an ER diagram, an identifying relationship has a diamond next to the identifying entity. Cardinality of the identifying entity is always 1(1), so the diamond replaces the cardinality symbol.

For weak entities, identifying relationships replace identifying attributes. Ex: In the animation above, If each project has at most one task, ProjectNumber identifies Task. If each project has many tasks, (ProjectNumber, TaskName) identifies Task. The second attribute, TaskName, must be singular, required, and unique within each project.

When a weak entity is identified by a weak entity or multiple entities, the identifying attribute may be complex. Ex: In the animation above:

* If each task has many subtasks, a composite attribute such as (ProjectNumber, TaskName, SubtaskCode) identifies Subtask.
* If each person and each flight has many bookings, a composite attribute such as (FlightNumber, PassengerID, BookingDate) identifies Booking.

In these cases, the identifying attribute depends on business rules and may not be apparent in the ER diagram.

**Database design**

After entities, relationships, attributes, and cardinality are determined, the database designer distinguishes strong and weak entities. For each weak entity, the identifying relationship is noted. Weak entities and identifying relationships are documented in the glossary and ER diagram.

Most database designers use software tools to manage ER diagrams. Software tools usually allow users to choose from alternative conventions and automatically switch between conventions.

4.5 Supertype and subtype entities

**Supertype and subtype entities**

An entity type is a set of entity instances. A ***subtype entity*** is a subset of another entity type, called the ***supertype entity***. Ex: Managers are a subset of employees, so Manager is a subtype entity of the Employee supertype entity. On ER diagrams, subtype entities are drawn within the supertype.

A supertype entity usually has several subtypes. Attributes of the supertype apply to all subtypes. Attributes of a subtype do not apply to other subtypes or the supertype.

A supertype entity identifies its subtype entities. The identifying relationship is called an ***IsA relationship***. Ex: Manager-IsAn-Employee relates each manager instance to the corresponding employee instance. Since a supertype entity always identifies its subtypes, the IsA relationship is assumed and can be omitted from the ER diagram.

**Similar entities and optional attributes**

Supertype and subtype entities are often created from similar entities and optional attributes.

***Similar entities*** are entities that have many common attributes and relationships. Similar entities become subtypes of a new supertype entity, as in the animation below. Common attributes and relationships move to the new supertype entity. Attributes and relationships that are not shared remain with the subtype entities.

An entity with many optional attributes also suggests new supertype and subtype entities. The entity becomes a supertype entity and retains all required attributes. Optional attributes become required attributes of new subtype entities.

Creating a new supertype for similar entities, or a new subtype for optional attributes, is neither an automatic nor objective decision. Similar entities with many common attributes are good candidates for a new supertype. Entities with many optional attributes are good candidates for a new subtype.

**Partitions**

A ***partition*** of a supertype entity is a group of mutually exclusive subtype entities. A supertype entity can have several partitions. Subtype entities within each partition are disjoint and do not share instances. Subtype entities in different partitions overlap and do share instances.

In diagrams, subtype entities within each partition are vertically aligned. Subtype entities in different partitions are horizontally aligned.

Each partition corresponds to an optional ***partition attribute*** of the supertype entity. The partition attribute indicates which subtype entity is associated with each supertype instance.

**Database design**

After entities, relationships, attributes, cardinality, and strong and weak entities are determined, the database designer looks for supertype and subtype entities. Similar entities and optional attributes suggest new supertype and subtype entities and warrant special attention. Mutually exclusive subtype entities are grouped into partitions. For each partition, a partition attribute is added to the supertype entity.

4.6 Alternative modeling conventions

**Diagram conventions**

ER diagram conventions vary widely. Ex: Some ER diagrams may:

* Depict relationship names inside a diamond.
* Depict weak entities and identifying relationships with double lines.
* Depict subtype entities with IsA relationships rather than inside of supertype entities.
* Use color, dashed lines, or double lines to convey additional information.

Variations in cardinality conventions are common. One popular convention, called ***crow's foot notation***, depicts cardinality as a circle (zero), a short line (one), or three short lines (many). The three short lines look like a bird's foot, hence the name "crow's foot notation".

**Model conventions**

ER modeling concepts also vary. Ex: Some ER models may:

* Allow relationships between three or more entities.
* Decompose a complex model into a group of related entities, called a ***subject area***.
* Refer to strong entities as ***independent*** and weak entities as ***dependent***.

Several model conventions are standardized and widely used. Leading conventions include:

* ***Unified Modeling Language***, or ***UML***, is commonly used for software development. Software data structures are similar to database structures, so UML includes ER conventions.
* ***IDEF1X*** stands for Information DEFinition version 1X. IDEF1X became popular, in part, due to early adoption by the United States Department of Defense.
* ***Chen notation*** appeared in an early ER modeling paper by Peter Chen. Chen notation is not standardized but often appears in literature and tools.

By and large, differences between conventions are stylistic rather than substantial. The choice of convention does not usually affect the resulting database design.

4.7 Implementing entities

**Selecting primary keys**

In the first step of the logical design phase, each entity becomes a table and each attribute becomes a column. Tables and columns are revised in subsequent steps.

