DATABASE TECHNOLOGY

Database Keys

Keys are very important part of Relational database. They are used to establish and identify relation between tables. They also ensure that each record within a table can be uniquely identified by combination of one or more fields within a table.

Super Key

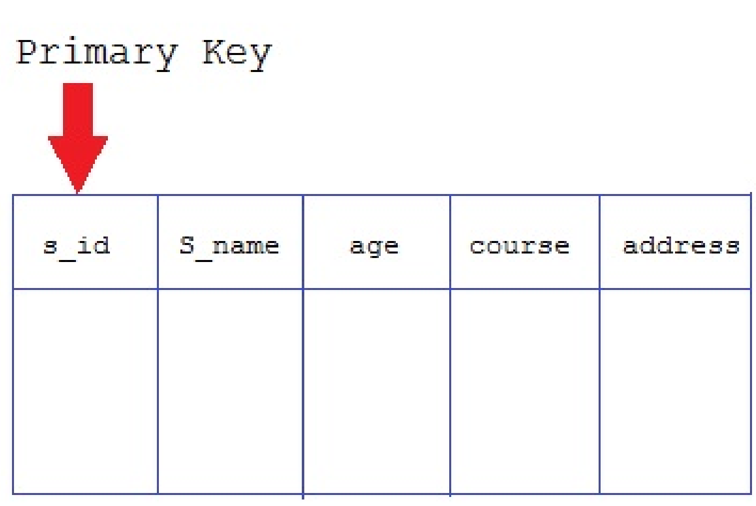
Super Key is defined as a set of attributes within a table that uniquely identifies each record within a table. Super Key is a superset of Candidate key.

Candidate Key

Candidate keys are defined as the set of fields from which primary key can be selected. It is an attribute or set of attribute that can act as a primary key for a table to uniquely identify each record in that table.

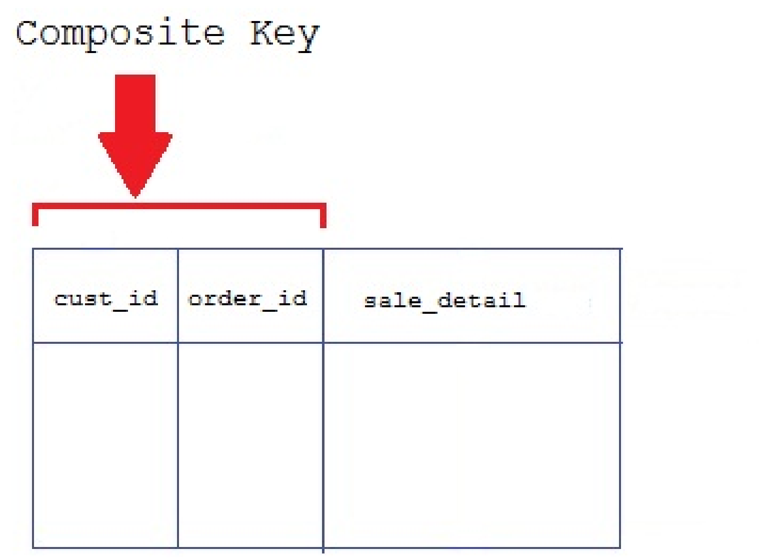
Primary Key

Primary key is a candidate key that is most appropriate to become main key of the table. It is a key that uniquely identify each record in a table.



Composite Key

Key that consist of two or more attributes that uniquely identify an entity occurance is called **Composite key**. But any attribute that makes up the **Composite key** is not a simple key in its own.



Secondary or Alternative key

The candidate key which are not selected for primary key are known as secondary keys or alternative keys

Non-key Attribute

Non-key attributes are attributes other than **candidate key** attributes in a table.

Non-prime Attribute

Non-prime Attributes are attributes other than **Primary attribute**.

← Prev (rdbms-concept.php)

Next → (database-normalization.php)

Understanding Database Normalization - Developer.com

Normalization is a very basic and important concept to consider when designing a workable relational schema in the database. The idea was proposed by E.F. Codd in 1972; since then, it has been the cornerstone of every relational database design. In fact, any schema, if it is working, must follow certain rules of it. It basically takes a relational schema through a series of test to minimize redundancy and insertion, deletion, and update anomalies by decomposing into smaller relations.

Before We Begin...

Before we begin with the rules of normalization and applying them, we must understand the following concepts.

Redundancy

One of the primary considerations for table design is to minimize storage space requirements. The tables should be designed in such a manner that very minimal data is repeatedly kept in one or more table. Storing redundant data not only takes more space but also leads to more serious problems.

EMPLOYEE_DEPARTMENT

SSN (PK)	**NAME**	**ADDRESS**	**DNO**	**DNAME**	**MGR_SSN**
123456789	Wiles, A.	123, Fondren, Houston, TX	5	R&D	234567890
234567890	Newton, I.	234, Voss, Houston, TX	5	R&D	234567890
345678901	Turing, A	2121, Castle, Spring, TX	4	Administration	234567890
456789012	Gauss, C.F.	786, Berry, Bellaire, TX	4	Administration	234567890
234567890	Euler, L.	123, Fondren, Houston, TX	5	R&D	234567890
678901234	Rieman, G.F.B	980, Dallas, Houston, TX	1	Headquarters	678901234

Table 1: Employees working in different departments showing redundant data

Observe how *DNO* and *DNAME* are repeatedly stored in the table. This type of redundant data leds to updation, insertion, and deletion anomalies.

