SQL Tuning – Reading Recent Data Fast

Dan Tow
singingsql.com

Introduction

Time is the key to SQL tuning, in two respects: Query execution time is the key measure of a tuned query, the only measure that matters directly to the user, and time is also the key dimension for data pertaining to the large business-event tables that tend to dominate query tuning problems. Specifically, real-world business queries normally look for recent business events, events that are relevant to decisions and tasks that the business needs right away. Data that is even three months old is often the business equivalent of "ancient history."

Data types

Business data, and the tables that hold it, fall in two categories: Relatively unchanging reference data, such as data on customers, products, regions, employees, et cetera, and data on business *events*, such as orders, payments, invoices, customer interactions, et cetera. The events-type tables tend to grow fastest, become largest, and dominate SQL tuning problems, partly because they are so large as to be hard to cache. Fortunately, as we will see, it is unnecessary to cache all events, only the most recent ones, to see excellent performance, as long as we follow design principles described in this paper.

Natural heap tables, with natural clustering of recent rows

Fortunately, Oracle's default table structure, simple heap tables, naturally places the most recent rows on top, clustered in the topmost blocks with other recent rows. This layout is not only simple for Oracle to handle at insert time, but it is also ideal for the usual scenario where almost all business users from all corners of the application continually reference those most recent rows, pertaining to the most recent events. These most recent blocks tend to end up very well cached, either by reads from other users, or by our own reads (self-caching) earlier in the query or from our own earlier queries. Recent-event master-table rows tend also to point to recent-event detail-table rows, and vice-versa, so joined-to event tables also see excellent caching when we reach them through nested loops. (The cost of such nested loops plans, in terms of actual runtime, tends to be better than optimizers estimate, owing to this surprisingly good caching of the joined-to table and index blocks.)

An analogy between good paper-based business processes, good event-based workflow processes, and good data-based applications

There are some old rules-of-thumb regarding well-designed paper-based business processes, from back when those processes dominated, rules designed to minimize inefficient paper shuffling, and to avoid paper "slipping between the cracks:"

- The paper is touched or read by the minimum possible number of people, as few times as possible.
- The paper is modified as few times as possible, by as few people as possible.
- As soon as possible, the paper is either discarded or filed away where it will likely never need to be touched again.

There are analogous rules that apply to good workflow processes, rules that are equally applicable to paperless workflows:

- The business event involves the minimum possible number of people, as few times as possible.
- The event generates as few workflow steps as possible, by as few people as possible.
- As soon as possible, all activity related to the event is completed, and the employees need never refer to the event, again, except under rare circumstances.

These paperless workflow rules can be translated into rules for how we ought to access data in a well-designed application running well-designed business processes:

- The row or rows related to a business event are touched by the database as few times as possible.
- The event-related workflow triggers as few updates as possible.
- As soon as possible, all database activity for an event-related row is completed, and the row ends up in a state where it need never again be touched by the database, except under rare circumstances.

The last bullet, above, has some corollaries, or logical consequences:

1. If some rows do *not* quickly end up in this "closed" state, but instead figure into reports months or years later, again and again, then the business process has an unintended, endless loop!
2. Purging old data should have little effect on performance, if design is ideal, because those old rows would never be touched, anyway!
3. Summarizing or reporting old events need only happen at most once, for any given event date range. Re-summarizing the same old data repeatedly implies that we either "forgot" to re-use the former result, or we suspect that history has been rewritten, both of which tend to point to a process failure!
4. A repeatedly-executed query that violates this rule, querying the same old rows with every repeat, usually points to a design flaw in the application, or a defect in the business processes, or both!

Types of Conditions

The most common conditions in WHERE clauses of queries (when we have old-fashioned queries that have not promoted join conditions into the FROM clause) are joins, usually simple matches of foreign and primary keys that let the database navigate from master rows to matching details, and vice-versa. The important conditions in tuning problems are the other conditions seen – filter conditions, which discard (or, preferably, avoid reading *in the first place*) the subsets of the data that are not needed by the query. From the perspective of this paper, there are four sorts of filter conditions, defined by their relationship to the dimension of time in the events-based tables.