As tables and columns are specified, primary keys are selected. Primary keys must be unique and not NULL, and thus correspond to unique and required attributes. Primary keys should also be:

* **Stable**. Primary key values should not change. When a primary key value changes, statements that specify the old value must also change. Furthermore, the new primary key value must cascade to matching foreign keys.
* **Simple**. Primary key values should be easy to type and store. Small values are easy to specify in an SQL WHERE clause and speed up query processing. Ex: A 2-byte integer is easier to type and faster to process than a 15-byte character string.
* **Meaningless**. Primary keys should not contain descriptive information. Descriptive information occasionally changes, so primary keys containing descriptive information are unstable.

Stable, simple, and meaningless primary keys are desirable but not required. Depending on database standards, these guidelines may be violated in some cases.

On table diagrams, solid bullets (●) denote primary key columns.

**Implementing strong entities**

A strong entity becomes a ***strong table***. The primary key must be unique and non-NULL, and should be stable, simple, and meaningless. Single-column primary keys are best, but if no such column exists, a composite primary key may have the required properties.

An ***artificial key*** is a single-column primary key created by the database designer when no suitable single-column or composite primary key exists. Usually artificial key values are integers, generated automatically by the database as new rows are inserted to the table. Artificial keys are stable, simple, and meaningless.

**Implementing subtype entities**

A subtype entity becomes a ***subtype table*** and is implemented as follows:

* The primary key is identical to the supertype primary key.
* The primary key is also a foreign key that references the supertype primary key.

The foreign key implements the **IsA** identifying relationship. Foreign keys that implement identifying relationships usually have the following referential integrity actions:

* Cascade on primary key update and delete
* Restrict on foreign key insert and update

On table diagrams, open bullets (**⚬**) denote foreign key columns.

**Implementing weak entities**

A weak entity becomes a ***weak table***. The primary key is usually composite and includes:

* A foreign key that references the primary key of the identifying table.
* Another column that makes the composite primary key unique. If no suitable column is available in the weak table, an artificial column can be created.

If the identifying relationship is one-one, maximum, the second column is unnecessary and the primary key includes the foreign key only.

The foreign key implements the identifying relationship and usually has the following referential integrity actions:

* Cascade on primary key update and delete
* Restrict on foreign key insert and update

On table diagrams, parentheses enclose composite primary key columns. An arrow between two tables indicates a foreign key. The foreign key is at the tail of the arrow and the referenced primary key is at the head.

If an entity is identified by multiple entities, the primary key includes one foreign key for each identifying table. The primary key may include additional columns for uniqueness, depending on business rules.

**Database design**

The implement entities step creates an initial table design and specifies primary keys. If no suitable primary key is available, an artificial key is specified. The design is augmented in subsequent steps, as relationships and attributes are implemented. The final SQL specification stabilizes as tables are reviewed for normal form.

Some implementation decisions are affected by the database system. Ex: Some database systems have tools to automatically generate new artificial key values. A database designer might choose artificial primary keys more often when these tools are available.

4.8 Implementing relationships

**Implementing many-one relationships**

The 'implement relationships' step converts relationships into keys or tables, depending on relationship cardinality.

A many-one or one-many relationship becomes a foreign key:

* The foreign key goes in the table on the 'many' side of the relationship.
* The foreign key refers to the primary key on the 'one' side.
* The foreign key name is the primary key name with an optional prefix. The prefix is derived from the relationship name and clarifies the meaning of the foreign key.

### Implementing one-one relationships

A one-one relationship becomes a foreign key:

* The foreign key can go in the table on either side of the relationship. Usually, the foreign key is placed in the table with fewer rows, to minimize the number of NULL values.
* The foreign key refers to the primary key on the opposite side of the relationship.
* The foreign key name is the primary key name with an optional prefix. The prefix is derived from the relationship name and clarifies the meaning of the foreign key.

### Implementing many-many relationships

A many-many relationship becomes a new weak table:

* The new table contains two foreign keys, referring to the primary keys of the related tables.
* The primary key of the new table is the composite of the two foreign keys.
* The new table is identified by the related tables, so primary key cascade and foreign key restrict rules are specified.
* The new table name consists of the related table names with an optional qualifier in between. The qualifier is derived from the relationship name and clarifies the meaning of the table.

In the figure below, the many-many relationship Airline-Schedules-Flight becomes the new table AirlineFlight. The primary keys from Airline and Flight become foreign keys in AirlineFlight, and the composite key (AirlineCode, FlightNumber) becomes AirlineFlight's primary key.

**Database design**

The 'implement relationships' step adds foreign keys to the initial table design. Each many-one and one-one relationship becomes a new foreign key. Each many-many relationship becomes a new dependent table containing two foreign keys.

Foreign keys that implement dependency relationships usually have the following referential integrity actions:

* Cascade on primary key update and delete
* Restrict on foreign key insert and update

Referential integrity actions are described elsewhere in this material.

4.9 Implementing attributes

**Implementing plural attributes**

In the 'implement entities' step, entities become tables and attributes become columns. Singular attributes remain in the initial table, but plural attributes move to a new weak table:

* The new table contains the plural attribute and a foreign key referencing the initial table.
* The primary key of the new table is the composite of the plural attribute and the foreign key.
* The new table is identified by the initial table, so primary key cascade and foreign key restrict rules are specified.
* The new table name consists of the initial table name followed by the attribute name.

