Database Best Practices

There are things which some believe are wise to do with databases. As with every set of so-called "best practices", there will be disagreement. Some of these tips are subject to considerable debate, and every side of these debates has a wealth of evidence to support it. Therefore, the best advice to the working database practitioner is to be aware of the controversies surrounding each tip, and to apply rational judgement and scientific evaluation to every real-world case.

These tips are generally intended to apply to a SQL database, which is typically (and perhaps erroneously) called a [RelationalDatabase](http://c2.com/cgi/wiki?RelationalDatabase). They may or may not apply to other paradigms.

**The following practices are generally agreed-upon:**

* Always have a [PrimaryKey](http://c2.com/cgi/wiki?PrimaryKey). Always. Generate it if there isn't one that makes sense. This is a requirement of being "relational," according to [DrCodd](http://c2.com/cgi/wiki?DrCodd), though modern research from [ChrisDate](http://c2.com/cgi/wiki?ChrisDate) and others has reduced the requirement to having at least one [CandidateKey](http://c2.com/cgi/wiki?CandidateKey). As a [PrimaryKey](http://c2.com/cgi/wiki?PrimaryKey) is simply a [CandidateKey](http://c2.com/cgi/wiki?CandidateKey) that has been given preferential status - either arbitrarily or for psychological reasons - these are effectively the same requirement.
* Never make a change to the schema directly if you **ever** plan to have more than one schema deployed in the field, or if you maintain parallel environments separate from the production environment. For example, you might maintain a load testing environment and a QA environment, in addition to the production environment. Use a script that is retained in an on-going log so the same change can be applied to other schemas. When you make changes directly to the schema you may bypass some critical steps put in place to ensure schema integrity, as well as running a risk of losing synchronization between the schemas.
* If you have a table with multiple records inserted per "entry", have a unique identifier for the "entry" whether there is a separate table to store the entries or not. Huh? this seems to imply duplicate rows which in a properly normalized database should never happen if you are following best practices.No it doesn't, it says there should be a column grouping these records, such as days of the month being grouped by month.
* Use consistent column name suffixes and prefixes. This considerably simplifies generation of automated SQL.
* Be a part of the [VowelGeneration](http://c2.com/cgi/wiki?VowelGeneration), or at least create good [RelationalDatabaseFieldNames](http://c2.com/cgi/wiki?RelationalDatabaseFieldNames).
* Don't use short column names. It may be very tempting to name a column "id", but just try searching through your code for all references to that column after it is in there 100 times. Always use a [MeaningfulName](http://c2.com/cgi/wiki?MeaningfulName). Some issues about column naming can be found in [HowImportantIsLeanCode](http://c2.com/cgi/wiki?HowImportantIsLeanCode).
* Never make untested changes to a production database.
* Always carefully design the database and follow excellent normalization practices. Once a database is in production it can be dangerous and expensive to change it. Good normalization will define domains which can easily be mapped to objects. Beware of using a spreadsheet as the basis for a table! It may not be normalized.
* As a general rule of thumb, if a table contains more than 20 fields it may be improperly normalized. Obviously, this needs to be carefully examined on a case-by-case basis.
* If you are evaluating a database-based software product and the vendor allows you to see the schema, see if the database is poorly normalized. If so, you may wish to consider finding another vendor. A badly designed database usually means badly designed, brittle and buggy software. Lack of quality in one usually reflects lack of quality in the other.
* VIEWs are your friends. They can be used to isolate database changes from the application, reducing the need for maintenance if the database changes.
* Where possible, stick to the ANSI core SQL and avoid vendor specific 'extensions'. However, this really depends on whether or not you eventually may need to deploy to a different RDBMS. Some argue that if you know you won't have to change DBMSs, then you can make full use of the power of the database. Don't spend time that could be used improving your code by reinventing what you just purchased. The best practice in this case is to carefully consider whether or not you will use vendor specific extensions.
* Divide operating system, data and logging onto different physical disks with different channels on the disk controllers for maximum performance. If your database is large enough, use another device for indices.
* For better SELECT performance, index your JOIN columns.
* Use RAID, use RAID 0+1 or 1 for maximum performance. RAID 5 may be too slow for some applications.
* [ContinuousDatabaseRefactoring](http://c2.com/cgi/wiki?ContinuousDatabaseRefactoring) might be a good idea.
* One word: Backups.

