IDERA® | Database Tools Whitepaper

SQL SYNTAX AND QUERY PERFORMANCE

BY JOE CELKO

TABLE OF CONTENTS

SQL SYNTAX AND QUERY PERFORMANCE

The book ‘Me Talk Pretty One Day’ (ISBN 978-0316777728) is a collection of essays by American humorist David Sedaris. He moved to a rural part of Normandy, France, and had no idea how to speak or read French at the time. The humor comes from his frustrated attempts to learn the language and culture. The locals knew him as the American idiot who scares their animals by screaming. The warning on electrified fences was in French and a great source of amusement for the neighbors.

New SQL developers on every platform also do not ‘talk pretty’ and keep struggling while trying to get out of the mindset of their prior language. The SQL language made a massive jump in the SQL-92 Standards. With those standards, it added many new language features based on the actual usage of the language, improvements in the hardware, and attempts to make ‘talk SQL pretty’ easier and more natural.

The query that runs well on a local set of test data can fail miserably on the production system. Very often, a simple change in the syntax can make a huge performance difference. Much of SQL advice is ‘it depends’ because tuning a database is both art and science.

However, there are some principles you can follow that should produce good SQL most of the time and give you a maintainable schema. These are heuristics, not mathematical formulas.

WRITE SQL THAT LOOKS LIKE SQL

FORTRAN, BASIC, COBOL, and most other languages have a limited set of logical operators based on simple Boolean (that is, two-valued) algebra. The problem is not just the SQL has a three-valued logic. However, it has shorthands that deal with 'sets of predicates' instead of a linear sequence of infixed Boolean expressions.

BETWEEN PREDICATES

Consider this simple generic logical expression, which occurs in real code all the time.

```
((a <= x) and (x >= b))
```

That expression works just fine in SQL, too. But an experienced SQL programmer would use:

```
x BETWEEN a AND b
```

See the difference in the mindsets? A pair of two very simple predicates reduces to a trinary operator. Between-ness started as a geometric property for points on a line. The optimizations that use it are based on those theorems.

Now that the higher-level concept is locked in a bit of syntax, we can generalize it with additional syntax. The BNF for the BETWEEN is as follows, and the asymmetric option is the default equivalent to the code I just gave.

```
<between predicate> ::=
 <row value predicand> <between predicate part 2>
<between predicate part 2> ::=
 [NOT] BETWEEN [ASYMMETRIC, SYMMETRIC]
 <row value predicand> AND <row value predicand>
X BETWEEN SYMMETRIC y AND z
```

means

```
((x BETWEEN ASYMMETRIC y AND z)
 OR (x BETWEEN ASYMMETRIC z AND y))
```

See how we are getting a lot of programing power with minor syntax extensions as we expand the language? Try another example of this principle:

MEMBERSHIP PREDICATES

Consider this simple generic logical expression, which you see in a lot of programs:

```
((foo = 1) or (foo = 2) or (foo = 3))
```

That expression works just fine in SQL, too. But an experienced SQL programmer would use:

```
foo IN (1,2,3)
```

We stole this syntax for the simple IN() from Pascal. It is an n-ary (that is, with n > 0) logical operator. Is there any advantage to using this over a chain of OR-ed expressions?

The first distinct advantage is the amount of typing you can save.

The next benefit is that the right-hand list is much easier to read. You see that foo compares to an element in a set of values. That short expression is the conceptual difference between a bunch of trees and a forest, employees versus the abstraction of Personnel. Cantor called this 'the many that is treated as a one' when he defined set theory, and so forth.

The final advantage is that some SQL product optimizes the search of the list. The values can be put into a binary tree or a hash table to speed up searching when the list is long enough. That optimization leads to the counterintuitive situation where adding extra elements to the list makes the query run faster by making the list long enough for the optimization to kick in. These additional values can be absurd so that the code looks strange.

That syntax is easy to generalize to any expression that returns a list or a one-column table like 'x IN (SELECT y FROM ..)'. However, there exists another syntax for this that most programmers do not know: It generalizes the theta operator rather than the source of the list.

