SQL Tuning – Reaching Recent Data Fast

dantow@singingsql.com
www.singingsql.com

Time is the Key to Tuning

- Obvious Point: Low-run*time* of the SQL is the key goal (not logical I/Os, physical I/Os,…)
- Subtler Point: The data has a time-dimension, too, and well-designed, normal business queries mainly read *recent* data.

Business Table (and Entity) Types

- Reference tables: Customers, Products, etc., entities the business references in its interactions. These tables grow relatively slowly, and tend to be well-cached.
- “Events” tables: Orders, Payments, Invoices, Customer Interactions. These tables tend to grow fastest, and to dominate query-tuning problems. Sensible queries almost always concern *recent* business events, with reference to non-event entities.

Heap "Organization"

- What does a typical Oracle heap table have in common with an archeologist's midden heap?
- Both have the most-recent *stuff* on top!
- Archeologists want to see old stuff, which requires slow digging, but businesses generally focus on *new data*, the "top layer", so to speak, which is often conveniently available in the cache.

"Excavating" the Heap Table

- A well-designed query and execution plan won't "dig deep" just to see last-week's data!
- This usually implies that you will drive from a condition that correlates somehow to a recent event or to a set of recent events, avoiding a path that touches data for old events.

Optimal Paper-Based Business Process Rules

- 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.

Optimal Event-Based Business Process Rules

- 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.

Optimal Data-Based Business Process Rules

- The row(s) 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.

Optimal Data-Based Business Process Rules

- *As soon as possible… the row ends up in a state where it need never again be touched by the database, except under rare circumstances.*
  - Corollary #1: If some rows do *not* 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!
  - Corollary #2: Purging old data should have little effect on performance, if design is ideal, because those old rows would never be touched, anyway!

Optimal Data-Based Business Process Rules

- *As soon as possible… the row ends up in a state where it need never again be touched by the database, except under rare circumstances.*
  - Corollary #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!

Optimal Data-Based Business Process Rules

- *As soon as possible… the row ends up in a state where it need never again be touched by the database, except under rare circumstances.*
  - Corollary #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

- Joins – usually matches between primary keys and foreign keys.
- Filter conditions – anything that isn't a join; these conditions are the proper focus when solving tuning problems.

Types of *Filter* Conditions

- Fixed-fraction subsets, unrelated to time, often restrictions based on non-event reference data.
- Date-range restrictions specifically related to a date pertaining to the events.
- Conditions on the workflow-related status of the events.
- Conditions defining data for a *single*, *specific* event.

Types of Conditions

- Fixed-fraction subsets, unrelated to time, often restrictions based on non-event reference data.
  - E.g., “North American orders”, “orders from May’s Diner”
  - These conditions are usually *combined* with some other condition(s) correlating to recent events, else the query would eventually return very old rows unlikely to be useful, and often would return far too many rows to “digest,” and too many to perform well. *Fix queries that don’t make sense in light of this!*
  - Concatenated indexes can combine the selectivity of time-related conditions and reference data.

Types of Conditions

- Date-range restrictions specifically related to a date pertaining to the events.
  - E.g., “Orders shipped in the past week,” “This month’s customer complaints”
  - These begin as unselective, before history accumulates, but become highly selective as history piles up – design for the old system, not for the new!
  - These date columns are useful to index, but they should usually be the last column of the index, because they are reached with range conditions, not with equalities.

Types of Conditions

- Conditions on the workflow-related status of the events.
  - E.g., “Open, Ready-to-ship orders”
  - These are ideal to reach rows requiring the next step in the workflow process, once, but if we reach those rows more than once following such a condition, the process is inefficient!
  - Efficient workflow implies these are *recent* rows!
  - These usually require an index *and* a histogram, so the optimizer “knows” the few-valued “status” column is highly selective for the “open” statuses.