**The following best practices are highly controversial:**

* A previous author wrote: "Generally, avoid [*PrimaryKey*](http://c2.com/cgi/wiki?PrimaryKey)s that have meaning. Names are not unique, and many seemingly unique identifiers such as Social Security numbers actually are not, due to real-world data reliability problems." In short, this is a recommendation to always have an auto-generated (typically numeric) SurrogateKey[?](http://c2.com/cgi/wiki?edit=SurrogateKey) instead of a domain-based LogicalKey[?](http://c2.com/cgi/wiki?edit=LogicalKey). This is a rather pat answer to a complex issue, though it is one that will suffice in a number of cases and is at least preferable to having no [PrimaryKey](http://c2.com/cgi/wiki?PrimaryKey) at all. The controversy is addressed in [AutoKeysVersusDomainKeys](http://c2.com/cgi/wiki?AutoKeysVersusDomainKeys).
* Avoid over-using [StoredProcedures](http://c2.com/cgi/wiki?StoredProcedures). See [BusinessLogicInStoredProcedures](http://c2.com/cgi/wiki?BusinessLogicInStoredProcedures).

**The following "best practices" seem to be spurious, but have been retained to encourage further research in case there is some practical basis for these that the authors have failed to mention:**

* Don't use Sybase. (Why not?)
* Don't hire consultants (Oracle or otherwise) to build your product for you.

It has been recommended that the above be reformulated in [PatternForm](http://c2.com/cgi/wiki?PatternForm).

**Creation Scripts**

For example: To version a database, a person can make SQL scripts and store them under [VersionControl](http://c2.com/cgi/wiki?VersionControl) (such as [SubVersion](http://c2.com/cgi/wiki?SubVersion), sccs, [ConcurrentVersionsSystem](http://c2.com/cgi/wiki?ConcurrentVersionsSystem), [StarTeam](http://c2.com/cgi/wiki?StarTeam) and so on). The scripts would reflect how to make the tables, procedures, triggers, and so on.

Yes, but versioning SQL creation scripts is insufficient for an [ExtremeProgramming](http://c2.com/cgi/wiki?ExtremeProgramming) project. This is especially true with respect to releasing to production at the end of every iteration.

This of course raises the question as to what **is** sufficient.

At least, SQL creation scripts is the absolute minimum required to properly version a database. Another SQL scripts to fill-in initial data is another must if there are rows to be filled. Export/import in many database don't always work across different platforms/codepages, much less across different brands. I have even created script generators to use the same text source to generate SQL scripts for different brands of DBs (due to different names for datatypes, etc). If you need to version data stored in a running database, perhaps you will need to create custom export/import scripts/programs. -- [*OliverChung*](http://c2.com/cgi/wiki?OliverChung)

**Actually no.** A database is best viewed as a [ImageBasedLanguage](http://c2.com/cgi/wiki?ImageBasedLanguage), therefore maintenance of creation scripts can be a big waste of time as well as a significant risk, modifications are done against the live data (like ALTER TABLE ADD COLUMN , and so on, so forth). Serious database products allow DBAs to do version management of metadata in the database itself, for example Oracle Change Manager, allows one to automatically save metadata at some point in time, do a diff against two live schemas or against a live schemas and one saved in repository or against two saved in repository, generate a migration script from a diff (for example generate the migration script so that the QA database reaches the same schema as the development database). It only make sense that if metadata is stored in tables, metadata versions should also be stored in tables and the power of a relational database should be used for change management. Most of the databases are able to reverse engineer the creation script on auto-pilot, which is nice as an insurance policy, but that generated script is not good for change management (because the order of object creation is not guaranteed, a diff between 2 CVS versions of the creation script is practically unusable).

The illusion that the creation script is the work product of the database design activity and supposed to be the focal point of change management generates extreme and gratuitous complications. For example: how do you test this script ? You either have to drop the development database and recreate it, or create a parallel database (and it does take time), you add a simple column and maybe recompile some stored procedures, you see the effect immediately. Now you have the redundant activity to maintain the script (but what if you make a typo in the script), let's test the script => it may take a lot of time. This creation script is unapplicable against a live production database (maybe you can get away with, recreating the QA database but not the production environment). --[CostinCozianu](http://c2.com/cgi/wiki?CostinCozianu)

May I disagree with you? We were 15 people working on a product for over a year. We had a strict policy of reinstalling the database schema (drop db, create db, run all creation scripts. all this done using ant resetupdb) after any svn checkout. We never had an issue on our development machines. On the contrary, work was less and less every day, yet the product was fully functional every day. Bugs were never checked in because they were detected early. When we delivered the project to our customer, the database admin was totally shocked to find out we would run scripts on his db, so he forced us to write documents for every change. Since we changed everything daily, but delivered every 1 or 2 weeks, the document was almost always 40 to 100 pages long. He wanted to do everything manually, but later he realized he couldn't, so he run every little script by hand. The result was that he messed the database up and we couldn't for sure know the state of the database in production.