If a plural attribute has a small, fixed maximum, the plural attribute can be implemented as multiple columns in the initial table. However, implementing plural attributes in a new table simplifies queries and is usually a better solution.

**Implementing attribute types**

Each attribute has an attribute type included in the attribute name. During the discovery step, attribute types are selected from a list of standard attribute types in the glossary.

During logical design, an SQL data type is defined for each attribute type and documented in the glossary. When an attribute becomes a column, the attribute type determines the column data type. Ex: The attribute type Code has standard data type CHAR(3), so the FlightCode column is implemented with data type CHAR(3).

**Implementing attribute cardinality**

Attributes can be unique, required, or optional:

* Each unique attribute instance describes at most one entity instance.
* Each entity instance has at least one required attribute instance.
* Each entity instance can have zero optional attribute instances.

Unique and required attributes are implemented with keywords following the column name in the CREATE TABLE statement:

* UNIQUE is specified on columns derived from unique attributes.
* NOT NULL is specified on columns derived from required attributes.
* PRIMARY KEY is specified for primary key columns. The PRIMARY KEY keyword automatically enforces unique and required, so additional keywords NOT NULL and UNIQUE are unnecessary.

UNIQUE and NOT NULL are also specified on foreign key columns derived from unique and required relationships.

Optional attributes and relationships become columns with NULLs allowed and do not require special keywords.

**Database design**

The 'implementing attributes' step specifies columns, column rules, and data types. Plural attributes become new dependent tables. Unique and required cardinality is enforced with UNIQUE, NOT NULL, and PRIMARY KEY keywords.

After the 'implementing attributes' step, the database is completely specified in SQL as CREATE TABLE statements. The final step, 'review tables for third normal form', ensures that tables do not contain redundant data and fine-tunes the design if necessary.

4.10 First, second, and third normal form

**Functional dependence**

Column A ***depends on*** column B means each B value is related to at most one A value. Columns A and B may be simple or composite. 'A depends on B' is denoted B → A.

Dependence of one column on another is called ***functional dependence***. Functional dependence reflects business rules. Ex: "Each student receives one letter grade in a course" indicates the Grade column depends on the composite column (StudentID, CourseCode).

Examples in this section illustrate functional dependence with static table data. However, functional dependence cannot be inferred from values in a table at one point in time. Today, each value of column B may relate to at most one value of column A, but future updates may alter the data.

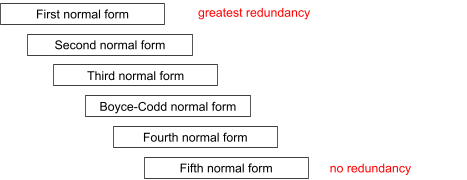
Functional dependence is not the only type of dependence. ***Multivalued dependence*** and ***join dependence*** entail dependencies between three or more columns. However, multivalued and join dependencies are complex, uncommon, and not discussed in this material.

**Normal forms**

***Redundancy*** is the repetition of related values in a table. Ex: In the Booking table, above, (222, Elvira Yin) is repeated. Redundancy causes database management problems. When related values are updated, all copies must be changed, which makes queries slow and complex. If copies are not updated uniformly, the copies become inconsistent and the correct version is uncertain.

***Normal forms*** are rules for designing tables with less redundancy. Normal forms are numbered, first through fifth. An additional normal form, Boyce-Codd, is an improved version of third normal form. The six normal forms comprise a sequence, with each successive normal form allowing less redundancy.

Figure 4.10.1: Normal forms.



Redundancy occurs when a dependence is on a column that is not unique. Ex: In the Booking table, (222, Elvira Yin) is repeated because PassengerName depends on PassengerNumber, which is not unique. Boyce-Codd normal form eliminates all dependencies on non-unique columns and, in practice, is the most important normal form.

Fourth and fifth normal forms eliminate multivalued and join dependencies, respectively. Since multivalued and join dependencies are complex and uncommon, fourth and fifth normal forms are primarily of theoretical interest.

First, second, and third normal forms are described below. Boyce-Codd normal form is described elsewhere in this material. Fourth and fifth normal forms are not described in this material.

**First normal form**

Every cell of a table contains exactly one value. A table is in ***first normal form*** when, in addition, the table has a primary key. This definition has two corollaries:

* **In a first normal form table, every non-key column depends on the primary key**. Each primary key value appears in exactly one row, and each non-key cell contains exactly one value. So each primary key value is related to exactly one non-key value.
* **A first normal form table has no duplicate rows**. Every row contains a different primary key value and therefore every row is different.

In practice, databases allow tables with duplicate rows and no primary key. However, such tables are usually temporary. Ex: A database user might load external data with duplicate rows into a temporary table. Normally, when data is moved to a permanent table, duplicate rows are removed and a primary key is created.

