Sql server database naming conventions best practices

What's in a [SQL] name? One of Java's great strengths, in my opinion, is that most naming conventions have been established by language creators. For example: Class names are in PascalCase Members' names are in camelCase Constants are in SNAKE_CASE If someone does not adhere to these conventions, the resulting code quickly looks non-idiomatic. You might also like: to format SQL CODE what about SQL correctly? SQL is different. While some people argue that MAJUSCULE IS THE RAPID: Others disagree on the correct case: It seems that there is a tendency towards writing identifiers in small letters with no agreement on the case of keywords. Also, in most dialects, people prefer snake_case for identifiers, although in SQL Server, people seem to prefer PascalCase or CamelCase. This is for style. And I would love to hear your opinion on the style and naming conventions in the comments! What about naming conventions? In many languages, naming conventions (identifiers) are not really relevant because the way the language designs the name, there is a relatively small risk of conflict. In SQL, this is a little different. Most SQL databases only support a set of 3-4 layer namespaces: Table Schema Catalog (or procedure, type) Column (or parameter, attribute) Some dialect-dependent warnings: While SQL Server supports both catalog and schema, most dialects support only one of them MySQL treats catalog (database) as Oracle schema supports a package space for procedures , schema and procedure In any case, there is no concept, would be package (scheme) hierarchies as there are in languages like Java, making namespacing in SQL quite complicated. A problem that can easily happen when you write stored procedures: function get_name (id NUMBER) is the number of the result; START SELECT NAME IN THE RESULT OF THE CLIENT WHERE ID = id; - Ehm... RETURN result; THE END; After you can see above, both the column CUSTOMER.ID and the GET_NAME.ID parameter can be resolved by the unqualified ID expression. This is easy to work around, but a boring problem to think about all the time. Another example is when you join tables, which probably have duplicate column names: SELECT * FROM client c Address JOIN an ON c.id = a.customer_id This query can produce two ambiguous ID columns: CUSTOMER.ID and ADDRESS.ID. In SQL language, it is mostly easy to distinguish between them by qualifying them. But in customers (e.g. Java), they are less easy to qualify properly. If we put the query in a view, it becomes even more complicated. Therefore, SQL and procedural languages are a in which a particular type of Hungarian notation might be useful. Unlike the Hungarian notation itself, where the data type is encoded in the name, in this case, we could encode another information in the name. Here's a list of rules I've found very useful in the past: 1. Prefixing Prefixing after Semantic Tables, views and other tab things can quickly conflict with each other. Especially in Oracle, if one does not simply create a scheme because of all the security hassles this produces (schemes and users are pretty much the same thing, which is nuts, of course. A schema should exist completely independent of a user), it may be useful to encode a schema in the name of the object: In addition, when using views for security and access control, you might have additional prefixes or suffixes to designate the viewing style: This list is clearly incomplete. I am undecided if this is necessarily a good thing in general. For example, should packages, procedures, sequences, constraints be prefixed? Often, they do not lead to ambiguities in the resolution of the namespace. But sometimes I do. The importance, as always, is to be consistent with a framework of rules. So once this practice is embraced, it should be applied everywhere. 2. Single or plural table name who cares. Just pick one and use it constantly. 3. Setting Standard Aliasing Another technique that We've found very useful in the past is a standard approach to aliasing things. We need alias tables all the time, for example, in queries like this: SELECT* from the client c join address an ON c.id = a.customer_id But what if we have to join the account as well? We've already used A for ADRESA, so we can't reuse A. But if we don't reuse the same aliases in each query, the queries start to get a little confusing to read. We could only use no aliases and always fully qualify all identifiers: SELECT* from the client JOIN address on customer.id = address.customer_id But that quickly turns out to be verbose, especially with longer table names, so also not very legible. The standard approach to aliasing things I've found very useful is to use this simple algorithm that produces four-letter aliases for each table. Given the Sakila database, we could fix: The algorithm to shorten a table name is simple: If the name does not contain an underline, take the first four letters, for example, the CLIENT becomes CUST If the name contains an underline, take the first two letters of each word, for example, becomes FIAC If the name contains two underlines , take the first two letters of the first word, and the first