Whitepaper

embarcadero

Cardinality is not For the Birds

By Joe Celko

For Embarcadero Technologies

August, 2015

Table of Contents

Cardinality is not for the Birds

Cardinality has nothing to do with song birds or Catholic church officials. It is a term from set theory that deals with the size of a set. Originally, Georg Cantor invented the term, so that he could talk about infinite sets like integers, real numbers, irrationals, power sets and so forth.

Yes, some infinite sets are "bigger" than others, but you do not want to say that because (1) it just sounds strange to non-mathematicians (2) Mathematicians might work with infinities, but database guys do not. Big data is not that big.

History Lessons

Back in the dark ages, data was kept on punch cards and magnetic tape files. The physical data and logical data were welded into one unit of work. In fact, a punch card was called a "unit record" in IBM-speak back then. It is also how programs were written; each record was processed one at a time, with summary computations collected at each step. The goal was to mount a single tape and create an output for a single task. No parallelism – how could multiple users share a tape drive, anyway? How can a third party update a tape file while it is in use?

The first magnetic tape drives could only read forward, just like a punch card reader. In fact, the tape was often made up of card images. The advantages were obvious (1) a tape can hold a lot card images (2) later tape drives could buffer data and read backwards, which was impossible with card readers (3) Unlike a deck of cards, when you drop a magnetic tape it does not fly all over the computer room floor. Until you have to reassemble a deck of cards, you have no idea what a blessing #3 is.

The typical work pattern was to merge new data from transaction tapes or cards into a Master tape, save the old master tape for several generations and then use the new Master for pre-defined reports like invoices, summaries, and basic accounting functions. It was possible to flag a record as "inactive" by setting a bit at the front of the record so the system could skip over it and then clean them out in the next generation tape.

The cardinality of a sequential magnetic tape file is basically a count of the active physical records. If I doubled the number of records I processed, I pretty much doubled the time total time of the job. The bad news was having to mount (or "hang" to use the slang) more than one tape; this was a physical event that requires time from a person.

Random access to single records on a tape drive is not productive. It is vital that the data is sorted on the search key for sequential files. If you want to find a value in a non-sorted field, you expect to have to scan the whole tape, record by record. This meant an expected time of (n/2) reads, where (n) is the number of physical records on the tape.

The tape drive can skip ahead faster than it can read data, but if you skip ahead too far, you have rewind and try again. The classic algorithm for this problem was given a file of (n) records, first take steps of (√n) records and see what happens. This means keep the prior and current search values, so you know if you have the target value in that neighborhood. Once you locate the neighborhood, search in the other direction in steps of (√(√n)) records. Is this always good? No; the distribution of data determines performance, but if you assume unique key values with a uniform distribution, it works pretty good.

What do you notice about the cardinality (n) and the expected and worst seek time? The first thing is that seek time will increase with the cardinality. Secondly, if you are not using the data in the one and only available sorted order, you are dead. This second factor sometimes lead to data redundancy – the same data would go onto two or more tapes so it could be sorted two or more different ways. The best sort times are (n log2(n)), but require multiple tape drives to get usable results.

Disk Access

The first disk files mimicked the tape files. They stored the data in sorted order based on the disk hardware, for speed and familiarity. But the reads/write heads could be positioned anywhere on the disk. This lead to ISAM (Indexed Sequential Access Method) files. A second disk file with some of the search key and the physical location of each record made up the index.

The model used to explain ISAM was the thumb or notched index on unabridged dictionaries when dictionaries were still physical books. You put your thumb into a notch on the outside edge of the binding, one notch per each letter of the alphabet, and immediate turned to the section of the dictionary that started the entries with that letter. Using the notched index analogy, you probably noticed that the paper dictionary has more words in the "S" section than the "Q" section, so the notches are not evenly spaced.

Since an index record is shorter than the original data and there were fewer index records, the system could buffer a lot of them. But this leads to some questions. We now have the cardinality of the data file and the cardinality of the index file to consider.