Alternative definitions of first normal form

*First normal form is commonly defined in several ways:*

* *The table has a primary key.*
* *Every non-key column depends on the primary key.*
* *The table cannot have duplicate rows.*
* *Every cell contains exactly one value.*

*The first three definitions are equivalent, but the last is different. The last definition is true of any relational table, and allows for duplicate rows and no primary key.*

**Second normal form**

A table is in ***second normal form*** when all non-key columns depend on the whole primary key. In other words, a non-key column cannot depend on part of a composite primary key. A table with a simple primary key is automatically in second normal form.

**Third normal form**

Redundancy can occur in a second normal form table when a non-key column depends on another non-key column. Informally, a table is in ***third normal form*** when all non-key columns depend on the key, the whole key, and nothing but the key. A formal definition appears elsewhere in this material.

4.11 Boyce-Codd normal form

**Redundancy and dependence**

*Column A depends on column B* means each B value is related to at most one A value. Columns A and B may be simple or composite. 'A depends on B' is denoted B → A. Dependence of one column on another is called *functional dependence*, and is discussed elsewhere in this material.

Redundancy occurs when a column depends on another column that is not unique.

In a Boyce-Codd normal form table, all dependencies are on unique columns. Dependence on a unique column never creates redundancy, so Boyce-Codd normal form eliminates all redundancy arising from functional dependence.

**Third normal form**

Informally, a table is in third normal form when all non-key columns depend on the key, the whole key, and nothing but the key. This definition is accurate when the primary key is the only unique column. The formal definition, below, accounts for tables with several unique columns.

A ***candidate key*** is a simple or composite column that is unique and minimal. ***Minimal*** means all columns are necessary for uniqueness. A table may have several candidate keys. The database designer designates one candidate key as the primary key.

A ***non-key*** column is a column that is not contained in a candidate key.

A table is in ***third normal form*** if, whenever a non-key column A depends on column B, then B is unique. Columns A and B may be simple or composite. Although B is unique, B is not necessarily minimal and therefore is not necessarily a candidate key.

**Boyce-Codd normal form**

The definition of third normal form applies to *non-key* columns only, which allows for occasional redundancy. Boyce-Codd normal form applies to *all* columns and eliminates this redundancy.

A table is in ***Boyce-Codd normal form*** if, whenever column A depends on column B, then B is unique. Columns A and B may be simple or composite. This definition is identical to the definition of third normal form with the term 'non-key' removed.

Boyce-Codd normal form is considered the gold standard of table design. Although fourth and fifth normal forms remove additional types of redundancy, these redundancies are uncommon and of little practical concern.

Trivial dependencies

*When the columns of A are a subset of the columns of B, A always depends on B. Ex: FareClass depends on (FlightCode, FareClass). These dependencies are called****trivial****.*

*Technically, trivial dependencies must be excluded in definitions of normal form: A table is in Boyce-Codd normal form if, for all****non-trivial****dependencies B → A, B is unique.*

5.1 Storage media

**Storage media**

Computers use a variety of media to store data, such as random-access memory, magnetic disk, optical disk, and magnetic tape. Computer media vary on four important dimensions:

* **Speed**. Speed is measured as access time and transfer rate. ***Access time*** is the time required to access the first byte in a read or write operation. ***Transfer rate*** is the speed at which data is read or written, following initial access.
* **Cost**. Cost typically ranges from pennies to dollars per gigabyte of memory, depending on the media type.
* **Capacity**. In principle, any media type can store any amount of data. In practice, capacity is limited by cost. Expensive media have limited capacity compared to inexpensive media.
* **Volatility**. ***Volatile memory*** is memory that is lost when disconnected from power. ***Non-volatile memory*** is retained without power.

Three types of media are most important for database management:

* ***Main memory***, also called ***random-access memory (RAM)***, is the primary memory used when computer programs execute. Main memory is fast, expensive, and has limited capacity.
* ***Flash memory***, also called ***solid-state drive (SSD)***, is less expensive and higher capacity than main memory. Writes to flash memory are much slower than reads, and both are much slower than main memory writes and reads.
* ***Magnetic disk***, also called ***hard-disk drive (HDD)***, is used to store large amounts of data. Magnetic disk is slower, less expensive, and higher capacity than flash memory.

Main memory is volatile. Flash memory and magnetic disk are non-volatile and therefore considered storage media. Databases store data permanently on storage media and transfer data to and from main memory during program execution.

Table 5.1.1: Media characteristics (circa 2018).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Media type | Access time (microseconds) | Transfer rate (gigabyte/second) | Cost ($/gigabyte) | Volatile |
| Main memory | .01 to .1 | > 10 | > 1 | Yes |
| Flash memory | 20 to 100 | .5 to 3 | around .25 | No |
| Magnetic disk | 5,000 to 10,000 | .05 to .2 | around .02 | No |

**Sectors, pages, and blocks**

Magnetic disk groups data in ***sectors***, traditionally 512 bytes per sector but 4 kilobytes with newer disk formats. Flash memory groups data in ***pages***, usually between 2 kilobytes and 16 kilobytes per page.

Databases and file systems use a uniform size, called a ***block***, when transferring data between main memory and storage media. Block size is independent of storage media. Storage media are managed by controllers, which convert between blocks and sectors or pages. This conversion is internal to the storage device, so the database is unaware of page and sector sizes.