letter of the other words, for example, becomes FICD If the name contains three or more underlines, take the first letter of each word If a new abbreviation causes a conflict with existing ones, make a pragmatic choice This technique worked well for large schemas with more than 500 tables. You that abbreviations like FICD are meaningless, and indeed, they are, at first. But once you start writing a ton of SQL against this scheme, you start learning abbreviations, and they become significant. What's more, you can use these abbreviations everywhere, not just when writing joins: SELECT cust.first_name, cust.first_name, addr.city FROM customer cust JOIN address addr ON cust.id = addr.customer_id But also when aliasing columns in views or derived tables: SELECT cust.first_name AS cust_first_name, cust.last_name AS cust_last_name, addr.city AS addr_city from client cust JOIN address addr ON cust.id = addr.customer_id This becomes invaluable when your queries become more complex (say , 20-30 joins) and starts designing tons of columns in a library of views that you select from other points of view that you select from other points of view. It's easy to keep consistent, and you can easily recognize things like: Which table does a particular column come from If that column has an index that you can use (in a query against the view!) If two columns that look the same (i.e. FIRST_NAME) really are the same I think if you work with large-scale views (I've worked with 1000+ views schemes in the past), then such a naming convention is almost mandatory. Conclusion There is really no fair way to name things in any language, including SQL. But given the limitations of some SQL dialects, or the fact that, after joining, two names can easily conflict, I found the two above tools very useful in the past: 1) Prefixing identifiers with a hint about their types of objects, 2) Setting a standard for aliasing tables, and always aliasing column names accordingly. When you use a code generator, it would be jOOQ, the column names generated on the views will already include the table name as the prefix, so you can easily see what you're querying. I'm curious about your own naming conventions, looking forward to your comments in the comments section! Next Reading Nine of the most common mistakes in the Design database to properly format SQL Code What is in a name: Java Naming Conventions? There are only two difficult problems in Computer Science: invalidation of the cache and the naming of things. - Phil Karlton In this post I will enter the latter. Specifically, I will describe naming conventions for database objects, why they are so important and what you should and shouldn't do. Warning! This is a fairly opinionated post and I welcome feedback from people who suggest alternatives. Our target audience Our company, JackDB, uses PostgreSQL internally to store the persistent state and naming conventions in this post have been written with PostgreSQL in mind. Most recommendations must be equally valid for other relational databases, such as MySQL, Oracle, or Microsoft SQL Server. A lot of them will also apply to NoSQL databases, though not everything. De The suggestion below to use full English words is against the recommended approach to naming fields in MongoDB. When in doubt, find a guide to the specific database type. Why naming conventions are important names are long-lived data structures are meant to last much longer than the application code. Anyone who has worked on a long-term system can for that. Well-defined data structures and table layouts will live longer than any application code. It is not unusual to see a fully rewritten application without changes made to its database schema. Names are contracts Database objects are referenced by their names, so that object names are part of the contract for an object. In a way that you can consider the name of the table and columns of the database as the API of the database model. Once they are set, changing them can interrupt dependent apps. This is one more reason to call things properly before first use. Switching developer context With consistent naming conventions in the data model means that developers will have to spend less time searching for the names of tables, views, and columns. Writing and troubleshooting SQL is easier when you know that person_id must be a foreign key to the id field of the person table. Naming Conventions Avoid quotation marks. If you need to subpoena an identifier, then you should rename it. The iDs cited are a serious pain. Manually writing using quoted identifiers is frustrating, and dynamic SQL writing involving quoted identifiers is even more frustrating. This also means that you should not include white space in the name of identifiers. Ex: Avoid using names like First Name or All Employees. Lowercase. Identifiers must be written entirely in lowercase. This includes tables, views, column, and everything else. The name of the mixed case identifiers means that each use of the identifier will have to be quoted in double quotation marks (which I have already said are not allowed). Ex: Use first_name, not First_Name. Data types are not names. Database object