Violating Independence

by David McGoveran

(Originally published in the Data Independent, Premier Issue, Jan. 1995: Updated Sept. 2014)

Introduction

A key aspect of the relational model is the separation of implementation details (sometimes referred to as the physical layer[1]) from logical and conceptual requirements. The assumption is that implementation issues are more likely to change than are the logical requirements. Even if this is not true in some circumstances, it certainly true that they are likely to change different times; that is, logical requirements and their physical realization or implementation are largely independent of each other and their very nature.

Application and database design often mixes these two ideas, to the detriment of performance, storage space, flexibility, extensibility, and maintenance. For example, procedural application code prescribes step-by-step operations, all too often implementing algorithms which depend on detailed knowledge of data structure, timing, memory management, and so on. This, of course, means that the application either does not perform well when significant changes are made to the available physical resources or cannot take advantage of those resources. The end result is that scalability is lost.

These problems are greatly aggravated in client/server applications. Two key benefits which client/server technology should be able to deliver are scalability and flexibility. Ordinarily, application code resides on a client platform which is separate from the DBMS and shared data on a server platform. This separation should permit IS to add physical resources to each platform independently, putting computing power where it is most needed, and fine tuning the operating system for a specific task. This specialization of resources is intended to provide greater efficiency. Unfortunately, the entanglement of application code with the details of data storage and access destroys that independence. The result is synchronization of client and server processing and higher network traffic as each platform is forced to communicate its state to the other. At the extreme, the application performs extract processing and the database server simply becomes a passive repository.

How DBMS Products Violate Independence

Vendors of so-called relational DBMSs are perhaps those most guilty of violating logical and physical independence. From the beginning of commercial relational products, the only physical storage structure vendors implemented was the table or flat file. This implementation was simply a bad approximation to the logical data structure imposed by the relational model, namely, the relation.

[1] Throughout this article I will use the terms relation, attribute, tuple, and domain to refer to logical elements of the relational model and the terms table, column, row, and storage type to refer to the corresponding physical elements of the relational model.

Although the physical page layout differed from vendor to vendor, very few DBA managed options for physical structures have been available. It is not possible, for example, to store the data as a linked list, a tree, or some other structure that might be efficient given the access pattern. Likewise, few products have ever offered more than the ubiquitous B-tree type of index, although it is known that B-tree indexing is not useful for all types of data and access.

Even worse than these limitations on data and index storage structures, few products have ever permitted rows from multiple tables to be physically co-located. The concept of providing indexes to multiple tables is all but unheard of. As a result, one often hears of the poor performance of joins: this cost is due almost entirely to disk I/0 imposed by forcing logically distinct but related data into separate physical locations and the lack of multi-table indices. This "problem with the relational model" is "solved" via the process colloquially known as "denormalization," resulting in even worse problems than it was intended to solve. I will return to this issue below.

Once a vendor has given up the distinction between logical and physical views, there is little hope for relational DBMS performance that can compete with a 3GL application. The great potential power of dynamic query optimization lies in there being a clean distinction between physical data storage the logical view of that data as a set of relations (not flat files, arrays, stored "tables" or anything else). Only when this separation is rigidly maintained can logical laws be invoked by the optimizer, transforming queries into simpler forms and mapping them with impunity to the most efficient set of physical operations on the data. The optimizers found in commercial relational DBMSs make extremely poor use of the power of logic on which the relational model is based. As a result, users end up writing procedural code to force the DBMS to execute a more efficient sequence of operations than it would when left to its own intelligence.

Some violations of independence are subtle. For example, a relation should never contain duplicates, a rule that all SQL DBMS products violate and one of the main reasons that SQL tables are not the same as relations. When SQL row uniqueness is enforced, the behavior on attempting to insert a duplicate row betrays a confusion between logical and physical. The proper logical behavior should mimic the union of a set with itself: the result is just the set (and so no duplicates). In particular, the logical behavior on insertion of a duplicate is for the duplicate to simply disappear! I must emphatically insist that this is not an "error" and requires no message or intervention by the user. Reporting "errors" of this sort convey information about the initial state of the table, but do so because of the physical way in which row insertion is interpreted by vendors.[2]

[2] If a user wants to know if a particular row insertion is redundant because the corresponding tuple is already in the relation, that can be achieved via a query. Of course, vendors could provide this side query as a convenient option on behalf of, and return the result to, the user. Similarly, options to automatically run and return the results of queries such as counting number of tuples in the set (before and/or after an operation) or the number of tuples affected during an operation could be implemented.