Block size must be uniform within a database but, in most database systems, can be specified by the database administrator. Database systems typically support block sizes ranging from 2 kilobytes to 64 kilobytes. Smaller block sizes are usually better for transactional applications, which access a few rows per query. Larger block sizes are usually better for analytic applications, which access many rows per query.

**Row-oriented storage**

Accessing storage media is relatively slow. Since data is transferred to and from storage media in blocks, databases attempt to minimize the number of blocks required for common queries.

Most relational databases are optimized for transactional applications, which often read and write individual rows. To minimize block transfers, relational databases usually store an entire row within one block, which is called ***row-oriented storage***.

Row-oriented storage performs best when row size is small relative to block size, for two reasons:

* **Improved query performance.** When row size is small relative to block size, each block contains many rows. Queries that read and write multiple rows transfer fewer blocks, resulting in better performance.
* **Less wasted storage.** Row-oriented storage wastes a few bytes per block, since rows do not usually fit evenly into the available space. The wasted space is less than the row size. If row size is small relative to block size, this wasted space is insignificant.

Consequently, database administrators might specify a larger block size for databases containing larger rows.

Sometimes a table contains a very large column, such as 1 megabyte documents or 10 megabyte images. For tables with large columns, each row usually contains a link to the large column, which is stored in a different area. The large column might be stored in files managed by the operating system or in a special storage area managed by the database. This approach keeps row size small and improves performance of queries that do not access the large column.

**Column-oriented storage**

Some newer relational databases are optimized for analytic applications rather than transactional applications. Analytic applications often read just a few columns from many rows. In this case, column-oriented storage is optimal. In ***column-oriented*** storage, also called ***columnar storage***, each block stores values for a single column only.

Column-oriented storage benefits analytic applications in several ways:

* **Faster data access**. More column values are transferred per block, reducing time to access storage media.
* **Better data compression**. Databases often apply data compression algorithms when storing data. Data compression is usually more effective when all values have the same data type. As a result, more values are stored per block, which reduces storage and access time.

With column-oriented storage, reading or writing an entire row requires accessing multiple blocks. Consequently, column-oriented storage is a poor design for most transactional applications.

PostgreSQL and Vertica are examples of relational databases that support column-oriented storage. Many NoSQL databases, described elsewhere in this material, are optimized for analytic applications and use column-oriented storage.

Terminology

*In this material, the terms****column-oriented****and****columnar****refer to a data storage technique. Occasionally, these terms refer to a type of NoSQL database, commonly called a****wide column database****and described elsewhere in this material.*

5.2 Table structures

**Heap table**

Row-oriented storage performs better than column-oriented storage for most transactional databases. Consequently, relational databases commonly use row-oriented storage. A ***table structure*** is a scheme for organizing rows in blocks on storage media.

Databases commonly support four alternative table structures:

* Heap table
* Sorted table
* Hash table
* Table cluster

Each table in a database can have a different structure. Databases assign a default structure to all tables. Database administrators can override the default structure to optimize performance for specific queries.

In a ***heap table***, no order is imposed on rows. The database maintains a list of blocks assigned to the table, along with the address of the first available space for inserts. If all blocks are full, the database allocates a new block and inserts rows in the new block.

When a row is deleted, the space occupied by the row is marked as free. Typically, free space is tracked as a linked list, as in the animation below. Inserts are stored in the first space in the list, and the head of the list is set to the next space.

Heap tables optimize insert operations. Heap tables are particularly fast for bulk load of many rows, since rows are stored in load order. Heap tables are not optimal for queries that read rows in a specific order, such as a range of primary key values, since rows are scattered randomly across storage media.

**Sorted table**

In a ***sorted table***, the database designer identifies a ***sort column*** that determines physical row order. The sort column is usually the primary key but can be a non-key column or group of columns.

Rows are assigned to blocks according to the value of the sort column. Each block contains all rows with values in a given range. Within each block, rows are located in order of sort column values.

Sorted tables are optimal for queries that read data in order of the sort column, such as:

* JOIN on the sort column
* SELECT with range of sort column values in the WHERE clause
* SELECT with ORDER BY the sort column

Maintaining correct sort order of rows within each block can be slow. When a new row is inserted or when the sort column of an existing row is updated, free space may not be available in the correct location. To maintain the correct order efficiently, databases maintain pointers to the next row within each block, as in the animation below. With this technique, inserts and updates change two address values rather than move entire rows.

When an attempt is made to insert a row into a full block, the block splits in two. The database moves half the rows from the initial block to a new block, creating space for the insert.

In summary, sorted tables are optimized for read queries at the expense of insert and update operations. Since reads are more frequent than updates and inserts in many databases, sorted tables are often used, usually with the primary key as the sort column.

**Hash table**

In a ***hash table***, rows are assigned to buckets. A ***bucket*** is a block or group of blocks containing rows. Initially, each bucket has one block. As a table grows, some buckets eventually fill up with rows, and the database allocates additional blocks. New blocks are linked to the initial block, and the bucket becomes a chain of linked blocks.

The bucket containing each row is determined by a hash function and a hash key. The ***hash key*** is a column or group of columns, usually the primary key. The ***hash function*** computes the bucket containing the row from the hash key.