Most, but not all, indexes use a tree structure to speed up the search. Which type of tree is used varies from product (binary, B-tree, B+ tree, 2/3 tree, red-black, etc.). Unlike the ISAM model, we do not assume that the indexed file is in sequential order. The leaf nodes of the tree will have a pointer to the location of the desired record.

An index can be built on more than one column and the cardinality of each column will determine the shape of the tree. That is, "CREATE INDEX Foo ON MyTable (a, b);" versus "CREATE INDEX Foo ON MyTable (b, a);" will build different trees. The one trick that people miss is that the sort order of each column can be ascending or descending order. Most indexes default to ascending order, but if you are dealing with temporal data, you probably want the most recent data (descending order) first. This can make a huge difference in performance.

Consider a sex_code versus a birth_date The index that starts with the sex_code will be more like bush than a tree, since it will have only four branches at the root ('male', 'female', lawful person' and 'unknown'). The index that uses a birth_date can a hundred or more branches next to the root. The real question is how is the data accessed for aggregations or individual rows. Oh, it might be a good ideas to have both of these indexes for reporting.

Some queries only need the index and not its base table. This is called a covering index. But some SQL products allow you to include other columns that are not used to build the tree structure of the index. Does this mean that the "cardinality" is the depth of the tree, because that it how many disk accesses we need to find a record?

The idea is that any tree will be faster than a full table scan, but this is not true. At some point, the extra disk access to the index will cost more than a table scan.

Logical and Physical

When we were first creating the SQL language, we added simple aggregate functions as far back as the SQL-86 Standard. These functions include the extrema functions (MIN() and MAX()) and arithmetic functions (AVG() and SUM()). But we also made a design mistake with two kinds of COUNT() functions.

The COUNT([DISTINCT | ALL] <expression>) function behaves like the other aggregate functions. The <expression> can be numeric, temporal or string. It is computed and generates a set of values. This is at the column level. If the optional DISTINCT is used, then redundant duplicates are removed (that means we leave one copy of the value in the set). Finally, the function drops all of the NULLs.

This group of functions counts logical rows, not physical records. While this is important, the physical records have more to do with query performance than the logical counts.

But wait! Unlike the old file systems, SQL has virtual data. It can be a computed column within a table, defined by an expression in the DDL. There are two choices; the column can be materialized and inserted with row or the SQL engine can wait until the row is invoked. In the later case, it does not exist until run time and takes no disk space. How do you want to count it?

A VIEW is a stored query that is given a name (perhaps renaming the columns as part of the declaration). Since it does not exist until it is invoked, what is the cardinality of a VIEW? Zero? Or can the SQL engine look at the code and estimate the materialized size? Does the schema statistics include the last cardinality from a prior invocation?

Fill Factor & Cardinality

Disk systems are based on data pages, which are block of bytes that the system reads or writes and caches that data. Tapes also blocked records, but not the same way. The data pages are linked by pointers and we try to keep the pages arranged so that the read/write head moves as little as possible as it follows that pointer chain.

Disk files have a "fill factor" on the data pages. This is an amount of unused disk space on each page. The purpose is to allow data to be inserted on a page in the middle of the file, a trick that was impossible with tape. At one extreme, a disk file can have 100% fill and be static. This is a look-up table. At the other extreme, a small fill factor makes it easy to add new data. But when you want to add data on a full data page, you have to split that data across two new pages, and modify the pointer chain.

The new pages can spread all over the physical disk and require more read/write head moves. This is called fragmentation and we have utility programs to de-fragment the disk. They can take some time to do their job and lock the file while they run.

What do I use for cardinality in a query plan in this environment? The records in a simple tape file had equal importance; the cost of each read was uniform. But now, a densely packed disk file reduces the cost of accessing a record. A badly fragmented file increases access time. And a bigger data page will give you more records per access.

What Happens in the Real World

You have to trust your SQL engine to make reasonable initial guesses. This is usual safe; a modern optimizer will incorporate the CHECK() constraints, the last statistics it has and heuristics handed down from older SQL engine writers. I am not trying to make a joke with that last statement. The first community of SQL engine programmers began with Micheal Stonebraker's Ingres project and Chamberlain's System R project. When they did not know the best way to do something, they made an educated guess. But nobody had any experience.