names, especially column names, must be a noun that describes the field or object. Avoid using words that are only data types, such as text, or timestamp. The latter is particularly bad because it provides zero context. Emphasizes separate words. The object name that is composed of multiple words should be separated by underlines (for example, snake case). Ex: Use word_count or team_member_id, not word count, or word count. Full words, not abbreviations. Object names should be complete English words. Generally avoid abbreviations, especially if they are just the guy who removes vowels. Most SQL databases support at least 30-character names, which should be more than enough for a few English words. PostgreSQL supports up to 63 characters for identifiers. Ex: Use middle_name, not Use common abbreviations. For a few long words the abbreviation is both more common than the word itself. Internationalization and localization are the two that occur most often as i18n and l10n, respectively. In these cases, use the abbreviation. If in doubt, use the English word. It should be obvious if the abbreviation makes sense. Avoid using any word considered a reserved word in the database you are using. Use. there are so many of them, so there is too much effort to choose a different word. Depending on the context, reserved words may require a quote. This sometimes means you will write user and sometimes only user. Another benefit of avoiding reserved words is that highlighting the less intelligent editor's syntax will not highlight them incorrectly. Ex: Avoid using words such as user, lock, or table. Here's the list of words reserved for PostgreSQL, MySQL, Oracle, and MSSQL. Single relationship tables, views, and other relationships that hold data should have singular names, not plural names. That means our tables and points of view will be called teams, not teams. Rather than going into the relational algebra explanation of why this is correct I'll give you some practical reasons. They're consistent. You may have a relationship that has only one row. Is it still plural? They're unambiguous. Using only single names means that you don't have to cause nouns to pluralize. Ex: Does a Person object enter a People relationship or one people? How about an Octopus object? Octopuses? În Octopi? Octopods? Simple 4GL translation. Single names allow you to translate directly from 4GL objects into database relationships. You may need to remove some underlines and move on to the camel case, but the name translation will always be straight ahead. Ex: team_member becomes in the family class teammember mode or variable team_member in Python. Primary Key Fields Single-column primary key fields must be named ID. This means that when you write SQL you don't have to remember the names of the fields to join. CREATE TABLE person (id bigint PRIMARY KEY, full_name text NOT NULL, birth_date data NOT NULL); Some guides suggest prefixing the table name in the name of the primary key field, that is. person_id vs. id. The additional prefix is redundant. All field names in non-trivial SQL statements (i.e. those with multiple tables) must be explicitly qualified, and prefixing as a nameform is a bad idea. External Keys External key fields must be a combination of the name of the table referenced and the names of the fields referenced. For a single column of foreign keys (by far the most common case) this will be something like foo_id. CREATE TABLE team_member (team_id bigint NOT NULL REFERENCES team(id), person_id bigint NOT NULL REFERENCES person(id), CONSTRAINT team_member_pkey PRIMARY KEY (team_id, person_id)); Prefixes and suffixes (are bad) Relationship type prefixes Some (older) guidelines suggest naming tables with a prefix TB_, views with VW_ prefix or procedures stored with a SP_ prefix. The reasoning being that a programmer reading through some unknown SQL would immediately recognize this and know the type of object based on the name. That's a bad idea. Object names must not include the object type in them. This way can change it later. A view that is replaced by a table maintains the original contract of a view (for example: you can select from it). An dependent system should not be updated after such a change. I have seen many such systems where at some point a point of view will become a table. Then you will end up with the INSERT declaration issuing code in vw_foobar. There is even a very powerful feature of PostgreSQL, which allows you not to define DML rules on views (for example, you can INSERA/UPDATE/DELETE from them). Adding object prefixes adds additional typing now and additional confusion along the way. Application Name Prefixes Another suggestion (older) is to have a common prefix for all database objects. For example, our Foobar app would have the names of Foobar_Users tables, Foobar_Teams, etc. Again, this is a bad idea. All modern databases support some form of namespacing. For example, you can create schemas for grouping database objects in PostgreSQL. If you have multiple apps that share the same database and want to prevent them from clabbering each other, use schemas instead. That's exactly what I'm for! Most people won't need them, though. It is