That was a very problematic issue. Because sometimes he would stop running the scripts after the first failure, but wouldn't roll back. On other occasions he would happily continue with the next script, even if the scripts were depedent on each other. Needless to say, we were unable to repro the problem on our testing databases. End result? We had to wipe out the database and use strictly ant resetupdb (delete all data) and ant upgradedb (upgrade database without loosing data). All changes would never fail from then on. -- [GuillermoSchwarz](http://c2.com/cgi/wiki?GuillermoSchwarz)

**Too Many Small Tables**

I am bothered by (1:1)-to-(0:1) relationships. It creates lots of skinny tables, more joins, and creates change-problems. For example, if managers have certain fields only for managers, but later \*all\* employees get some of the same perks, then moving those fields back into Employees instead of Managers creates head-aches. Thus, I suggest keeping it all together. It is more ChangeFriendly[?](http://c2.com/cgi/wiki?edit=ChangeFriendly).

Some say keeping those together violates normalization rules. If so, then the normalization rules are wrong on that issue IMO. Perhaps the rules were shaped back in the days when field sizes were physically fixed in size. Empty fields should not take up any storage in most modern systems. Lack of an entry for a physical record can be translated as an empty value (null, blank, or whatever). A field dictionary makes it so that the DB doesn't have to keep the name of each field in each record.

I think "table per subtype" is an anti-pattern. See "parking space" discussion in [SparseColumns](http://c2.com/cgi/wiki?SparseColumns).

1:1 relationships indicate a schema problem. 0:1 relationships are a regular part of a normalized database. They allow you to avoid NULLs, and write more efficient queries.

Schema refactoring is often difficult, but avoiding 0:1 relationships won't save you much work. When you decide that all employees have those fields, you've changed the database constraints, and you'll still have to examine all queries to make sure that they do the right thing.

I am not sure what you mean. Please clarify. Empty/null cells are not necessarily a bad thing compared to the alternative. They don't have to take up physical space, depending on the implementation.

The empty/null columns issue isn't about space in this case. It's about rules. Take for example an Employee table that avoids a 0:1 relationship and has the following fields: EmployeeId*[?](http://c2.com/cgi/wiki?edit=EmployeeId)* INT, Name VARCHAR(100), ManagerId*[?](http://c2.com/cgi/wiki?edit=ManagerId)* INT /\*self ref to EmployeeId*[?](http://c2.com/cgi/wiki?edit=EmployeeId)*\*/, ProfitSharingAmt*[?](http://c2.com/cgi/wiki?edit=ProfitSharingAmt)* MONEY. Lets say you define a business rule that only Managers (in this case defined as a row that is referenced by another employee row via ManagerId*[?](http://c2.com/cgi/wiki?edit=ManagerId)* -> EmployeeId*[?](http://c2.com/cgi/wiki?edit=EmployeeId)*) can have a value in ProfitSharingAmt*[?](http://c2.com/cgi/wiki?edit=ProfitSharingAmt)*. There's no easy way to enforce this in your database because you've created a row level rule as opposed to a table level rule. Employees who aren't managers could have a non null value in ProfitSharingAmt*[?](http://c2.com/cgi/wiki?edit=ProfitSharingAmt)* because your database can't easily stop someone via a simple constraint from putting something there. You could potentially try to use triggers or some other means to attempt to enforce row level rules, but RDBMS's are designed to work easiest with table level rules like constraints, data types, and nullability settings. The problem with avoiding 0:1 relationships is that basically, you have columns that may or may not apply to a row depending on what data is in that row. If you follow that track, you could add any number of additional columns that only work for certain types of data - hourly employees vs. salaried - consultants vs. regular employees, etc. Then, you have to somehow keep track of all those rules. If you separate them out in your model, then those rules are always enforced and they are instantly obvious from looking at a schema diagram. Remember, the number one concern with a database is data integrity. Concerns about performance, space, and query complexity are all worthless if your data is garbage.

I don't really see a problem with triggers that check such things. Sure, they are not as clean as using basic referential integrity, but that is because life is not simple. It is not realistic to expect referential integrity to enforce every business rule.

