Dispensing with DDL – an account from having tried

DBMS systems are typically made to work using two distinct languages: one to control the database's structure, and one to control its content. In SQL systems, these languages are typically labeled “DDL”, and “DML”, respectively, those names obviously reflecting the “Definition” and “Manipulation” to which they are targeted. Note that the property and/or distinction is not unique to SQL systems. The IDMS system, for example, had languages called “Device Media Control Language” and “Schema Language”, which allowed to define and prepare the physical files in which the system's data were to be recorded, and the actual database structure, respectively.

Codd's 4th rule

SQL installations typically contain a component called “the catalog”. The term refers generically to “a system-provided database containing information about all the databases in the installation”. Note that “all the databases in the installation” is not limited to just the databases defined by the user of the installation, but obviously includes that catalog itself too.

The feature of a “system catalog” is typically seen as a consequence of a system having to comply with the 4th of Codd's famous 12 rules, which says that “The system must support an online, inline, relational catalog that is accessible to authorized users by means of their regular query language. That is, users must be able to access the database's structure (catalog) using the same query language that they use to access the database's data.”.

The rule has been formulated in many different ways, sometimes giving rise to some subtle differences in what is actually being said. For example, http://www.cse.ohio-state.edu/~sgomori/570/coddsrules.html gives the following definition : “The database description is represented at the logical level in the same way as ordinary data, so authorized users can apply the same relational language ***to its interrogation*** as they apply to regular data.”.

Note how this formulation explicitly limits the use of the catalog to “interrogation”, thus ruling out “updating”, at least directly. Such a definition seems to take it for granted that a user will not want to “update the catalog directly”, and “that it is only obvious” that the user will be using some other language to do that.

In what follows, we will object to the “obviousness” of that assumption and explore the consequences for (an implementation of) a DBMS engine.

Cause and consequence

Typically, maintaining the catalog is done as a side-effect of the DDL by which the databases are defined. A user issuing a `CREATE TABLE` statement, also causes an `INSERT INTO SYSTABLES VALUES …` to be executed as a consequence. It is often the case the user will be prohibited from issuing that same insert statement himself, and will be left no other option than to issue the `CREATE TABLE` instead.

The most direct consequence of this is that there is an extra language/grammar to master. And there is an extra language/grammar to be interpreted by the system. Extra parsers and compilers are to be built and maintained. Special measures have to be implemented to make the catalog read-only, and updatable by the system only.

An alternative

As an alternative, we propose to dispense with all the 'dedicated' DDL syntax, replace it with 'catalog DML' (CDML for short in the remainder of the paper) and see what happens. For the user as well as for the implementor.

As a first point, we would have to demonstrate that CDML is a viable alternative to DDL. That is, that at least everything that can be done through a DDL, can also be done through some CDML. For that, it is needed that there are no DDL statements to which no CDML at all corresponds. But if there were such a DDL statement, that would mean that whatever aspect of the database structure is managed by such a DDL statement, is then actually undocumented in the catalog, and is thus not queryable for the user, which violates Codd's 4th rule. So it seems reasonable to assume that in a relational system, every DDL statement will always cause *some* CDML to be issued as well.

Since, in the 'usual approach', the DBMS engine is the 'triggering instance' for the CDML, and the CDML gets computed from the DDL, the CDML can mathematically be regarded as a function of the DDL. If no other stuff is taken into account (in the process of computing the CDML), then we can therefore conclude that exactly one CDML statement corresponds to any possible single DDL statement.

If, otoh, the same DDL could cause one of multiple distinct possible effects, based upon e.g. the current state of the database/catalog, then more than one CDML statement could correspond to a single DDL statement. But it seems reasonable to dismiss this possibility on account of its being “bad language design” (the same command doing different things depending on 'external circumstances' just doesn't look right, certainly not in the context we're dealing with).