much more common for a database to be used with a single logical data model than more than several separate data models. Therefore, the schemes will not be necessary. If you need them, it should be pretty obvious. The exception to this rule is if you develop an agnostic database code database, it would be a frame or a plugin. Supporting multiple naming methods is complicated by so many frames instead of relying on prefixing the app name. However, most people develop apps, not plugins or frames, and their apps will dwell by themselves in a single type of database. Thus, there is no reason to add application name prefixes to all database objects. Data type suffixes Some guides (again, generally the oldest), suggest fixing column names with the data type of the field. For example, a text field for a name will be name_tx. There will even be extended lists to translate from data types to suffixes, text -&gt; tx, date -&gt; dt, etc. It's a bad idea! You can change the field data types. A date field can become a time stamp, an int can become bigint or numeric. Explicit name Some database commands that create database objects do not require a name to be specified. An object name will be generated either randomly (for example: fk239nxvnvsdvi) or by a formula (for example, foobar_ix_1). Unless you know exactly a name will be generated and you are satisfied with should be explicitly specifying names. This also includes the names generated by ORMs. Many default ORs to create indexes and constraints with long gibberish names generated. A few minutes of short-term time savings are not worth the headaches in remembering what fkas9dfks relates to in the long run. Indexes should be explicitly named and include the name of the table and the name (name) of the indexed column. Including column names makes it much easier to read through SQL explaining plans. If an index is named foobar_ix1 then you'll need to look for which columns cover the index to understand if it's being used correctly. CREATE TABLE person (id bigserial PRIMARY KEY, email text NOT NULL, first_name text NOT NULL, last_name text NOT NULL, CONSTRAINT person_ck_email_lower_case CHECK (email = LOWER(email))); CREATE INDEX person_ix_first_name_last_name ON person (first_name, last_name); Explain that the plans will now be easy to understand. We can clearly see that the index on first name and last name, that is. person_ix_first_name_last_name, use: =# EXPLAIN SELECT * From the person where first_name = alice and last_name = smith; INTEROGATION PLAN -- Index Scan using person_ix_first_name_last_name per person (cost=0.15.8.17 rows=1 width=72) Index Cond: ((first_name = 'alice'::text) AND (last_name = 'smith'::text)) (2 rows) Constraints similar to indexes, constraints should indicate descriptive names. This is especially true for verification constraints. It is much easier to diagnose a wandering insertion if the verification restriction that has been breached allows you to know the cause. CREATE TABLE Team (id bigserial PRIMARY KEY, text name NOT NULL); CREATE TABLE team_member (team_id bigint REFERENCES team(id), person_id bigint REFERENCES person(id), CONSTRAINT team_member_pkey PRIMARY KEY (team_id, person_id)); Notice PostgreSQL does a good job of giving descriptive names to key foreign constraints. =# \d team_member Table public.team_member Column | Type | -----------+--------+----------- team_id Modifiers | bigint | not person_id null | bigint | null indexes: primary key team_member_pkey, btree (team_id, person_id) Foreign key constraints: team_member_person_id_fkey FOREIGN KEY (person_id) REFERENCE person(id) team_member_team_id_fkey EXTERNAL KEY (team_id) REFERENCE Team (id) If we try to enter a row that violates one of these constraints we immediately know the cause only on the basis of the constraint name: =&gt; INSERT INTO team_member(team_id, person_id) ERROR: Insert or update in the team_member table violates the external key restriction team_member_team_id_fkey DETAIL: Key (team_id)=(1234) is not present in the team table. Similarly, if we try to enter an email address that is not smaller in the table of people created above, we will receive a restriction violation error that tells us exactly what is wrong: -- This insertion will work: =&gt; INTRODUCTION IN person (e-mail, first_name, last_name) VALUES (alice@example.com, Alice, Anderson); INSERT 1 -- This insertion will not work: =&gt; INSERT INTO person (e-mail, first_name, last_name) VALUES (bob@EXAMPLE.com, Bob, Barker); ERROR: New row for relationship person violates check person_ck_email_lower_case DETAIL: No row contains (2, bob@EXAMPLE.com, Bob, Barker). Final Thoughts If you are starting a new project, then I suggest you follow the conventions presented here. If you're working on an existing project, then you need to be a little more careful with any new objects you create. The only thing worse than bad naming conventions is several naming conventions. If your existing project already has a standard approach to naming its database objects, then continue to use it. Do you have anything to add to this list, a way to improve some of these guidelines, or just think some of these are terrible? Let me know! Know!