The Singular Success of SQL

PAT HELLAND

**SQL HAS A
BRILLIANT FUTURE
AS A MAJOR
FIGURE IN THE
PANTHEON OF DATA
REPRESENTATIONS**

SQL has been singularly successful in its impact on the database industry. Nothing has come remotely close to its ubiquity. Its success comes from its high-level use of relational algebra allowing set-oriented operations on data shaped as rows, columns, cells, and tables.

SQL's impact can be seen in two broad areas. First, the programmer can accomplish a lot very easily with set-oriented operations. Second, the high-level expression of the programmer's intent has empowered huge performance gains.

This column discusses how these features are dependent on SQL creating a notion of stillness through transactions and a notion of a tight group of tables with schema fixed at the moment of the transaction. These characteristics are what make SQL different from the increasingly pervasive distributed systems.

SQL has a brilliant past and a brilliant future. That future is not as the singular and ubiquitous holder of data but rather as a major figure in the pantheon of data representations. What the heck happens when data is not kept in SQL?

SQL: THE MIRACLE OF THE AGE, OF THE AGES,
AND OF THE AGED

I launched my career in database implementation when

*It's Not Your Grandmother's
Database Anymore

Jimmy Carter was president. At the time, there were a couple of well-accepted representations for data storage: the *network* model was expressed in the CODASYL [Conference/Committee on Data Systems Languages] standard with data organized in sets having one set owner (parent) and multiple members (children); the *hierarchical* model ensured that all data was captured in a tree structure with records having a parent-child-grandchild relationship. Both of these models required the programmer to navigate from record to record.

Then along came these new-fangled relational things. INGRES (and its language QUEL) came from UC Berkeley. System-R (and its language SQL) came from IBM Research. Both leveraged relational algebra to support set-oriented abstractions allowing powerful access to data.

At first, they were really, really, really slow. I remember lively debates with database administrators who fervently believed they must be able to know the cylinder on disk holding their records! They most certainly did not want to change from their hierarchical and network databases. As time went on, SQL became inexorably faster and more powerful. Soon, SQL meant database and database meant SQL.

A funny thing happened by the early 2000s, though. People started putting data in places other than “the database.” The old-time database people (including yours truly) predicted their demise. Boy, were we wrong!

Of course, for most of us who had worked so hard to build transactional, relational, and strongly consistent systems, these new representations of data in HTML, XML, JSON, and other formats didn’t fit into our worldview. The

The data and metadata (schema) must remain still while SQL does its thing.

radicals of the '70s and '80s became the fuddy-duddies of the '00s. A new schism had emerged.

SQL, VALUES, AND RELATIONAL ALGEBRA

Relational databases have tables with rows and columns. Each column in a row provides a cell that is of a well-known type. DDL (Data Definition Language) specifies the tables, rows, and columns and can be dynamically changed at any time, transforming the shape of the data.

The fundamental principle in the relational model is that all interrelating is achieved by means of comparisons of values, whether these values identify objects in the real world or indicate properties of those objects. A pair of values may be meaningfully compared, however, if and only if these values are drawn from a common domain.

The stuff being compared in a query must have matching DDL or it doesn't make sense. SQL depends on its DDL being rigid for the duration of the query.

There's not really a notion of some portion of the SQL data having extensible metadata that arrives with the data. All of the metadata is defined before the query is issued. Extensible data is, by definition, not defined (at least at the receiver's system).

SQL's strength depends on a well-defined schema. Its set-oriented nature uses the well-defined schema for the duration of the operations. The data and metadata (schema) must remain still while SQL does its thing.

THE STILLNESS AND ISOLATION OF TRANSACTIONS

SQL is set oriented. Bad stuff happens when the set of data

slides around during the computation. SQL is supposed to produce consistent results. Those consistent results are dependent on input data that *appears to be unchanging*.

Transactions and, specifically, transactional isolation provide the sense that nothing else is happening in the world.

The Holy Grail of transaction isolation is *serializability*. The idea is to make transactions *appear* as if they happened in a serial order. They don't actually have to occur in a serial order; it just has to seem like they do.

In figure 1, the red transaction Ti depends upon changes made by the green transactions (Ta, Tb, Tc, Td, and Tf). The blue transactions (Tk, Tl, Tm, Tn, and To) depend on the changes made by Ti. Ti definitely is ordered after the green transactions and before the blue ones. It doesn't matter if any of the yellow transactions (Te, Tg, Tj, and Th) occur

FIGURE 1: TRANSACTION SERIALIZABILITY

before or after T_i . There are many correct serial orders. What matters is that the concurrency implemented in the system provides a view that is serializable.