GENERALIZED GROUP PREDICATES

A theta operator is the fancy name for a comparison operator and can be any of '=, <, >, <>, <=, >=' and not just the equality of the IN() predicate. The ANSI/ISO Standard defines the IN() predicate as a shorthand for the '= ANY' predicate and '<> ANY' as a shorthand for NOT IN().

There are two shorthand notations that compare a single value to those of a subquery, and take the general form <value expression> <theta op> <quantifier> <subquery>. The first predicate '<value expression> <theta op> [ANY, SOME] <table expression>' is equivalent to taking each row, s, (that is, assume that they are numbered from 1 to n) of <table expression> and testing '<value expression> <theta op> s' with ORs between the expanded expressions:

```
((<value expression> <theta op> s1)
OR (<value expression> <theta op> s2)
 ...
OR (<value expression> <theta op> sn))
```

When you get a single TRUE result, the whole predicate is TRUE. As long as <table expression> has cardinality greater than zero and a single non-NULL value, you get a result of TRUE or FALSE. The keyword SOME is the same as ANY, and the choice is just a matter of style and readability.

THE <THETA> ALL <SUBQUERY> SYNTAX

Likewise, '<value expression> <comp op> ALL <table expression>' takes each row, s, of <table expression> and tests <value expression> <comp op> s with ANDs between the expanded expressions. We generalized the logic what are called quantifiers.

```
((<value expression> <comp op> s1)
AND (<value expression> <comp op> s2)
 ...
AND (<value expression> <comp op> sn))
```

When you get a single FALSE result, the whole predicate is FALSE. As long as <table expression> has cardinality greater than zero and all non-NULL values, you get a result of TRUE or FALSE.

That sounds reasonable so far. Let Empty_Set be an empty table (that is, no rows, cardinality zero). Also, let NullTable be a table with only NULLs in its rows and a cardinality greater than zero. The rules for SQL say that '<value expression> <comp op> ALL NullTable' always returns UNKNOWN, and likewise '<value expression> <comp op> ANY NullTable' always returns UNKNOWN. Those rules make sense because every row comparison test in the expansion would return UNKNOWN. Consequently, the series of OR and AND operators would behave in the usual way.

However, '<value expression> <comp op> ALL Empty_Set' always returns TRUE and '<value expression> <comp op> ANY Empty_Set' always returns FALSE. Most people have no trouble seeing why the ANY predicate works that way. You cannot find a match so that the result is FALSE. But most people have lots of trouble seeing why the ALL predicate is TRUE. This convention is called existential import, and it was a big debate at the start of modern logic. It boiled down to deciding if the statement 'All x are y' implies that 'some x exists' by definition. The modern convention is that it does not. Lewis Carroll went with existential import, John Venn and George Boole did not. And neither did modern logic. Let me explain.

If I were to walk into a bar and announce that I can beat any pink elephant in the bar, that would be a true statement. The fact that there are no pink elephants in the bar merely shows that the problem reduces to the minimum case. If this seems unnatural, then convert the ALL and ANY predicates into EXISTS predicates and look at the way that this rule preserves the formal mathematical properties that:

```
1)  (∀x)P(x)  = ¬(∃x) ¬P(x)
2)  (∃x)P(x)  = ¬(∀x)¬P(x)
```

If you ave not had a mathematical logic course or just forgot everything, the (∀x) means 'for all x ..' and (∃x) means 'there exists one or x, such that ..' and an optimizer can use these rules (that is, with allowances for NULL and three-valued logic).

THE ALL PREDICATE AND EXTREMA FUNCTIONS

It is counter-intuitive at first that these two predicates are not the same in SQL:

```
x >= (SELECT MAX(y) FROM Table1)
x >= ALL (SELECT y FROM Table1)
```

But you have to remember the rules for the extrema functions – they drop out all the NULLs before returning the greatest or least values. The ALL predicate does not drop NULLs so that you can get them in the results.