For example, one early myth from the Ingres team was that if you a have a (<column> = <constant>) predicate and no other information, assume it is TRUE for 5% of the rows. But this makes no sense when the <column> allows only a few values, like a sex_code. Today, we keep statistics and get cardinality estimates of the values within a column from the SQL engine.

The danger is that when the statistics are seriously out of date, the optimizer makes a bad plan. It can ignore a useful index because it thinks there is not enough qualifying data and a table scan will be quicker. On the other hand, it can pick a useless index because it thinks a table scan will be more expensive.

But there two more problems that need to be considered; the number of sessions using the table and the transaction level used by each session.

While a tape is hung on one drive and used by one program, a database is shared simultaneously among many user sessions. If a low cardinality table is accessed by one user session many times or by many user sessions at once, or (worse case) accessed a lot by everyone at the same time, then you can get a 'hot spot" or "choke point" that destroys performance. This problem is hard to find because we think that only high cardinality tables will have access problems. Cardinality has gone form addition to multiplication.

In SQL, you can set the transaction level for a user session. This option exposes a subset of the rows in the workspace for a table to the user session. You will always show the persisted rows as part of the cardinality, but you can also include the committed rows (everyone's work or just this user session's new work) and the uncommitted rows

(everyone's work or just this user session). If work is rolled back at the end of any user session, did it really exist at all? How do you count such rows?

Let me make this more concrete. Your database is used for the automobile license tags of a state which is divided into many counties. Each county issues tags and submits its data as it completes its process. The few urban counties have 70-80% of the state's population; the rural countries have most of the farm equipment.

The first batch of data that comes into the database is a few rural counties with a small population. This defines the query plan. We now expect that 30% of the tags are issued for heavy trucks and 10% are for farm equipment. All I need is an index on the county codes for my reports; table scans do the work.

Then the first major urban area sends it data. It is almost all passenger cars, taxis, SUVs, delivery truck and construction equipment. Virtually no farm equipment; nobody in the big city drives a corn harvester to work. Now I want to use the "tag_type" codes for reporting and that index is very important.

Mother Celko's Index Heuristics

By now, you have see that cardinality is not just a simple count of physical data. In fact, an SQL optimizer has formulas with at least a dozen parameters, well beyond what a human being can compute in his head.

Then for the computer science majors, we have a proof that picking "secondary indexes" (those are indexes used to improve access; a primary index enforces uniqueness in keys and is not optional) is known to be NP-Complete. We are doomed before we start!

Do not despair. We have heuristics we can use. A heuristic is defined as "what you do next when you have no idea what to do next", and it is not guaranteed to be optimal. Here is my list:

1.- Do not blindly create indexes. It is too often easy to add an index that makes one query or statement faster, but now every other statement has the extra overhead of maintaining that index, every other query has to consider it in its execution plan. Think globally and act locally. Take one for the team. Fill in your cliché here.

2.- Drop unused indexes. There are utility programs that can help find them. They were probably created for a one-off situation (see prior bullet point) or for an old schema. Or just because programmers will test on production data bases. Programmer are afraid to drop any thing ("nothing more permanent in programming than an undocumented temporary patch; nobody want to touch it" – programmer folklore)

3.- Consider new indexes. There are utility programs that can help suggest them.

4.- Drop redundant indexes. If you have an index on (a, b, c), this implies that the (a,b) index is redundant. If you have two indexes with the same columns in the same order, them one is redundant.

5.- Consider if an index should have an INCLUDE column rather than making ti part of the tree structure.... .This might not be an option in your SQL product.

6.- When you do a big data load, re-compute the statistics. Even if you think that the new load is typical data without any weird skew. You might be wrong. Skew can be subtle. The optimizer might have changed. You are spending time off-line to do the load, so sneak this into the mix while it is cheap.

7.-Put lots of CHECK() constraints on a table. Their predicates are picked up by the optimizer.

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.