Hash functions are designed to scramble row locations and evenly distribute rows across blocks. The ***modulo function*** is a simple hash function with four steps:

1. Convert the hash key by interpreting the key's bits as an integer value.
2. Divide the integer by the number of buckets.
3. Interpret the division remainder as the bucket number.
4. Convert the bucket number to the physical address of the block containing the row.

As tables grow, a fixed hash function allocates more rows to each bucket, creating deep buckets consisting of long chains of linked blocks. Deep buckets are inefficient since a query may read several blocks to access a single row. To avoid deep buckets, databases may use dynamic hash functions. A ***dynamic hash function*** automatically allocates more blocks to the table, creates additional buckets, and distributes rows across all buckets. With more buckets, fewer rows are assigned to each bucket and, on average, buckets contain fewer linked blocks.

Hash tables are optimal for inserts and deletes of individual rows, since row location is quickly determined from the hash key. For the same reason, hash tables are optimal for selecting a single row when the hash key value is specified in the WHERE clause. Hash tables are slow on queries that select many rows with a range of values, since rows are randomly distributed across many blocks.

**able clusters**

***Table clusters***, also called ***multi-tables***, interleave rows of two or more tables in the same storage area. Table clusters have a ***cluster key***, a column that is available in all interleaved tables. The cluster key determines the order in which rows are interleaved. Rows with the same cluster key value are stored together. Usually the cluster key is the primary key of one table and the corresponding foreign key of another, as in the animation below.

Table clusters are optimal when joining interleaved tables on the cluster key, since physical row location is the same as output order. Table clusters perform poorly for many other queries:

* **Join on columns other than cluster key**. In a join on a column that is not the cluster key, physical row location is not the same as output order, so the join is slow.
* **Read multiple rows of a single table**. Table clusters spread each table across more blocks than other structures. Queries that read multiple rows may access more blocks.
* **Update cluster key**. Rows may move to different blocks when the cluster key changes.

Table clusters are not optimal for many queries and therefore are not commonly used.

5.3 Single-level indexes

**Single-level indexes**

A ***single-level index*** is a file containing column values, along with pointers to rows containing the column value. The pointer identifies the block containing the row. In some indexes, the pointer also identifies the exact location of the row within the block. Indexes are created by database designers with the CREATE INDEX command, described elsewhere in this material.

Single-level indexes are normally sorted on the column value. A sorted index is not the same as an index on a sorted table. Ex: An index on a heap table is a sorted index on an unsorted table.

If an indexed column is unique, the index has one entry for each column value. If an indexed column is not unique, the index may have multiple entries for some column values, or one entry for each column value, followed by multiple pointers.

An index is usually defined on a single column, but an index can be defined on multiple columns. In a ***multi-column index***, each index entry is a composite of values from all indexed columns. In all other respects, multi-column indexes behave exactly like indexes on a single column.

**Query processing**

To execute a SELECT query, the database can perform a table scan or an index scan:

* A ***table scan*** is a database operation that reads table blocks directly, without accessing an index.
* An ***index scan*** is a database operation that reads index blocks sequentially, in order to locate the needed table blocks.

***Hit ratio***, also called ***filter factor*** or ***selectivity***, is the percentage of table rows selected by a query. When a SELECT query is executed, the database examines the WHERE clause and estimates hit ratio. If hit ratio is high, the database performs a table scan. If hit ratio is low, the query needs only a few table blocks, so a table scan would be inefficient. Instead, the database:

1. Looks for an indexed column in the WHERE clause.
2. Scans the index.
3. Finds values that match the WHERE clause.
4. Reads the corresponding table blocks.

If the WHERE clause does not contain an indexed column, the database must perform a table scan.

Since a column value and pointer occupy less space than an entire row, an index requires fewer blocks than a table. Consequently, index scans are much faster than table scans. In some cases, indexes are small enough to reside in main memory, and index scan time is insignificant. When hit ratio is low, additional time to read the table blocks containing selected rows is insignificant.

**Binary search**

When hit ratio is low, index scans are always faster than table scans. Consider the following scenario:

* A table has 10 million rows.
* Each row is 100 bytes.
* Each block is 4 kilobytes.
* Each index entry is 10 bytes, including a 6-byte value and a 4-byte pointer.
* Magnetic disk transfer rate is 0.1 gigabytes per second.

If hit ratio is low, an index scan is roughly 10 times faster than a table scan:

* **Table scan**. The table contains 1 billion bytes (10 million rows × 100 bytes/row). The table scan takes around 10 seconds (1 gigabyte / 0.1 gigabytes/sec).
* **Index scan**. The index contains 100 million bytes (10 million rows × 1 entry/row × 10 bytes/entry). The index scan takes around 1 second (0.1 gigabyte / 0.1 gigabytes/sec). The index scan returns pointers to blocks containing rows selected by the query. When hit ratio is low, additional time to read table blocks is insignificant.

Although index scans are faster than table scans, index scans are too slow in many cases. If a single-level index is sorted, each value can be located with a binary search. In a ***binary search***, the database repeatedly splits the index in two until it finds the entry containing the search value:

1. The database first compares the search value to an entry in the middle of the index.
2. If the search value is less than the entry value, the search value is in the first half of the index. If not, the search value is in the second half.
3. The database now compares the search value to the entry in the middle of the selected half, to narrow the search to one quarter of the index.
4. The database continues in this manner until it finds the index block containing the search value.

