What is the difference between Primary Key and unique key constraint?

Asked 14 years, 2 months ago Modified 8 years, 11 months ago

Viewed 16k times

20

What is the difference between `Primary key` And `unique Key constraint`?

What's the use of it??

sql-server sql-server-2005 database-design primary-key

Share

Improve this question

Follow

edited Sep 29, 2010 at 21:41

Martin Smith

452k ● 94 ● 767 ● 870

asked Sep 29, 2010 at 9:13

pvaju896

1,417 ● 6 ● 26 ● 47

Are you asking about difference between key and constraint (PK and unique constraint) or about PK constraint and UK constraint? – Gennady Vanin Геннадий Ванин Oct 1, 2010 at 0:16

5 Answers Sorted by: Highest score (default)

12

Both are used to denote candidate keys for a table.

You can only have one primary key for a table so would just need to pick one if you have multiple candidates.

Either can be used in Foreign Key constraints. In SQL Server the Primary Key columns cannot be nullable. Columns used in Unique Key constraints can be.

By default in SQL Server the Primary Key will become the clustered index if it is created on a heap but it is by no means mandatory that the PK and clustered index should be the same.

Share Improve this answer

Follow

answered Sep 29, 2010 at 9:15

Martin Smith

452k ● 94 ● 767 ● 870

What do you mean by "By default in SQL Server the Primary Key will become the clustered index if it is created on a heap"? – Gennady Vanin Геннадий Ванин Sep 30, 2010 at 15:12

2 @vgv8 - A heap is a table without a clustered index. If you run `ALTER TABLE dbo.tbl ADD CONSTRAINT PKNAME PRIMARY KEY (foo)` on a heap it will create a clustered index with the keys of the index being the PK columns. If the table already has a clustered index it will create a non clustered unique index. – Martin Smith Sep 30, 2010 at 15:17

Ah, it would have been more comprehensible to say, that clustered index can be only one on table and if there was no clustered index yet when PK was being created then PK is created clustered (making the table clustered, non-heaped), otherwise PK is CREATED non-clustered. Your phrase in answer was ambiguous because index, clustered or non-clustered, are always (created) in the B-tree, it is the actual table data might be on the heap if this data is non-clustered. Do I understand it wrong? – Gennady Vanin Геннадий Ванин Sep 30, 2010 at 17:55

7

A primary key is one which is used to identify the row in question. It might also have some meaning beyond that (if there was already a piece of "real" data that could serve) or it may be purely an implementation artefact (most `IDENTITY` columns, and equivalent auto-incremented values on other database systems).

A unique key is a more general case, where a key cannot have repeated values. In most cases people cannot have the same social security numbers in relation to the same jurisdiction (an international case could differ). Hence if we were storing social security numbers, then we would want to model them as unique, as any case of them matching an existing number is clearly wrong. Usernames generally must be unique also, so here's another case. External identifiers (identifiers used by another system, standard or protocol) tend to also be unique, e.g. there is only one language that has a given ISO 639 code, so if we were storing ISO 639 codes we would model that as unique.

This uniqueness can also be across more than one column. For example, in most hierarchical categorisation systems (e.g. a folder structure) no item can have both the same parent item and the same name, though there could be other items with the same parent and different names, and others with the same name and different parents. This multi-column capability is also present on primary keys.

A table may also have more than one unique key. E.g. a user may have both an id number and a username, and both will need to be unique.

Any non-nullable unique key can therefore serve as a primary key. Sometimes primary keys that come from the innate data being modelled are referred to as "natural primary keys", because they are a "natural" part of the data, rather than just an implementation artefact. The decision as to which to use depends on a few things:

1. Likelihood of change of specification. If we modelled a social security number as unique and then had to adapt to allow for multiple jurisdictions where two or more use a similar enough numbering system to allow for collisions, we likely need just remove the uniqueness constraint (other changes *may* be needed). If it was our primary key, we now also need to use a new primary key, and change any table that was using that primary key as part of a relationship, and any query that joined on it.

2. Speed of look-up. Key efficiency can be important, as they are used in many `WHERE` clauses and (more often) in many `JOIN`s. With `JOINS` in particular, speed of lookup can be vital. The impact will depend on implementation details, and different databases vary according to how they will handle different datatypes (I would have few qualms from a performance perspective in using a large piece of text as a primary key in Postgres where I could specify the use of hash joins, but I'd be very hesitant to do so in SQLServer [Edit: for "large" I'm thinking of perhaps the size of a username, not something the size of the entire Norse Eddas!]).
3. Frequency of the key being the only interesting data. For example, with a table of languages, and a table of pieces of comments in that language, very often the only reason I would want to join on the language table when dealing with the comments table is either to obtain the language code or to restrict a query to those with a particular language code. Other information about the language is likely to be much more rarely used. In this case while joining on the code is likely to be less efficient than joining on a numeric id set from an `IDENTITY` column, having the code as the primary key - and hence as what is stored in the foreign key column on the comments table - will remove the need for any JOIN at all, with a considerable efficiency gain. More often though I want more information from the relevant tables than

that, so making the JOIN more efficient is more important.