Subset conditions unrelated to time

Most queries of business-events data include joins to reference tables that are more or less fixed, data that is referred to frequently in relationship to events, such as data about customers, employees, products, object types, regions, etc. Often, conditions, such as

```
AND Region.Name=:region
```

refer directly to these reference tables, not to events data, so these conditions are necessarily removed from the dimension of time, referring to a filter that would apply equally to old and new events. Conditions such as this can be applied late in the join order, if they are not very selective, without much inefficiency in terms of effort spent by the database reading rows that are later discarded. Alternatively, if the condition is guaranteed to read just a single row, it is safe to read that rows up-front, at the start of the join order, even in a Cartesian join. From this first, unique condition, we might reach related events with a highly-selective join to a foreign key, such as a read of orders by a single customer, or, better still, we could reach related *recent*-events with nested loops to a concatenated index on both the foreign key and on some time-related column, such as a read of open orders by a specific customer, or recently-shipped orders by a specific customer. When the condition on reference data is not unique, nor very selective in terms of the fraction of events it would point to, we may still choose a hash join of an early read of the reference data and an independent read of just the recent events.

Time-independent conditions can reference time-independent columns of events-type tables, too, though, such as

```
AND Order.Order_Type='INTERNAL'
```

These time-independent conditions on events data would filter roughly the same fraction of new events and old events, so they fail to correlate even approximately with the age of the event. Conditions like this usually make poor driving conditions, by themselves, since they point to progressively more rows as history accumulates, but these columns may be useful as part of a concatenated filter index, where some other column of the index has the greater selectivity of the condition that points especially to recent rows.

When a WHERE clause contains only filter conditions unrelated to time, the query will return a rowcount that grows steadily according to the amount of history that has accumulated in the database. Such a query result may look reasonable when the application is young, but will grow steadily more cumbersome as the application ages, reporting the same old, tired data over and over again, and likely becoming far too slow, with a high rowcount that is too large for a human to properly *digest*.

Conditions explicitly restricting a date range for the events

A type of event may have several dates, each relating to some workflow event in the process of handling the business event. For example, an order may be created on one date, booked on another, shipped on a third, received on a fourth, and paid-for on a fifth. Reports commonly restrict event data based on some date range for one of these workflow event dates, for example, orders shipped (at least partially) in the past week. These conditions may not look very selective to application developers building a new application, with little history in their "toy" development databases, but in real applications, in production use, these conditions grow steadily more selective the more history accumulates. These date columns can be useful to index, often in combination with other columns that further restrict the subset of data desired in that date range. However, when these date columns are indexed in multi-column indexes, they should usually be the last column in the index, since the date condition is almost always a range, rather than an equality, and this prevents use of any columns following the date column to narrow the index range scan.

Conditions on events' workflow status

An example of a workflow-status condition would appear in a search for open orders ready to ship, a search likely performed specifically in order to perform that next workflow step on the ready orders. If such a query reads the same row more than once, for the same workflow step, however, this tends to point to something broken in the process; if the order is *really* ready to ship, it should *ship* the first time we report it for that purpose! An efficient workflow, on the other hand, will promptly complete processing related to business events, soon placing the event into the final, "closed" state, in which it requires no further attention from the business, except possibly one final summary of completed work in the latest time period.

Because open events (events requiring further processing) should be recent, the open event statuses should be selective, once a reasonable amount of history accumulates, justifying indexes on these event-status columns. Since there are relatively few steps in most workflows, though, there are likely to be few distinct workflow status values, so the optimizer won't recognize the high selectivity of the open status values unless we also generate histograms on these columns.

One special case of a workflow-status search looks specifically for unusual cases (which ideally should not happen) where the event is both old and open. The optimizer would normally estimate (in the absence of dynamic sampling) that the combination of an open status condition and a condition on a range covering all old dates would only be slightly more selective (since almost all dates stored are old) than the status condition, alone. In fact, the combination may be super-rare, or even point to no rows at all, so an execution plan reaching both conditions early, then following nested loops to the rest of the tables, will be best, although manual tuning might be required to get this plan, since the optimizer won't see the anti-correlation between these conditions. These searches for old, open events can be useful to discover where the business processes allow work to "slip between the cracks," failing to be processed in a timely fashion. Of course, the correct response to such cases is to fix the root-cause problems in the business process, making such cases even rarer.