For an index with N blocks, a binary search reads log2 N blocks, rounded up to the nearest integer. In the example above, the index has 25,000 blocks (10,000,000 rows × 10 bytes/index entry / 4,000 bytes/ block) . The binary search reads at most log2 25,000, rounded up, or 15 blocks. This search takes about 0.0006 seconds (15 blocks × 4 kilobytes/block / 0.1 gigabytes/sec).

**Primary and secondary indexes**

Indexes on a sorted table may be primary or secondary:

* A ***primary index***, also called a ***clustering index***, is an index on a sort column.
* A ***secondary index***, also called a ***nonclustering index***, is an index that is not on the sort column.

A sorted table can have only one sort column, and therefore only one primary index. Usually, the primary index is on the primary key column(s). In some database systems, the primary index may be created on any column. Tables can have many secondary indexes. All indexes of a heap or hash table are secondary, since heap and hash tables have no sort column.

Indexes may also be dense or sparse:

* A ***dense index*** contains an entry for every table row.
* A ***sparse index*** contains an entry for every table block.

When a table is sorted on an index column, the index may be sparse, as illustrated in the animation below. Primary indexes are on sort columns and usually sparse. Secondary indexes are on non-sort columns and therefore are always dense.

Sparse indexes are much faster than dense indexes since sparse indexes have fewer entries and occupy fewer blocks. Consider the following scenario:

* A table has 10 million rows.
* Each row is 100 bytes.
* Table and index blocks are 4 kilobytes.
* Each index entry is 10 bytes.

The table occupies 250,000 blocks (10 million rows × 100 bytes/row / 4 kilobytes/block). A sparse index requires 250,000 entries (one entry per table block) and occupies 625 blocks (250,000 entries × 10 bytes/entry / 4 kilobytes/block). The sparse index can easily be retained in main memory.

Primary indexes are usually sparse and sparse indexes are fast. As a result, database designers usually create a primary index on the primary key of large tables.

Terminology

*The meanings of****primary index****and****clustering index****vary. In some database systems, primary and clustering indexes are indexes on unique and non-unique sort columns, respectively. In this material, the terms are synonymous and refer to an index on any sort column.*

*A****clustering index****is not the same as a****cluster key****. A cluster key refers to a table cluster storage structure, described elsewhere in this material, and is not an index.*

**Inserts, updates, and deletes**

Inserts, updates, and deletes to tables have an impact on single-level indexes. Consider the behavior of dense indexes:

* **Insert**. When a row is inserted into a table, a new index entry is created. Since single-level indexes are sorted, the new entry must be placed in the correct location. To make space for the new entry, subsequent entries must be moved, which is too slow for large tables. Instead, the database splits an index block and reallocates entries to the new block, creating space for the new entry.
* **Delete**. When a row is deleted, the row's index entry must be deleted. The deleted entry can be either physically removed or marked as 'deleted'. Since single-level indexes are sorted, physically removing an entry requires moving all subsequent entries, which is slow. For this reason, index entries are marked as 'deleted'. Periodically, the database may reorganize the index to remove deleted entries and compress the index.
* **Update**. An update to a column that is not indexed does not affect the index. An update to an indexed column is like a delete followed by an insert. The index entry for the initial value is deleted and an index entry for the updated value is inserted.

With a sparse index, each entry corresponds to a table block rather than a table row. Index entries are inserted or deleted when blocks split or merge. Since blocks contain many rows, block splits and mergers occur less often than row inserts and deletes. Aside from frequency, however, the behavior of sparse and dense indexes is similar.

5.4 Physical design

**MySQL storage engines**

***Logical design*** specifies tables, columns, and keys. The logical design process is described elsewhere in this material. ***Physical design*** specifies indexes, table structures, and partitions. Physical design affects query performance but never affects query results.

A ***storage engine*** or ***storage manager*** translates instructions generated by a query processor into low-level commands that access data on storage media. Storage engines support different index and table structures, so physical design is dependent on a specific storage engine.

MySQL can be configured with several different storage engines, including:

* **InnoDB** is the default storage engine installed with the MySQL download. InnoDB has full support for transaction management, foreign keys, referential integrity, and locking.
* **MyISAM** has limited transaction management and locking capabilities. MyISAM is commonly used for analytic applications with limited data updates.
* **MEMORY** stores all data in main memory. MEMORY is used for fast access with databases small enough to fit in main memory.

Different databases and storage engines support different table structures and index types. Ex:

* **Table structure**. Oracle Database supports heap, sorted, hash, and cluster tables. MySQL with InnoDB supports only heap and sorted tables.
* **Index type**. MySQL with InnoDB or MyISAM supports only B+tree indexes. MySQL with MEMORY supports both B+tree and hash indexes.

This section describes the physical design process and statements for MySQL with InnoDB. The process and statements can be adapted to other databases and storage engines, but details depend on supported index and table structures.