{My experience has been that basic referential integrity can enforce a far larger subset of typical business rules than most people think, given a normalized database schema. -- [*DanMuller*](http://c2.com/cgi/wiki?DanMuller)}

Perhaps, but in my opinion one should focus on making tables easy to grok and manage rather than let "mechanical" issues such as simpler referential integrity be the primary guide. Machines should serve humans, not the other way around.

**"Ref" Convention**

I like the convention of using xID for primary field and xRef for foreign keys. It simplifies the SQL join syntax and makes it easier to identify the primary keys IMO. For example, "customerID" in Customers table, and "customerRef" in Invoices table. Comments?

Why not call the foreign key customerID, too? Works for me, at least.

First, it is hard to tell what is the primary key. Second, "ref" simplifies the JOIN syntax because the keys don't have to be dot-qualified as often. Third, it reduces SQL parser/processor complaints about "ambiguous column name" when joins bring together two or more different tables with same-named ID columns.

Better still, if you call them both CustomerID (following your naming standard), then you get natural joins for free, as in:

SELECT \* FROM Customer NATURAL JOIN Order WHERE CustomerID = 23

And the best part is that it's not ambiguous, because the result of a natural join contains only one copy of the shared column names. (At least it works that way on [*PostgreSql*](http://c2.com/cgi/wiki?PostgreSql).)

Is this standard SQL? I don't see it my docs, which are admittedly a bit old.

Yes, it's standard SQL, the "new" syntax for joins. Maybe you want to update your docs...

Does the above "ref" convention make this not work? I would hate to think it uses field names to match. Even if you don't like the ref convention, relying on naming could hinder such joining for older tables that did not use either convention.

I strongly agree with the above comments about using the same column name for keys and referring foreign keys, to simplify natural joins. "If it means the same thing in two tables, give it the same name." The guideline has problems when a table can have multiple foreign keys referencing the same foreign table, or when you have self-referential tables, but in general, it's a good guideline that also makes the schema easy to read and understand. The cost is that for other SQL joins (unnatural joins?), you have to use explicitly-qualified field references. To ease this, I always define short referents for each table mentioned in a join, e.g. T1, T2, etc.

Perhaps it is a personal preference. I am more comfortable with the "ref" convention. If you can show some objective faults with it, I would be happy to take a look.

I (another I) often prefer naming the foreign key for something a bit more clear. Sometimes you must. Say, for example, you have an postal\_address table. Your invoice table references it twice: don't billing\_address and shipping\_address make the best sense? I even see this in situations where there's only one reference, though I can't think of any examples right now.

Unfortunately, this is a classic example of silliness in the SQL standard. This breaks NATURAL JOIN, because NATURAL join is looking at column names instead of the constraints that clearly indicate what two columns should be joined, regardless of name (because shipping\_address REFERENCES postal\_address (postal\_address\_id)).

The "ref" convention doesn't extend to the case when the primary key is also a foreign key, or when the primary key is compound and part of it is a foreign key, or when the primary key and the foreign key are both compound and share attributes, or when there are multiple foreign keys one or more of which is compound and ... you get the idea, I'm sure.

I am not sure what you mean by the first sentence. Do you mean within the same table? I agree there are situations where it does not work well, but generally anywhere there is an "ID" or "No" (number) kind of primary key, it will work smoothly.

**Duplication of Address and Contact Information**

Rather than repeat address (location) info in many tables, have a Contact table and reference that instead. ([ContactAndAddressModels](http://c2.com/cgi/wiki?ContactAndAddressModels))

This sounds more like a simple example of applying the [*NormalForm*](http://c2.com/cgi/wiki?NormalForm)s, which is probably a [*BestPractice*](http://c2.com/cgi/wiki?BestPractice). Maybe I'm misunderstanding? - [*NathanielEliot*](http://c2.com/cgi/wiki?NathanielEliot)

Yes, but it is a common mistake even among those who know about normalization rules for some reason. Part of the problem is that some wish to implement it using inheritance (incorrect solution IMO), and since current RDBMS don't have schema inheritance, feel they have to replicate address fields instead.

Don't confuse structural similarity with similar meaning. Logical schema design is determined by what the data means, not what it looks like. It can be perfectly legitimate to replicate the address structure in different tables. The only reason to put them together is if there's a need to have a distinct concept of 'addresses' in your database.

A somewhat similar counter example is templates. I work on a bookkeeping application, and we have many cases of user-definable templates for something, e.g. paychecks. The templates are of course structurally similar to the paychecks -- but the template components do not belong in the same tables -- that complicates all sorts of queries. (Some older parts of our application do this, and it's a nightmare.)