Conditions defining data for a single event

An example of a single-event query would read the master and detail data, for example, pertaining to a single client visit, or to a single order. All this data was likely created at or very close to the time of the triggering event, so it will be well-clustered around that point in time. It is less-obvious, simply looking at the query, that the data is *recent*, but in the common course of running a business, users are far more likely to trigger queries of recent events than of old ones, so almost all of these queries will read recent data, though nothing in the WHERE clause appears to guarantee that. The usual driving condition on such a query will be a primary-key value (usually some arbitrary ID) pointing to a single row of the master-level event table mapping one-to-one to the single event. From this ID, we can follow indexed foreign keys into any necessary detail tables, using nested loops, for a very fast execution plan.

"Good-citizen" queries

Queries driving from conditions that specifically tend to reach recent rows not only benefit from the likelihood that such rows are well-cached, they tend to reinforce the useful tendency to hold recent rows in cache, acting as "good citizens," in a sense, in the community of queries. On the other hand, queries that drive to event-type tables using conditions that apply equally to old and new events (such as a condition specifying a Region_ID) bring old rows into the cache that are unlikely to be useful to any other queries, tending to flush the more-useful recent rows from the cache. Thus, such queries not only run long, since they find poor caching, they harm the performance of other, better-tuned queries that reach recent rows more directly.

Exceptions: good reasons to read old data, rarely

There are a few good reasons to read old data:

- Looking for new ways that workflow items are "slipping between the cracks," staying in the workflow longer than the processes should allow. (This applies only to *moderately old* data, ideally, because the *old* ways for workflow items to slip between the cracks should already have been fixed.)
- Reorganizing the database schema for a new version of the application.
- Data-mining old data in new ways that were not formerly tried, to gain new insights. (For example: "Maybe we could predict… if we looked at the old trend for … in a new way.")
- Handling rare business exceptions, such as lawsuits, or unusual customer problems.
- Handling repetitive business, such as automated annual renewals (but this would only be *moderately* old data). Note, however, that this category of query will *still* tend to reach a narrow date range, although the narrow date range might be from a year ago and a year plus one day ago, for a daily process performing annual renewals, for example.

Conclusions and summary

There are two primary principles that guide the use of the time dimension in event-type data when tuning SQL:

- Queries should rarely return rows relating to old events.
- Queries should not even *touch* old-event data early in the execution plan, even if that data is discarded later in the plan, with rare exceptions.

From these guiding principles we can conclude several specific, useful rules:

- The index used to reach the first event-related table in the join order should use some column condition correlating to recent rows (potentially combined with conditions unrelated to time, if a multi-column index applies).
- The rest of the event-related tables should be reached, usually, with nested loops to join keys, reaching related recent master and detail data for the same global recent events.
- Time-correlated conditions pointing to recent rows see far better clustering and caching than non-time-correlated conditions with similar selectivity, so drive to recent rows first, then filter on non-time-dependent conditions, unless the non-time-dependent conditions are much more selective.
- Nested-loops joins between master and detail event-type heap tables tend to join recent rows to recent rows, and see much better caching on the joined-to table and index blocks than the optimizer anticipates.

- Nested loops are usually faster than they look, and faster than the optimizer *estimates*.
- Queries repeatedly returning the same old event-type rows show application design flaws (such as reports of unimportant data) or business-process design flaws (such as workflow items getting "stuck" in a process loop that fails to resolve), or both.
- Queries touching old event data early in the execution plan, then discarding it later in the plan, tend to indicate poor join orders, or poor join methods (hash joins that should be nested-loops joins, especially), or non-robust plans (plans that are only OK because the tables have not grown to mature size), or poor indexes.
- Good application design and good process design do not rewrite history, and do not re-summarize the same history repeatedly.
- Purging old data should have almost no effect on day-to-day performance if the application is well-designed and the query execution plans are well-tuned and robust. (Purging can save disk and make backups, recoveries, conversions, and other DBA tasks easier and faster, though.)
- *Although* purging old data should have almost no effect on day-to-day performance if the application is well-designed and the query execution plans are well-tuned and robust, correctly designed applications and processes should almost never touch old data, making such purges relatively safe and easy.
- A "read trigger" would be a useful innovation, here – "Notify me if, contrary to expectations, anyone ever reads these rows…" Such a trigger could discover failures to follow the recent-rows principles, and these failures would in turn point to opportunities for improvement in the application design, business-processes design, schema design, and specific poorly-tuned SQL.
- The natural data layout of simple heap tables is ideal for event-type tables, naturally clustering hot, recent rows together at the top – *don't mess with this useful natural result!* (Rebuilding heap tables with parallel threads is one way to shuffle recent rows in among old rows, with disastrous results to caching and performance!)