**CREATE INDEX, DROP INDEX, and SHOW INDEX statements**

In MySQL with InnoDB:

* Indexes are always B+tree indexes.
* A primary index is automatically created on every primary key.
* A secondary index is automatically created on every foreign key.
* Additional secondary indexes are created manually with the CREATE INDEX statement.
* Tables with a primary key have sorted structure. Tables with no primary key have a heap structure.

The ***CREATE INDEX*** statement creates an index by specifying the index name and table columns that compose the index. Most indexes specify just one column, but a composite index specifies multiple columns.

The ***DROP INDEX*** statement deletes a table's index.

The ***SHOW INDEX*** statement displays a table's index. SHOW INDEX generates a result table with one row for each column of each index. A multi-column index has multiple rows in the result table.

The SQL standard includes logical design statements such as CREATE TABLE but not physical design statements such as CREATE INDEX. Nevertheless, CREATE INDEX and many other physical design statements are similar in most relational databases.

Table 5.4.1: INDEX statements.

|  |  |  |
| --- | --- | --- |
| Statement | Description | Syntax |
| CREATE INDEX | Create an index | CREATE INDEX IndexName  ON TableName (Column1, Column2, ..., ColumnN); |
| DROP INDEX | Delete an index | DROP INDEX IndexName ON TableName; |
| SHOW INDEX | Show an index | SHOW INDEX FROM TableName; |

Feedback?

Table 5.4.2: SHOW INDEX result table (selected columns).

|  |  |
| --- | --- |
| Column Name | Column Meaning |
| Table | Name of indexed table |
| Non\_unique | 0 if index is on unique column 1 if index is on non-unique column |
| Key\_name | Name of index as specified in CREATE INDEX statement or created by MySQL |
| Seq\_in\_index | 1 for single-column indexes Numeric order of column in multi-column indexes |
| Column\_name | Name of indexed column |
| Cardinality | Number of distinct values in indexed column |
| Null | YES if NULLs are allowed in column Blank if NULLs are not allowed in column |
| Index\_type | Always BTREE for InnoDB storage engine |

**EXPLAIN statement**

The ***EXPLAIN*** statement generates a result table that describes how a statement is executed by the storage engine. EXPLAIN syntax is simple and uniform in most databases: EXPLAIN statement; The statement can be any SELECT, INSERT, UPDATE, or DELETE statement.

Although the EXPLAIN statement is supported by most relational databases, the result table varies significantly. In MySQL with InnoDB, the result table has one row for each table in the statement. If the statement contains multiple queries, such as a main query and a subquery, the result table has one row for each table in each query.

The type column of the EXPLAIN result table indicates how MySQL processes a join query. Processing join queries efficiently is a complex problem, so the type column has many alternative values. Example values appear in the table below. For more information, see [MySQL EXPLAIN result table](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html).

Table 5.4.3: EXPLAIN result table (selected columns).

|  |  |
| --- | --- |
| Column Name | Column Meaning |
| select\_type | The query type. Example query types: **SIMPLE** indicates query is neither nested nor union **PRIMARY** indicates query is the outer SELECT of nested query **SUBQUERY** indicates query is an inner SELECT of nested query |
| table | Name of table described in row of EXPLAIN result table |
| type | The join type. Example join types: **const** indicates the table has at most one matching row **range** indicates a join column is compared to a constant using operators such as BETWEEN, LIKE, or IN() **eq\_ref** indicates one table row is read for each combination of rows from other tables (typically, an equijoin) **ALL** indicates a table scan is executed for each combination of rows from other tables |
| possible\_keys | All available indexes that might be used to process the query |
| key | The index selected to process the query **NULL** indicates a table scan is performed |
| ref | The constant, column, or expression to which the selected index is compared |
| rows | Estimated number of rows read from table |
| filtered | Estimated number of rows selected by WHERE clause / estimated number of rows read from table |

**Physical design process**

A database administrator may take a simple approach to physical design for MySQL with InnoDB:

1. **Create initial physical design**. Create a primary index on primary keys and a secondary index on foreign keys. In MySQL with InnoDB, these indexes are created automatically for all tables. In other databases, this step is necessary for tables larger than roughly 100 kilobytes, but can be omitted for smaller tables.
2. **Identify slow queries**. The MySQL ***slow query log*** is a file that records all long-running queries submitted to the database. Identify slow queries by inspecting the log. Most other relational databases have similar query logs.
3. **EXPLAIN slow queries**. Run EXPLAIN on each slow query to assess the effectiveness of indexes. A high value for **rows** and a low value for **filtered** indicates either a table scan or an ineffective index.
4. **Create and drop indexes** based on the EXPLAIN result table. Consider creating an index when the **rows** value is high and the **filtered** value is low. Consider dropping indexes that are never used.
5. **Partition large tables**. If some queries are still slow after indexes are created, consider partitions. Partition when slow queries access a small subset of rows of a large table. The partition column should appear in the WHERE clause of slow queries. Often, a range partition is best.

Steps 2 through 5 are ongoing activities. As the database grows and usage increases, the database administrator periodically reviews the query log, runs EXPLAIN, and adjusts the physical design for optimal performance. Additional tuning techniques can be found in the "Exploring further" section.

The five steps above can be adapted to other databases and storage engines, but the details depend on supported table structures, index types, and partition types.