However, if you know that there are no NULLs in a column or are willing to drop the NULLS yourself, then you can use the ALL predicate to construct single queries to do work that would otherwise be done by two queries. For example, given the table of products and store managers, to find which manager handles the largest number of products, you first construct a grouped VIEW or CTE and group it again. Assume we have a table of the managers responsible for each product we stock:

Now, the query:

```
WITH TotalProducts (manager_name, product_tally)
AS
(SELECT manager_name, COUNT(*)
  FROM Product_Managers
 GROUP BY manager_name)

SELECT manager_name
 FROM TotalProducts
 WHERE product_tally
    = (SELECT MAX(product_tally)
       FROM TotalProducts);
```

But Alex Dorfman found a single query solution instead:

```
SELECT manager_name, COUNT(*) AS product_tally
 FROM Product_Managers
 GROUP BY manager_name
HAVING COUNT(*) + 1
    > ALL (SELECT DISTINCT COUNT(*)
        FROM Product_Managers
        GROUP BY manager_name);
```

The use of the SELECT DISTINCT in the subquery is to guarantee that we do not get duplicate rows when two managers handle the same number of products. You can also add a '.. WHERE dept IS NOT NULL' clause to the subquery to get the effect of a true MAX() aggregate function.

EXISTS()

The EXISTS predicate is very natural. It is a test for a non-empty table. If there are any rows in its subquery, then it is TRUE. Otherwise, it is FALSE. This predicate does not give an UNKNOWN result. The syntax is EXISTS <table subquery>. It is worth mentioning that a <table subquery> is always inside parentheses to avoid problems in the grammar during parsing.

Some early SQL implementations would work better with 'EXISTS(SELECT <column name> ..)', ' EXISTS(SELECT <constant> ..)' or 'EXISTS(SELECT * ..)' versions of the predicate. Today, there is no difference between the three forms in the major products. Consequently, the 'EXISTS(SELECT * ..)' is now the preferred form since it shows that we are working at the table and row level, without regard to columns.

Indexes are useful for EXISTS() predicates because you can search them while the base table is left entirely alone. For example, we want to find all the employees who were born on the same day as any famous person. The query could be

```
SELECT P.emp_name AS famous_person_birth_date_guy
 FROM Personnel AS P
WHERE EXISTS
   (SELECT *
     FROM Celebrities AS C
    WHERE P.birth_date = C.birth_date);
```

If the table Celebrities has an index on its birth_date column, the optimizer get the birth_date, P.birth_date, of the current employee and look up that value in the index. If the value is in the index, then the predicate is TRUE. Consequently, we do not need to look at the Celebrities table at all. If it is not in the index, then the predicate is FALSE. Consequently, there is still no need to look at the Celebrities table. That process should be fast since indexes are smaller than their tables and are structured for quick searching.

However, if Celebrities has no index on its birth_date column, then the query may have to look at every row to see if there is a birth_date that matches the birth_date of the current employee. There are some tricks that a good optimizer can use to speed things up in this situation.

A NULL might not be a value. However, it does exist in SQL. That NULL is often a problem for a new SQL programmer who is having trouble with NULLs and how they behave.

Think of them as being like a brown paper bag: You know that something is inside because you lifted it and felt a weight. However, you do not know exactly what that something is. If you felt an empty bag, then you know to stop looking. For example, we want to find all the Personnel who were not born on the same day as a famous person. Answer that inquiry with the negation of the original query, like this:

```
SELECT P.emp_name AS famous_birth_date_person
 FROM Personnel AS P
 WHERE NOT EXISTS
    (SELECT *
     FROM Celebrities AS C
     WHERE P.birth_date = C.birth_date);
```

But assume that among the Celebrities, we have a movie star who does not admit her age, shown in the row ('Gloria Glamour', NULL). A new SQL programmer might expect that Ms. Glamour would not match to anyone since we do not know her birth_date yet. She matches to everyone since there is a chance that they may match when some tabloid newspaper finally gets a copy of her birth certificate. But work out the subquery in the usual way to convince yourself:

You find another problem with NULLs when you attempt to convert IN() predicates to EXISTS predicates. Using our example of matching our Personnel to famous people, rewrite the query as:

```
SELECT P.emp_name AS famous_birth_date_person
 FROM Personnel AS P
 WHERE P.birth_date
    NOT IN
   (SELECT C.birth_date
     FROM Celebrities AS C);
```