Suddenly, the world is still and set orientation can smile on it.

A SENSE OF PLACE

SQL and its relational data are almost always kept inside a single system or a few systems close to each other. Each SQL database is almost always contained within a trust boundary and protected by surrounding application code.

I don't know of any systems that allow untrusted third parties to access their back-end databases. My bank's ATM, for example, has never let me directly access its back-end database with JDBC (Java Database Connectivity). So far, the bank has constrained me to a handful of operations such as deposit, withdrawal, or transfer. It's really annoying! In fact, I can't think of any enterprise databases that allow untrusted third parties to "party" on their databases. All of them insist on using application code to mitigate the foreigners' access to the system.

Interactions occur across these systems, but they are implemented with some messages or other data exchange that is loosely coupled to the underlying databases on each side. The messages hit the application code and not the database.

Each of these databases appears to be an island unto itself. Now, that island may have a ferry or even a four-lane bridge connecting it to other islands. Still, you always know the island upon which you stand when you access a database.

Distributed transactions across different SQL databases are rare and challenging.

DIFFERENT PLACES MEANS DIFFERENT TIMES

Multiple databases sharing a transactional scope is extremely rare. When a transaction executes on a database, there is no clear and crisp notion of its time when compared with the time on another system's database. Distributed transactions across different SQL databases are rare and challenging.

If you assume two databases do not share a transactional scope, then the simple act of spreading work across space (the databases) implies spreading the work across time (multiple transactions). This transition from one database to more than one database is a seminal semantic shift. Space and time are intrinsically tied to each other.

When you pop from one to many, SQL and its relational algebra cannot function without some form of restriction. The most common form is to cast some of the data into immutable views that can be meaningfully queried over time. The system projecting these views won't change them. When they don't change, you can use them across space.

Freezing data in time allows its use across spatial boundaries. Typically, you also project the data to strip out the private stuff as you project across trust boundaries. To span spatial boundaries, time must freeze, at least for the data being shared. When you freeze data, it's immutable.

IMMUTABILITY: THE ONE CONSTANT OF DISTRIBUTED SYSTEMS

Immutable data can be immortal and omnipresent. Think of it as the Gideon Bible, which seems to be in every hotel room; I suspect there will be Bibles there for a long time. If you want to do a query leveraging the Gideon Bible as an

input, you will not struggle with challenges of concurrency or isolation. It's relatively straightforward to cache a copy close to where you need it, too.

SQL's relational operations can be applied to immutable data at a massive scale because the metadata is immutable and the data is immutable. This empowers MapReduce, Hadoop, and the other big-data computation. By being immutable, the contents are still and the set-oriented computations make sense.

Immutable data can be everywhere at any time. That allows it to be both inside the singularity and outside of it. No big deal. Immutability truly is one of the unifying forces of distributed systems.

Classic centralized databases force their data to appear immutable using transactions. When distribution impedes the use of transactions, you snapshot a subset of your data so it can be cast across the boundaries with predictable behavior.

ESCAPING THE SINGULARITY

SQL databases are phenomenally powerful and have enjoyed singular success in providing access to and control over data. They allow the combination and analysis of data by leveraging relational algebra. Relational algebra relates values contained in the rows and the columns of its tables. This has provided incredible power in programming and huge performance gains in accessing relational data.

To do this, relational algebra requires a static set of tables unmolested by concurrent changes. Both the data and the schema for the data must be static while operations are performed. This is achieved with

transactional serializability or other slightly weaker isolation policies. Serializability provides the illusion that each user of the database is alone at a *single point in time*.

In a relational database, it is hard to provide full functionality when distributed except, perhaps, across a handful of machines in close proximity. Even more profoundly, SQL works well within a single trust boundary such as a department or a company. SQL databases provide the illusion that they exist at a *single point in space*.

Providing a single point in space and time yields both stillness and isolated location. This empowers the value-based comparisons of relational algebra. It looks just like a *singularity*.

The industry has leapt headlong toward data representations that are neither bound to a single point in time nor to a single point in space with distributed, heterogeneous, and loosely coupled systems. Nowadays, far more data is being generated outside the SQL environment than within it. This trend is accelerating.

This column explores various consequences of *escaping the singularity* and relaxing the constraints of both space and time. No, it ain't your grandmother's database anymore.

Pat Helland has been implementing transaction systems, databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems since 1978. For recreation, he occasionally writes technical papers. He currently works at Salesforce.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.