How SQL Violates Independence

Current implementations of SQL all contribute to the con- fusion between logical issues and physical issues. Before I point out some specific examples, let me make the distinction clear once and for all:

> *"Anything which pertains to the management of resource allocation and usage, response time, throughput, or concurrency is a physical issue."*

Given this, it is easy to see that statements such as SQL CREATE INDEX or ALTER TABLE P ADD TABLESPACE P_SPACE purely physical. SQL statements or explicit clauses in SQL statements are often intended to control transaction isolation via locking (a very physical notion): for example, statements such as LOCK TABLE (as found in ORACLE) or clauses such HOLDLOCK (as found in SYBASE). Consider the costly maintenance impact when the transaction mix is altered, making the hard coded control of transaction isolation invalid. Even worse, what if the vendor changes the lock mechanism, for example, from page locking to row locking? Perhaps a bit more subtle is the purely physical impact of the ORDER BY clause in SQL SELECT statements: row order is not a relevant part of the relational model. The ORDER BY clause asserts a relationship between rows which is physically manifested. All too often, applications are then written to take advantage of this order, as in, for example, the ubiquitous embedded SQL FETCH loop.[3] What happens to the application if the sort key values change so that the relationship is no longer legitimate?

It is unfortunate that SQL does not designate a distinct portion of the language as physical. Instead, it lumps physical statements together with the data definition language and the data manipulation language. Far more confusing, all dialects contain statements which are partially physical and partially logical. The worst is the SQL CREATE TABLE statement, which specifies (albeit very poorly) both the logical definition of a table, and its physical implementation! Every SQL dialect of which I am aware provides for a clause which permits the user to specify the physical storage characteristics of the table. All too often, these characteristics cannot be modified without redefining the table logically, sometimes having to drop or unload it altogether. What is needed is the ability to specify a relation definition and store it by name in the system catalog without any reference to physical storage. A completely separate statement should be used to initialize or modify storage allocation and structure.

[3] For example, a sequence number may be added to a table to facilitate record at a time updating. The application then retrieves a row with a sequence number greater than that of the last row updated and updates it. This provides a way of avoiding the locks that SQL SELECT … FOR UPDATE may imply, relying on application code to enforce consistency during concurrent updates. In this case, inserting new or deleting rows concurrently in the table invalidates the ordering assumed by the progress update. More subtle, but nonetheless potentially catastrophic, examples arise when COLLATION sequences are changed whenever application rely or ORDER BY for anything other than read-only reporting. To put it another way, ORDER BY should be understood as a transformation between relations and tables – with the result being something that is no longer a part of the relational DBMS.

Not only do the physical storage characteristics clause at the end of a CREATE TABLE belong to a completely different aspect of the relational model (i.e., the physical layer), but SQL confuses the concept of a physical data type with that of a domain when specifying columns. The data type specification for an attribute in a relation specification - which belongs exclusively to the logical layer of the relational model and should contain no physical aspects - should be the identification of the domain from which it is derived. In general, the logical specification of a relation should consist of a logical combination of a logical expression identifying the named attributes, the simple function of a particular domain that identifies each of them, and those integrity constraints which affect the relation or its components (domain, attribute, tuple, and multi-tuple constraints). In other words, the logical specification of a relation is its relation predicate.

Data types as specified by most SQL dialects refer to physically supported storage structures, so-called native data types and all too often these are selected from among those known to the CPU and/or operating system. Not only does this impede portability, but it represents the vendors' singular disregard for the very definition of a relation. Without support for relational domains (including domain constraints and domain operators), data independence is circumvented by the CREATE TABLE reference to physical data types. If you doubt that the data types found in the CREATE TABLE statement are physical, simply investigate the impact of column order and data type on physical storage in your favorite DBMS: you will find that they can be used to optimize physical storage and, sometimes, even I/O.

This confusion is carried forward to the ALTER TABLE statement, which is sometimes logical and sometimes physical. Likewise, SQL dialects often provide an explicit mechanism for the user to instruct the optimizer. This is particularly bad when the instructions are mixed with the data manipulation statement, as is the case with ORACLE "hints" and the SYBASE SQL Server FORCEPLAN. While I do believe that relational optimizer state of the art is so poor that such user written instructions are necessary today, I insist that they should be separate statements in the dialect. Perhaps worst of all is sensitivity of the optimizer to SQL syntax, including order of table references (FROM clause) and order of subquery evaluation. Invariably, applications take advantage of these product characteristics for optimizing performance. Of course, since they are hard-coded, dynamic optimization is lost. This problem becomes particularly apparent when storage structures are modified, when additional space is used by the table, or when the physical density of data changes (rows per page).