However, consider a more sophisticated version of the same query, where the celebrity has to have been born in New York City. The IN() predicate would be

```
SELECT P.emp_name -- born on a day without a famous New Yorker!
 FROM Personnel AS P
 WHERE P.birth_date
 NOT IN
  (SELECT C.birth_date
    FROM Celebrities AS C
 WHERE C.birth_city_name = 'New York');
```

and you would think that the EXISTS version would be:

```
SELECT P.emp_name, -- born on a day without a famous New Yorker!
 FROM Personnel AS P
 WHERE NOT EXISTS
    (SELECT *
     FROM Celebrities AS C
     WHERE C.birth_city_name = 'New York'
 AND C.birth_date = P.birth_date);
```

Assume that Gloria Glamour is our only New Yorker and we still do not know her birth_date. The subquery is empty for every employee in the NOT EXISTS predicate version, because her NULL birth_date does not test equal to the known employee birthdays.

That means that the NOT EXISTS predicate returns TRUE, and we get every employee to match Ms. Glamour. But now look at the IN() predicate version, which has a single NULL in the subquery result. This predicate is equivalent to (Personnel.birth_date = NULL), which is always UNKNOWN, and we get no Personnel back.

Likewise, you cannot, in general, transform the quantified comparison predicates into EXISTS predicates, because of the possibility of NULL values. Remember that 'x <> ALL <subquery> ' is shorthand for 'x NOT IN() <subquery>' and 'x = ANY <subquery> ' is shorthand for 'x IN <subquery> ' and it will not surprise you.

In general, the EXISTS predicates runs faster than the IN() predicates. The problem is in deciding whether to build the query or the subquery first. The optimal approach depends on the size and distribution of values in each, and that is not usually known until run-time.

THE OVERLAPS() PREDICATE

The OVERLAPS() predicate for finding out if two temporal periods overlap each. Let us call the periods (S1, T1) and (S2, T2). The S is for the starting timestamp of an event, T is for the termination timestamp of the event. The result of the following expression formally defines the result of the <OVERLAPS predicate>:

```
(S1 > S2 AND NOT (S1 >= T2 AND T1 >= T2))
OR (S2 > S1 AND NOT (S2 >= T1 AND T2 >= T1))
OR (S1 = S2 AND (T1 <> T2 OR T1 = T2))
```

The rules for the OVERLAPS() predicate sound like they should be intuitive. However, they are not. The principles that we wanted in the Standard were:

1. A time period includes its starting point but does not include its endpoint. We have already discussed this model and its closure properties.

2. If the periods are not 'instantaneous', then they overlap when they share a common period.

3. If the first term of the predicate is an INTERVAL and the second term is an instantaneous event (that is, a <datetime> data type), then they overlap when the second term is in the period (but is not the endpoint of the period). That follows the half-open model.

4. If the first and second terms are both instantaneous events, then they only overlap when they are equal.

5. If the starting time is NULL and the finishing time is a <datetime> value, then the finishing time becomes the starting time, and we have an event. If the starting time is NULL and the finishing time is an INTERVAL value, then both the finishing and starting times are NULL.

Please consider how your intuition reacts to these results when the granularity is at the YEAR-MONTH-DAY level. Remember that a day begins at 00:00:00 Hrs.

```
(today, today) OVERLAPS (today, today) = TRUE
(today, tomorrow) OVERLAPS (today, today) = TRUE
(today, tomorrow) OVERLAPS (tomorrow, tomorrow) = FALSE
(yesterday, today) OVERLAPS (today, tomorrow) = FALSE
```

The full definition is more complicated. However, the point is that SQL can hide the complexity with a single syntactic construct.

THE CASE EXPRESSION

The CASE expression is probably the most useful addition made in the SQL-92 Standard. ANSI stole the idea and the syntax from the ADA programming language and the selection operators from recursive function theory. Here is the BNF for a <case specification>:

```
<case specification> ::= <simple case>, <searched case>
<simple case> ::=
CASE <case operand>
<simple when clause>...
[<else clause>]
END

<searched case> ::=
CASE
<searched when clause>...
[<else clause>]
END

<simple when clause> ::= WHEN <when operand> THEN <result>
<searched when clause> ::= WHEN <search condition> THEN <result>
<else clause> ::= ELSE <result>
<case operand> ::= <value expression>
<when operand> ::= <value expression>
<result> ::= <result expression>, NULL
<result expression> ::= <value expression>
```

It can be used to move old procedural code into declarative code that can be optimized. If you can use the <simple case>, then do it. It optimizes easily. While it is logically equivalent to a searched case with only equality tests on the <case operand>, nobody implements it that way. It is much faster to use hashing and other techniques for more extended CASE expressions.