Finally, could there be multiple distinct DDL statements that all correspond to the same CDML ? To answer that, observe that, just like any other relvar in the system, the catalog relvars too have some external predicate that "formally" defines its meaning to the world outside. For example, the catalog relvar SYSTABLES might have an external predicate such as "the system has a table called §tablename§". The CDML statements deriving from some DDL, thus actually define the semantics of the DDL statement, since those CDML statements obviously assert the truth or falsehood of certain instantiations of those catalog relvar predicates. Now if a given CDML might derive from multiple distinct DDL statements, that would mean that there are multiple DDL statements sharing the very same semantics. Meaning the DDL language exposes redundancy, and the redundant parts can be removed without loss of functionality.

For all those reasons, we can assume that a well-designed DDL language, and the CDML that it "generates", are isomorphic in nature (the claim is too strong, but I've got no better term at this point).

Finally, one might wonder whether there are CDML statements to which no DDL statement corresponds. If there are, then this would make a CDML language strictly more powerful than a DDL, which would be in itself a justification for preferring CDML over DDL. Two cases come to mind which seem to give CDML an advantage:

- The system is already required to support set-level updates. Using this feature in CDML automatically makes for the counterpart of things like a "create multiple tables" statement in the DDL language.
- The system is already required to support multiple assignment. Using this feature in CDML automatically makes for the automatic availability of "multiple DDL", that is, the possibility to issue any arbitrary combination of "individual DDL statements" as a single ("atomic") database control operation.

Contrast this with, e.g. SQL's `ALTER TABLE` statement, which does not allow for the simultaneous addition of more than one column, or for the simultaneous addition of a column and a key:

```
<alter table statement> ::= ALTER TABLE <table name> <alter table action>
<alter table action>    ::=
         <add column definition>
     |   <alter column definition>
     |   <drop column definition>
     |   <add table constraint definition>
     |   <drop table constraint definition>
```

Yet another advantage of resorting to CDML for database control instead of DDL, is the issue that (Tutorial) D has solved by introducing the D_INSERT and I_DELETE operations. These days, on database fora you often run into questions such as "I need to set up an installation script that also creates the database, but that installation script cannot fail if the database already exists. So I need a sort of CREATE TABLE IF NOT EXISTS. How can I do that ?". Well, the CDML we propose does exactly that. No extra options need to be added to the DDL language because "`INSERT INTO SYSRELVARS ...`" is already readily available.

As a rationale for wanting to try and achieve database control through CDML, that should already do pretty well. Hence, onto some considerations on "standardisation".

Can such a language be “standard” or “standardized” ?

Hmmmmmm. Kind of a vicious question. The answer is “not any more or less than any DDL as we currently know it can be”.

It will be clear that in a CDML language replacing, e.g., SQL DDL, a `"CREATE TABLE ..."` statement will be replaced by something that looks syntactically like
`"INSERT INTO SYSRELVARS … ;"`, or even like

```
"INSERT INTO SYSRELVARS …,
 INSERT INTO SYSRELVARATTRIBUTES …,
 INSERT INTO SYSRELVARKEYS …,
 INSERT INTO SYSKEYATTRIBUTES … ;"
```

It will be clear that there is a correspondance here between DDL language keywords and catalog relvar names here. The language keyword “`TABLE`” corresponds to the catalog relvar name “`SYSRELVARS`”. And so on and so on.

And then of course the question arises whether the implementation cannot/should not be free to decide on its own relvar names in the catalog ? Well, the answer here is that the benefit of standardized catalog relvars has already been recognized, even by SQL[1], and that by adhering to/complying with such standards, a CDML language based on such standardized relvar names is indeed no more or no less “standardized” than a DDL such as SQL's is already.

Furthermore, even with a standard such as SQL DDL in place, there are still problems left. For example, SQL DDL only provides a standard syntax to define the logical structure of a database relvar. But at least one SQL DBMS did not allow to define any relvar unless certain physical design details were also immediately given. Standards or not, this resulted in vendor-specific extensions (such as a `STOGROUP` option in the `CREATE TABLE` statement) which were required to make a statement work!