How Database Designers Violate Independence

When database designers "denormalize," they are confusing logical and physical. To some extent, this confusion is forgivable since vendors fail to provide the facilities to handle the problem properly. However, it also represents a complete misunderstanding of the very concept of normalization. Again, for the record, let me state the function of normalization once and for all:

> *"Normalization is a process whereby tables are iteratively redefined so that any combination of relational operations on those tables produce only expected results. That is, a fully normalized database consists of relations and only relations."*

Normalization is thus a logical issue and not a physical one at all. Its value should be understood as guaranteeing that operations on the database produce correct and predictable results. In this regard, it is not an optional process. There is no reason that the physical implementation of the database should be "normalized." Indeed, there are many reasons (performance, concurrency, etc.) it should not. What matters is that the algorithms which the DBMS uses to process relational operators always manipulate normalized relations – that is, that the results as perceived by the user are as if those operations were performed on normalized relations. Physical access methods should be able to access the data on disk or in memory and translate it into relations before passing it on to the relational operators. Likewise, physical access methods should be able to take a relation as output from a relational operator and translate it into a DBA designated or system optimized storage structure. This layering of processing within the DBMS will insure that physical issues are always transparent to the relational operators and ultimately to the user. The proper concept which denormalization attempts to implement is an optimized physical design. Physical design should attempt to minimize disk I/O given (a) the relational operations that must be performed, (b) the storage structures available, and (c) the physical access methods available. However, that physical design should always be hidden from relational operators[4] and users. If it is, then join processing can be made extremely efficient through physical clustering of frequently joined tables; this technique does not remove the join operation, it simply makes it one which can be processed with significantly fewer disk I/Os. Of course, as with all physical design options, there is a trade-off: a single table scan will almost certainly require a larger number of disk I/Os than it would if the rows of each table were in contiguous storage.

How Application Developers Violate Independence

Given these violations of independence by vendors and SQL, it is not surprising that application developers would be confused as to the distinction between physical and logical. As noted above, application code often takes advantage permanent physical structures such as record order or temporary row order resulting from a DISTINCT, GROUP BY, or ORDER BY clause. If the DBMS supports clustered indexes (indexes that force the order of rows on disk and, as a side

[4] If relational operators are permitted to operate on non-relations derived from physical storage (which may include duplicate records, nested records, repeating groups, etc.), the results are – in general – unpredictable, possibly inconsistent, and at least dependent on the details of the physical implementation. In consequence, differences in physical implementation become visible to users (and programs). Once again, independence is lost – in this case, it is physical data independence.

effect, row retrieval order), applications are often written to depend on this order and then fail when the index is changed.

Applications may also use the existence of a particular unique index to impose a uniqueness constraint, and then use violations of that uniqueness constraint to conditionally determine whether a row-level operation on the table should be an insert or an update. This is, of course, the basic bulk merge problem.

In general, application code which includes positioned up- dates, positioned deletes, or fetch loops usually have taken advantage of some physical information. Certainly they are not making set-oriented and declarative requests to the DBMS. Even if the optimizer were very powerful, this step-by-step instruction of the DBMS would be quite effective in disabling it.

Conclusion

The confusion of logical and physical is so pervasive in today's DBMS products, as well as among vendors, so-called relational experts, and database designers, that it should not be surprising when we find that application developers and users are confused. Nor should we be surprised that the advocates of non-relational approaches (for example, object databases) make negative claims about the potential of products based on the relational model. Such claims arise from experience with products that violate independence, while simultaneously offering little control over physical access and storage options. If ever there was an aspect of the relational model which we should insist vendors adhere to, it is independence.

Note: Although no longer published, The Data Independent notice is repeated here by way of recognition. You are encouraged to access DBDebunk at http://www.dbdebunk.com for related publications.

The Data Independent is published jointly by C. J. Date, David McGoveran, and Fabian Pascal, six times a year. All articles are published by permission of the copyright holders, the respective authors or their assignees. Letters and subscription requests may be addressed to:

The Data Independent
P.0.Box 471831
San Francisco, CA 94147-1831
Telephone: 415/776-5184 CompuServe: 75107,1333
Subscription rates are $95.00 per year.