The next trick is to remember that the WHEN clauses are tested in left-to-right order. If you arrange them so that the most likely ones appear first, then you do not waste time with the rare events. In medicine, this principle is expressed as 'Look for a horse, not a zebra' when doing diagnostics.

Most optimizers also factor out redundant tests by considering the order of execution. That is easier to see with an example:

```
CASE WHEN x = 1 THEN 'alpha'
   WHEN x = 1 AND y = 5 THEN 'beta'
   ELSE 'gama' END
```

If (x = 1) then the first WHEN clauses execute. There is no need to re-test that condition in the following WHEN clauses. If it worked, then we never got any further than the first THEN clause.

However, most optimizers are weak in nested CASE expressions. That weakness is funny because we spent a lot of time with nested if-then-else constructs in procedural languages. It is probably better to flatten out nested CASE expressions to a single level.

DO NOT RETRIEVE MORE COLUMNS THAN YOU NEED

Blindly using 'SELECT *' in a query is a waste of resources since you probably do not need everything in the result set. You are also locking resources from other users. Not blindly using 'SELECT *' is especially crucial in columnar SQL databases.

But you can get extraneous columns in other ways. When you do a 'cut & paste' to get some code reuse you can get more than you need. Remember to look at the actual text. When you invoke a VIEW with more columns than needed, you are creating the same problem. If the VIEWs are nested very deeply, then you probably cannot see the extraneous columns. You should un-nest the poorly performing query and see everything. Look for a few over-sized columns that you do not use (that is, did you need a full product description or just the UPC?).

Symptoms of this problem is a WHERE clause with additional conditions and a FROM clause with additional outer joins.

DELETE AND UPDATE IN BATCHES

Deleting or updating large amounts of data from a large table scan can eat resources. The problem is that both of these statements run as a single transaction. And if you need to kill them or if something happens to the system while they are working, then the system has to roll back the entire transaction. That rollback consumes lots of time and blocks other transactions.

The solution is to do deletes or updates in smaller batches. If the transaction gets killed, then the ROLLBACK is relatively small. And while each small batch is committing to disk, other transactions can 'sneak in' and do some work. Consequently, concurrency is greatly enhanced. If you are archiving, then you might want to take a little longer than production jobs.

Check your SQL as you may find that you have CHECKPOINT or SAVEPOINT (that is, the name varies) features. These are like 'mini-commits' which let a localized ROLLBACK restore the database to the SAVEPOINT, or COMMIT the work done so far. Such mini-commits might be a variable option in your situation

USE STORED PROCEDURES

Stored procedures have several advantages. The parameters are parsed and verified easily by the SQL engine. Push less data across the network. The stored procedure call always is much shorter. Also, stored procedures are easier to trace in profile tools. A stored procedure is an actual object in your database. That means that it is much easier to get performance statistics on a stored procedure than on an ad hoc query and, in turn, find performance issues and draw out anomalies.

Also, stored procedures parameterize more consistently. That parametrization means that you are more likely to reuse your execution plans and even deal with caching issues, which can be challenging to pin down with ad hoc queries. Stored procedures also make it much easier to deal with edge cases and even add auditing or change-locking behavior. A stored procedure can handle many tasks that trouble ad hoc queries.

Stored procedures are also not subject to SQL injections attacks. Take a look at this cartoon at 'https://xkcd.com/327/' to get a feeling as to how SQL injection works. The attacker passes a string of code as an argument to a block of dynamic SQL. This string contains malicious SQL statements that execute when preparing the dynamic SQL.

Finally, when you find a better way to do something, you replace the procedure body. If you can do the task with a subset of the old parameter list, then life is good. If you need to add a new parameter, then you have to do some work.