Share Improve this answer Follow

edited Sep 29, 2010 at 11:03

answered Sep 29, 2010 at 9:58

Jon Hanna
113k ● 10 ● 149 ● 257

5

Primary key:

1. Primary key is nothing but it uniquely identifies each row in a table.
2. Primary key does not allow duplicate values, nor `NULL`.
3. Primary key by default is a clustered index.
4. A table can have only one primary key.

Unique Key:

1. Unique key is nothing but it uniquely identifies each row in a table.
2. Unique key does not allow duplicate values, but it allows (at most one) `NULL`.
3. Unique key by default is a non-clustered index.

> This is a fruit full link to understand the ***Primary Key*** Database Keys. Keep in mind we have only

> one clustered index in a table [Talking about SQL Server 2005]. Now if we want to add another unique column then we will use ***Unique Key*** column, because Unique Key column can be added more than one.

Share Improve this answer

Follow

edited Jan 26, 2016 at 19:59

Jethro

988 ● 1 ● 10 ● 20

answered Sep 29, 2010 at 9:46

Rais Hussain

71 ● 1 ● 5

5 What do you mean by "is nothing"? – Jon Hanna Sep 29, 2010 at 10:02

3

A primary key is just any one candidate key. In principle primary keys are not different from any other candidate key because all keys are equal in the relational model.

SQL however has two different syntax for implementing candidate keys: the PRIMARY KEY constraint and the UNIQUE constraint (on non-nullable columns of course). In practice they achieve exactly the same thing except for the essentially useless restriction that a PRIMARY KEY can only be used once per table whereas a UNIQUE constraint can be used multiple times.

So there is no fundamental "use" for the PRIMARY KEY constraint. It is redundant and could easily be ignored or dropped from the language altogether. However, many people find it convenient to single out one particular key per table as having special significance. There is a very widely observed convention that keys designated with PRIMARY KEY are used for foreign key references, although this is entirely optional.

Share Improve this answer

Follow

answered Sep 29, 2010 at 21:37

nvogel

25.5k ● 3 ● 46 ● 87

1

Short version:

- From the point of view of database theory, there is none. Both are simply candidate keys.
- In practice, most DMBS like to have one "standard key", which can be used for e.g. deciding how to store data, and to tell tools and DB clients which is the best way to identify a record.

So distinguishing one unique key as the "primary key" is just an implementation convenience (but an important one).

Share Improve this answer

Follow

answered Sep 29, 2010 at 11:06

sleske

83.5k ● 38 ● 194 ● 235

Not true, in terms of database theory, a unique key may have more than null value. In SQLServer specifically this isn't allowed (considered a flaw by many), but that's SQLServer specific, not a db-theory thing. Because of this, a unique key isn't enough to identify a row uniquely as far as db theory goes. – Jon Hanna Sep 29, 2010 at 22:22

@Jon Hanna: In database theory a candidate key cannot contain null values. The term "unique key" is a tautology and/or a misnomer. All keys are unique and keys are also non nullable but if we understand "unique" in this sense to refer to a SQL-style unqiue *constraint* then we are talking about something which may include nulls and therefore isn't a "key" at all! So sleske is correct if "unique key" is understood to mean "candidate key". I would recommend avoiding the term "unique key" completely. It's too open to misinterpretation. – nvogel Sep 30, 2010 at 6:57

@dportas, no because a key can not cover some rows by being null. – Jon Hanna Sep 30, 2010 at 7:26

1 @Jon: You are not talking about the relational model but ER-modelling concepts. From the context my assumption was that we are talking about keys in the relational model. A key in the relational model is quite precisely defined as a set of attributes that unqiuely identifies every tuple in a relation. For confirmation see the Relational Database Dictionary, Codd's papers or his book, or numerous other database theory textbooks. This is fundamental because every relation by definition has at least one key and often more than one. – nvogel Sep 30, 2010 at 8:39

1 This is a little off-topic but for what it's worth here's an example of a 1 to 0/1 relationship in SQL: CREATE TABLE r (x INT NOT NULL PRIMARY KEY REFERENCES s (x), z INT NOT NULL UNIQUE REFERENCES t (z)); This is one example of a table with two keys. Arbitrarily one of them can be designated as the "primary" key. It doesn't matter which

one is called primary because all keys serve the same purpose - to uniquely identify a row. – nvogel Sep 30, 2010 at 8:56