Consequences for the user

It will be clear that the user now needs to master “another” language. One that might be said to have less keywords, and that might therefore be easier to learn or so. And all the more so because the keywords are exactly the same the ones he will also need when it comes to updating his own databases …

A similar argument can be made regarding the catalog relvar names (and structure) the user will be required to memorise: there is quite a chance that the user will know those names already, because a user who updates the catalog, is very likely to also query it at some point in time, and doing that obviously requires knowing the relvar names already! So we conclude that a switch from DDL-as-we-know-it to CDML-as-proposed-here need not imply a huge additional intellectual effort on the part of the user, as far as “mastering the language” is concerned.

[1]The INFORMATION_SCHEMA portion of the standard, specifically.

Processing CDML

If we dispense with DDL and its keywords such as "`ALTER TABLE … ADD KEY …`", then it is clear that whatever processing was "hooked onto" the DDL keywords, must now be "hooked onto" the catalog updates that are done. If a (catalog tuple for a) new key is inserted, then it must be verified that current data does not violate the database constraint corresponding to that key. If a (catalog tuple for a) new physical storage file is inserted, that file must also be created (and possibly formatted and so) in the machine's/OS's file system. If a database constraint is altered, then certain objects within the engine's runtime environment representing/holding that database constraint in compiled form, must be invalidated and replaced. Etc. etc. etc.

In general, we can assume that there might be an additional action to perform for each type of update to each individual catalog relvar. That action might be a no-op, however. Anyhow, a multi-assignment to the catalog thus gives rise to a set of "additional actions" to be performed. One such set might be, e.g. {`ADDRELVARACTION ADDKEYACTION`}[2], in the case where a new relvar is defined, and another such set might be {`ADDKEYACTION`}, in the case where a new key is added for an already existing relvar. This example shows that the "activity" which must be performed by some individual action in the action set, may depend on the presence or absence of other actions: if our `ADDKEYACTION` occurs in the context of a new relvar being defined, then obviously no checking needs to be done to verify that current data doesn't violate the new key (how could it ?), while in the other case this checking activity must effectively be performed.

Consider another example set of "additional actions", consisting of :

- An `ADDRELVARATTRIBUTEACTION` reflecting the addition of an attribute to an existing relvar,
- An `ADDKEYACTION` reflecting the addition of a key to that same relvar,
- An `ADDKEYATTRIBUTEACTION` reflecting the fact that the new key consists of the new attribute,
- An `ADDRECORDATTRIBUTEACTION` reflecting the physical implementation details pertaining to our new relvar attribute,

In this scenario, the actions regarding the logical structure of the relvar, will require a new key constraint to be verified. However, that constraint (and thus also its corresponding relational expression that will have to be evaluated) involves an attribute that doesn't exist yet at the beginning of the processing ! Evaluating the relational expression corresponding to the new constraint will be a bit hard if the physical implementation details concerning the new attribute have not been properly completed ! In many cases, it will turn out that there is a need for "prioritising" between the several system actions that need to be carried out.

[2]Where both actions concern the same relvar, of course ...

Validating CDML

Every DDL operation comes with a set of "business rules" that have to be complied with, or otherwise the DDL operation is obviously invalid. In typical DDL, these business rules are enforced using "dedicated code". However, it will be intuitively clear that a great many of these rules are really nothing more than "just another database constraint on the catalog". Hence, declarative support for (enforcing) database constraints of arbitrary complexity can come in very handy[3] when a DBA is defining databases that are supposed to conform to the engine's prescriptions.

For example, if the engine declares a rule to the effect that "every relvar must have at least one key", this could simply be enforced by declaring a catalog constraint such as
`ISEMPTY(RELVAR NOT MATCHING KEY).`

And if the engine declares a rule to the effect that "the total length of a physical record cannot exceed the length of the physical file pages the record is stored in", this could simply be enforced by declaring a catalog constraint such as
`ISEMPTY((SUMMARIZE RECORDTYPE <summaries to get the total record length here> JOIN FILE) WHERE RECORDLENGTH > PAGELENGTH).`

Only "vast minority" of cases of CDML validation cannot be performed this way, e.g. the case of a newly defined key, or a modified key, where current database data must satisfy the new (version of) the key.