GUIDS ARE NOT WHAT YOU THINK

A UUID (universal unique identifier) or a GUID (globally unique identifier) is a 16-byte randomly generated number. It is not an attribute of anything in the database, so by definition, it cannot be a key. A key is a subset of the attribute of the entity. When you use this, you are faking a physical pointer and not doing RDBMS.

Ordering the data of your table on this column causes your table to fragment much faster than using a steadily increasing value like a DATE or SEQUENCE.

The 'G' in GUID means 'Global'. Never use GUIs for local data. It was created to locate external objects. Instead, look for a real identifier that has validations and verification rules.

DO NOT USE COUNT() IF YOU NEED EXISTENCE

Using count() when you need existence is a common error. Using a simple count to test for existence is insane.

```
IF (SELECT COUNT(*) FROM Foobars) > 0
THEN
<database actions>
END IF;
```

It is completely unnecessary. If you want to check for existence, then do this:

```
IF EXISTS (SELECT * FROM Foobars)
THEN
<database actions>
END IF;
```

Do not count everything in the table. Just get back the first row you find. SQL is smart enough to use EXISTS() correctly, and the second block of code returns much faster. The larger the table, the more significant difference this makes.

DO NOT DO NEGATIVE SEARCHES

Take the simple query:

```
SELECT *
 FROM Customers
WHERE region_nbr <> 3;
```

You cannot use an index with this query because it is a negative search that has to be compared row by row with a table scan. If you need to do something like this, then you may find it performs much better if you rewrite the query to use the index.

USE CTES

In linguistics and mathematics, the ability to replace a simple element with a complex element of the same type is called orthogonality. You can replace a noun in a sentence with a noun phrase, in a formula. You can also replace an integer by an integer function. And so forth.

If you had a good grammar class in middle school, you might have seen this sentence 'The rat the cat the dog chased killed ate the malt'. It is perfectly correct. It is a sentence with multiple center embeddings. We usually have no problem putting one clause inside another in English. We can take 'the rat ate the malt' and stick in more information to make 'the rat the cat killed ate the malt.' But the more clauses we add, the harder it gets to understand the sentence. In this case, the rat ate the malt. After that, a cat killed it. A dog chased that cat. The grammar of the sentence is fine. The style and readability are awful.

With a CTE, you have moved a derived table expression from the interior of SELECT statement to the front of it and put a name on it. The CTEs are created in left to right order, which means that a CTE can only reference previous declared CTEs.

However, once a CTE exists, it can be referenced by name in multiple places in the main statement. In theory, a good optimizer would find several occurrences of the same code, factor it out and decide if it should treat the code like a macro (that is, copied into the program text, then compiled everywhere it is used). Alternatively, a good optimizer should decide whether the code

should be materialized as a working table a single time. The theory is nice. However, deep nesting can fool an optimizer into taking the safe route and using the macro approach

The worst such construct I have seen is 16 pages of CTEs leading to a single SELECT * statement on the last page. Think about what nesting them as derived tables would look like, however! That would have been much worse. If the CTE gets a good name, then you can quickly see if it should be made into a VIEW and shared with other queries.

‘DOUBLE-DIPPING’ AND THE WINDOWS CLAUSE.

Double-dipping means avoid visiting the same table more than once. A horrible way to do this is with temporary tables built from the same base table, then joined together. If the optimizer of your SQL engine can decide to materialize VIEWs and share the results as a working table, then several sessions can benefit. If you have a less sophisticated optimizer, you can use temp tables and do this work manually.

The Window clause is the best one of the best ways to avoid this, and the performance improvements can be significant. The subclauses are similar to features of a subquery. The PARTITION BY is like a local GROUP BY that splits up the table into groupings. The ORDER BY imposes a sorted order. Finally, the window frame (ROW or RANGE) is like a local WHERE clause.