Insertion Anomaly

If we want to insert new employee who has not been assigned any department, the attribute values of the department for that particular employees has to be kept null. This is clearly a waste of space. Further, if we insert a new employee for a department, say, *4*, the other attributes of the department have to be consistent. For example, department *4*, its *DNAME* must be '*Administration*' and *MGR_SSN* must be '*234567890*'.

Updation Anomaly

If we change the value of one of the departments by, say, changing its *DNAME* or *MGR_SSN*, we also must update the value of all employees who work in that department. Otherwise, the database will be in an inconsistent state.

Deletion Anomaly

Suppose we delete an employee from the database, for example, the last employee in the above table, who is the sole representative of *DNO=1*, then all the information about the department is also lost. This is ridiculous because we want to delete an employee information, not the whole department.

Functional dependency is the base of normalization. It states the interdependency of attributes in a table. If we know the *SSN* of particular employee, we can find out the address of that employee. This means that the attribute address is ***functionally determined*** by *SSN*. Symbolically, we can write:

```
{ SSN } → { ADDRESS }
```

Similarly,

```
{ SSN } → { ENAME, ADDRESS }
{ SSN, DNO } → { MGR_SSN}
```

When one or more attributes uniquely identifies a row, that attribute is called the **primary key**.

Normalization Forms

Rules of normalization, when applied to a table, minimize the problem areas making the table level-up into a consistent state especially during the insertion, updation, and deletion processes. The first normal form or, 1NF, is the first rule, and so on. Let's have a closer look at the rules.

First Normal Form (1NF)

Based on the **attribute atomicity**, the first normal form essentially states that we should not put more than one attribute values in a single domain. For example, in the following table, more than one PHONES are attributed to a person, that too within a single domain. This is a pretty bad design.

EMPLOYEE_PHONE

SSN (PK)	NAME	PHONES
123456789	Wiles, A.	1122334455, 3344556677, 6677889944
234567890	Newton, I.	3399118822, 3399554773
345678901	Turing, A	2266001993
456789012	Gauss, C.F.	4466577853

Instead, what we should do is as follows

EMPLOYEE_PHONE

SSN (PK)	NAME	PHONE1	PHONE2	PHONE3
123456789	Wiles, A.	1122334455	3344556677	6677889944
234567890	Newton, I.	3399118822	3399554773	*NULL*
345678901	Turing, A	2266001993	*NULL*	*NULL*
456789012	Gauss, C.F.	4466577853	*NULL*	*NULL*

There are too many NULL values; moreover, we have no way to add another phone number. We can do better by decomposing the table into two, as follows.

EMPLOYEE_NAME

SSN (PK)	NAME
123456789	Wiles, A.
234567890	Newton, I.
345678901	Turing, A
456789012	Gauss, C.F.

EMPLOYEE_PHONE

SSN (PK)	PHONES (PK)
123456789	1122334455
123456789	3344556677

123456789	6677889944
234567890	3399554773
234567890	3399118822
345678901	2266001993
456789012	4466577853

Second Normal Form (2NF)

The second normal form is based on the concept of **full functional dependency** apart from the fact that the table must be in 1NF. Here, we must remove all non-key attributes that are not completely dependent on the primary key. For example,

EMPLOYEE_PROJECT

SSN (PK)	PROJECT_NO (PK)	EMPLOYEE_NAME	PROJECT_NAME	PROJECT_HOURS
...	...	...	...	...
...	...	...	...	...

In the preceding table:

```
{ SSN } → { EMPLOYEE_NAME }
{ SSN } → { PROJ_HOURS }
```

Also,

```
{ PROJECT_NO } → { PROJECT_NAME }
{ PROJECT_NO } → { PROJECT_HOURS }
```

This is clearly a violation of 2NF, because the attributes PROJECT_HOURS and PROJECT_NAME are functionally dependent on the PROJECT_NO, individually. Also, EMPLOYEE_NAME and PROJ_HOURS are uniquely determined by SSN. What we can do is decompose the table into the following tables.

SSN (PK)PROJECT_NO (PK)PROJECT_HOURS

Third Normal Form (3NF)

As should be obvious, for a table to be in 3NF, first it must be in 2NF and the core concept behind it is that a table must not hold **transitive dependency** of attributes. Transitive dependency is: X → Y, Y → Z, X → Z. That means any non-key field must not depend on a field that is not a primary key. For example,

SSN (PK)	EMPLOYEE_NAME	BIRTH_DATE	DEPT_NAME	DEPT_ADDRESS
...	...	...	...	...

Here,

```
{ SSN } → { EMPLOYEE_NAME }
{ SSN } → { BIRTH_DATE }
{ SSN } → { DEPT_NAME }
{ SSN } → { DEPT_ADDRESS }
```

However, the anomaly is

```
{ DEPT_NAME } → { DEPT_ADDRESS }
```

whereas DEPT_NAME is a non-key. We can remove this problem by decomposing the table as follows.

SSN (PK)EMPLOYEE_NAMEBIRTH_DATEDEPT_NAME
This decomposition is lossless-join decomposition.