Mixing catalog updates with updates of user databases

Imagine a D statement `VAR BASE RELATION {…} somename := RELATION { … }.`

From the perspective of catalog updating, this might give rise to a multi-statement containing both an update to the catalog relvars that define the logical structure of the user relvars, as well as an update(/assignment) to the newly defined user relvar itself. It will be clear that this too represents a case of "having to prioritise" the system's actions, however this case gives rise to a number of complications. First, there are assignments to the catalog to be done to define the relvar. Then, the new relvar itself needs to be assigned to, and then, because the initial value might be in violation of one of the keys declared for the new relvar, constraint checking is to be done on the new relvar itself too. Observe how this conflicts (or at least seems to conflict) with the desirability of being able to do <u>all</u> constraint checking <u>at the beginning</u> of the processing unit: how could we check a key constraint on a relvar that doesn't yet exist itself ?

The same observation can be made for a D multi-statement of the following kind:

```
VAR BASE RELATION {…} somename := RELATION { … },
CONSTRAINT … NOT(ISEMPTY(somename));
```

Once again, there is a need here to either do the constraint checking (the system action that is the consequence of the constraint declaration) <u>after</u> the "user part" of the update (the assignment of the initial value), or else a way is needed to verify constraints on relvars which don't even exist yet …

[3]This is actually a major understatement. Declarative support for database constraints of arbitrary complexity seems crucial. Without that feature, it would still represent a lot of work coding all those checks, and it's not entirely clear what they could possibly be "hooked" onto ...

It is unclear whether such stuff is supportable (and if so, how exactly), but currently such mixing of catalog and user-database updates is prohibited in SIRA_PRISE.

Observations regarding locking

It will be clear that measures need to be taken in order to avoid scenario's where one task is using a relvar, and another task is simultaneously changing some aspect of that relvar (be it its very logical structure, or some aspect of its physical implementation).

The most "drastic" approach to seeing that nothing nifty happens in the system, in this respect, is to simply apply 2PL to the relvar as a lockable object: every query that references the relvar, must take a 'read' (shared) lock on the relvar, and every catalog update operation that affects the relvar, must take a 'write' (exclusive) lock on the relvar. The catalog relvars can be exempt from this, as the catalog relvars themselves will obviously (???) never be changed while the engine is running.

Observations regarding commit/rollback of the CDML operation

TTM requires all constraints to be satisfied at end-of-statement. An engine supporting this prescription can thus safely be assumed to be such that the statements it executes are themselves "little transactions within the transaction" : the statement executes entirely or entirely not (A), the statement maintains data integrity (C), and the statement can be assumed to be at such isolation that data integrity can indeed be guaranteed (I).

CDML statements can thus safely be assumed to be, by and of themselves, full-fledged ACID transactions, minus the D property.

If a CDML statement is, on top of that, immediately committed (e.g. if it is issued in a transaction which performs autocommit), then we get the D property too. However, this is not necessarily the case. Are there consequences to the fact that a CDML statement need not necessarily be immediately committed, and can thus still be rolled back even after it has completed successfully ? Well, obviously yes there are. For example, CDML that manages the physical storage facilities for the data, will sometimes cause new OS files to be allocated, e.g. if an `INSERT FILE,RELATION{...}` is issued. If a rollback is issued after successful completion of such a statement, then strictly speaking the newly allocated OS files should be removed back again. If the catalog update constitutes a "segment" being inserted into an existing physical file, then this "segment" must be removed back again, and the existing physical file reset to the state it had before the catalog updates.

Since OS files are typically not "recoverable resources" (OS file systems typically are themselves not transactionally managed), this gives a number of complications we'd rather not have to resolve. In SIRA_PRISE therefore, catalog DML is always immediately committed, even if the transaction in which it is issued does not have its autocommit flag set[4].

[4]This commit would therefore also automatically commit any "non-catalog updates" that the transaction had "pending", were such a situation allowed.