That construct is complex, and you need to spend time with it. To get an idea of the power, let us do a typical, simple running total. Many report writers have it built-in. However, in SQL, before the windows clause, we had to use self-joins for this. Assume we have a table of Sales

```
CREATE TABLE Sales
(sales_invoice_nbr CHAR(5) NOT NULL PRIMARY KEY,
 emp_id CHAR(10) NOT NULL
 REFERENCES Personnel(emp_id),
 sales_date DATE NOT NULL,
 sales_amt DECIMAL(10,2) NOT NULL);
```

To do a simple running total, ordered by the date of the sales:

```
SELECT S1.sales_date, SUM(S2.sales_amt) AS sales_amt_runtot
 FROM Sales AS S1, Sales AS S2
 WHERE S2.sales_date <= S1.sales_date
 GROUP BY S1.sales_date
 ORDER BY S1.sales_date; -- not really needed
```

The table appears as the row with the current sales, S1, and a second copy, S2, of the table appears as all prior sales, we a self-join, apply the WHERE clause and then group the rows in the result set. Each row in S1 invokes a query with a join to S2. Over and over and over.

But with a windows clause, we can write;

```
SELECT sales_date, emp_id,
    SUM(sales_amt)
      OVER (ORDER BY sales_date
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS sales_amt_runtot
 FROM Sales;
```

That code uses only a single copy of the Sales table. The additional keywords tell the optimizer that it can build an ordered list, total it, and then for each row, add the current sales_amt. Mere self-joins cannot tell the optimizer about the intent of the query. If you want to get the running totals by salesman, then just run a partition clause:

```
SELECT sales_date, emp_id,
    SUM(sales_amt)
    OVER (PARTITION BY emp_id
         ORDER BY sales_date
       ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS sales_amt_runtot
 FROM Sales;
```

The full syntax of the window clause is elaborate and not fully implemented in many SQLs. The basic types of windowed functions are and aggregate functions, rank functions, and position functions.

If the window starts at UNBOUNDED PRECEDING, then the lower bound is always the first row of the partition, or you can give an actual count of rows to use for the frame. CURRENT ROW explains itself. The <window frame preceding> is an actual count of preceding rows.

UNBOUNDED FOLLOWING or an actual count of following rows are the same ideas on the other side of the current row. There are also <window frame exclusion> clauses that remove the current row, a group, and tires.

The aggregate functions are the usual ones from earlier SQL standards – SUM(). MIN(), MAX(), AVG() and COUNT(). We already saw an example of a summation and the other aggregates work the same way. The ORDER BY clauses does not affect MIN(), MAX() and COUNT()

The rank functions are ROW_NUMBER(), RANK(), DENSE_RANK(), PERCENT_RANK(), and CUME_DIST(). They assign an ordering number within the scope of the window. ROW_NUMBER() is the most important and the most obvious. It returns a number from 1 to (n) for all of that rows in the window. Again there are a lot of programming idioms with multiple calls to this function.

The position functions are not yet part of the standards at this time. However, they are common. There is LEAD(<column>[<n>]) and LAG(<column>[<n>]) which return the value in a column that is (n) rows preceding or following the current row. There are also FIRST and LAST functions that work with the rows at the edges of the window.

It is worth mentioning that the rules for an ORDER BY subclause have changed to be more general than they were in earlier SQL Standards. A sort can now be a <value expression> and is not limited to a simple column in the select list. However, it is still a good idea to use only column names so that you can see the sorting order in the result set. You put NULL either first or last as well as ASC and DESC options.

CONCLUSIONS

Just as it took Mr. Sedaris some time to learn 'talk pretty one day' when he went to live in France, it takes some time for new SQL developers to 'talk pretty' and get out of the mindset of their prior language. If I had to summarize all of the tips in this article, then it would be to write SQL that looks like SQL. That is similar to you trying to speak French that sounds like real French instead of the 'Fran-lish' that comes from reading a phrasebook with an American accent.

Learn each SQL statement and all of the tricks inside it, even if you have no immediate need.

Practice coding SQL. Do not expect to have an epiphany overnight. However, one day, you will talk pretty in SQL.

ABOUT THE AUTHOR

Mr. Joe Celko serves as Member of Technical Advisory Board of Cogito, Inc. Mr. Celko joined the ANSI X3H2 Database Standards Committee in 1987 and helped write the ANSI/ISO SQL-89 and SQL-92 standards. He is one of the top SQL experts in the world, writing over 700 articles primarily on SQL and database topics in the computer trade and academic press. The author of six books on databases and SQL, Mr. Celko also contributes his time as a speaker and instructor at universities, trade conferences, and local user groups.