Ideally, database systems would allow for complex user-defined data types. Then you could define a multi-part "address" data type and use it as an atomic data type wherever needed. And of course you can emulate this behavior in the physical database schema by doing what you describe -- but realize that this is a workaround for DBMS limitations, and don't let it confuse the logical design.

I am not sure what you propose here. I would have to look at a specific [UseCase](http://c2.com/cgi/wiki?UseCase). Personally, I am not a heavy believer in "types" ([ThereAreNoTypes](http://c2.com/cgi/wiki?ThereAreNoTypes)). User-defined templates perhaps should be implemented via a [ControlTable](http://c2.com/cgi/wiki?ControlTable) or an [AttributeTable](http://c2.com/cgi/wiki?AttributeTable).

No, it [factoring address fields to a separate Contact table] is \_not\_ an example of [*DatabaseNormalization*](http://c2.com/cgi/wiki?DatabaseNormalization) (no [*NormalForm*](http://c2.com/cgi/wiki?NormalForm) requires such factoring). As hinted at above, its not even clear what, if any, logical advantages it has over putting the address fields with each addressee entity/table, so the claim that it is a [*BestPractice*](http://c2.com/cgi/wiki?BestPractice) is rather dubious, too. (Assume for the sake of argument that you are using a variant of SQL with user-defined types or some sort of template/macro expansion support in DDL and DML so that the definition of the address fields is consistent across all the places it is used and is defined [*OnceAndOnlyOnce*](http://c2.com/cgi/wiki?OnceAndOnlyOnce).)

**View Columns**

It would be nice if more RDB vendors supported "column views". Table views are often too large a granularity in my opinion. Column views could allow you to provide virtual columns (such as moving a column to another table, but having a read-only place-holder) without diddling with table name-spaces, reducing the impact of change to code.

**Reserved Word Overlap**

I sometimes prepend "the" to common words when making column or table names. Example "theValue". It avoids clashing with reserved words. I have ran into column or table names that confuse SQL parsers due to overlapping with reserved words. By appending "the" to titles like "value", "description", "timestamp", etc., conflict is reduced and portability improved. I have ran into such overlaps more than once.

Uh, what is wrong with SyBase*[?](http://c2.com/cgi/wiki?edit=SyBase)*? (per above list)

I'm probably not the right person to answer this, but SyBase[?](http://c2.com/cgi/wiki?edit=SyBase) seems to have an astounding ability to totally misoptimize any query that joins, say, five tables or more (be they joined by subquery or explicit join).

**Category Per Column Rigidity**

I see a fair amount of problems caused by hard-wiring classification systems into database columns. It seems to me that a separate chargeCategory table may be more appropriate so that new charge categories can be added without changing schemas and code. However, this may be sacrificing efficiency because it would result in more joins. I would like to hear other's opinions on this. Example:

Before:

Table: customerCharges

---------------

customerRef

billDate

regularLabor // amounts....

overtimeLabor

approvedParts

unapprovedParts

managementFees

otherCharges

...etc...

After:

Table: customerCharges

----------

customerRef

billDate

categID // mnemonic or numeric category code

Amount

Table: chargeCategory

----------

categID // primary key

categDescript // example: "Overtime Labor", "Management Fees", etc.

--top

**Column Names in Application Language**

Just to offer another idea of naming tables and columns: I've been using the [JavaLanguage](http://c2.com/cgi/wiki?JavaLanguage) Class and field convention for a while and it works pretty well. For example, Customer table, [id] column, references Part table, [id] column using [partId] foreign key. When doing join, I'd use

from Customer c

join Part p on c.[partId] = p.[id]

The advantage being that it minimizes redundancy in naming, and when you do refactoring, even renaming class and table does not affect column name. And it still reads quite well: customer's partId equals to part's id.

The problem I see is that it risks name collisions when many tables are involved. Some SQL dialects have fits easily over such. True, there are usually work-arounds, but it is more SQL coding.

**Avoid Minor Variations as Table Division Criteria**

For example, there are probably not enough differences between cars and trucks to make separate Car and Truck tables (and perhaps some vehicles that could be classified either way). There should be a single Vehicle table. If there are lots of columns that only trucks or only cars have (and likely to stay that way over the long run), then make a separate Truck\_features and/or Car\_features table with a foreign key to the Vehicles table. With the "tall cars" and "short SUV's" coming out these days, I would be reluctant about hard-wiring any vehicle feature to any schema.