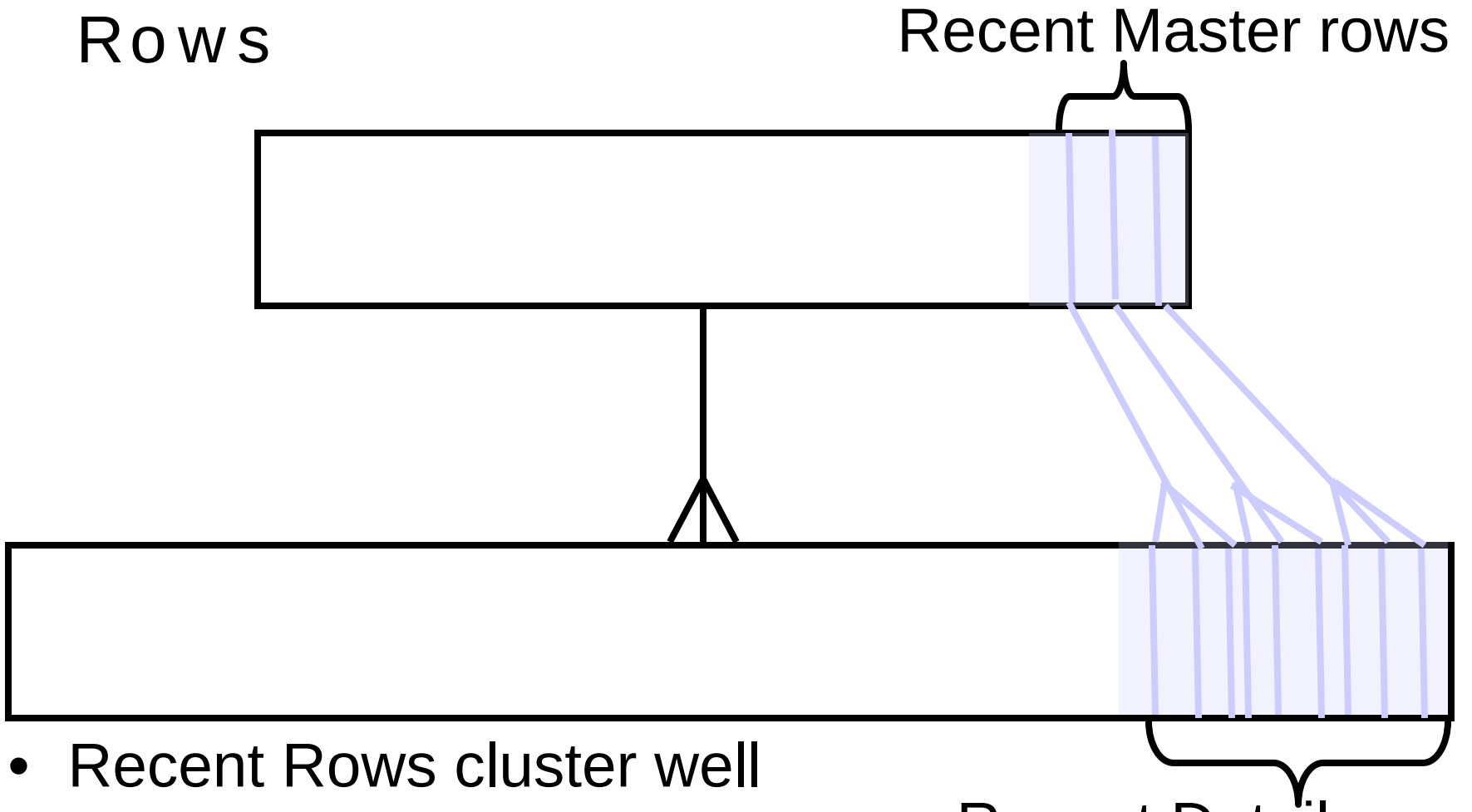
Types of Conditions

- Date-range restrictions specifically related to a date pertaining to the events *combined with*:
- Conditions related to the workflow-related status of the events.
  - Special Case: “Old event” and “Event still in an open workflow state,” combined, should be a rare, special exception belonging to two anti-correlated subsets (cost-based optimizers don’t handle anti-correlation well). Active monitoring for exceptions like this should point to process failures, if they exist, and opportunities for process improvements.

Types of Conditions

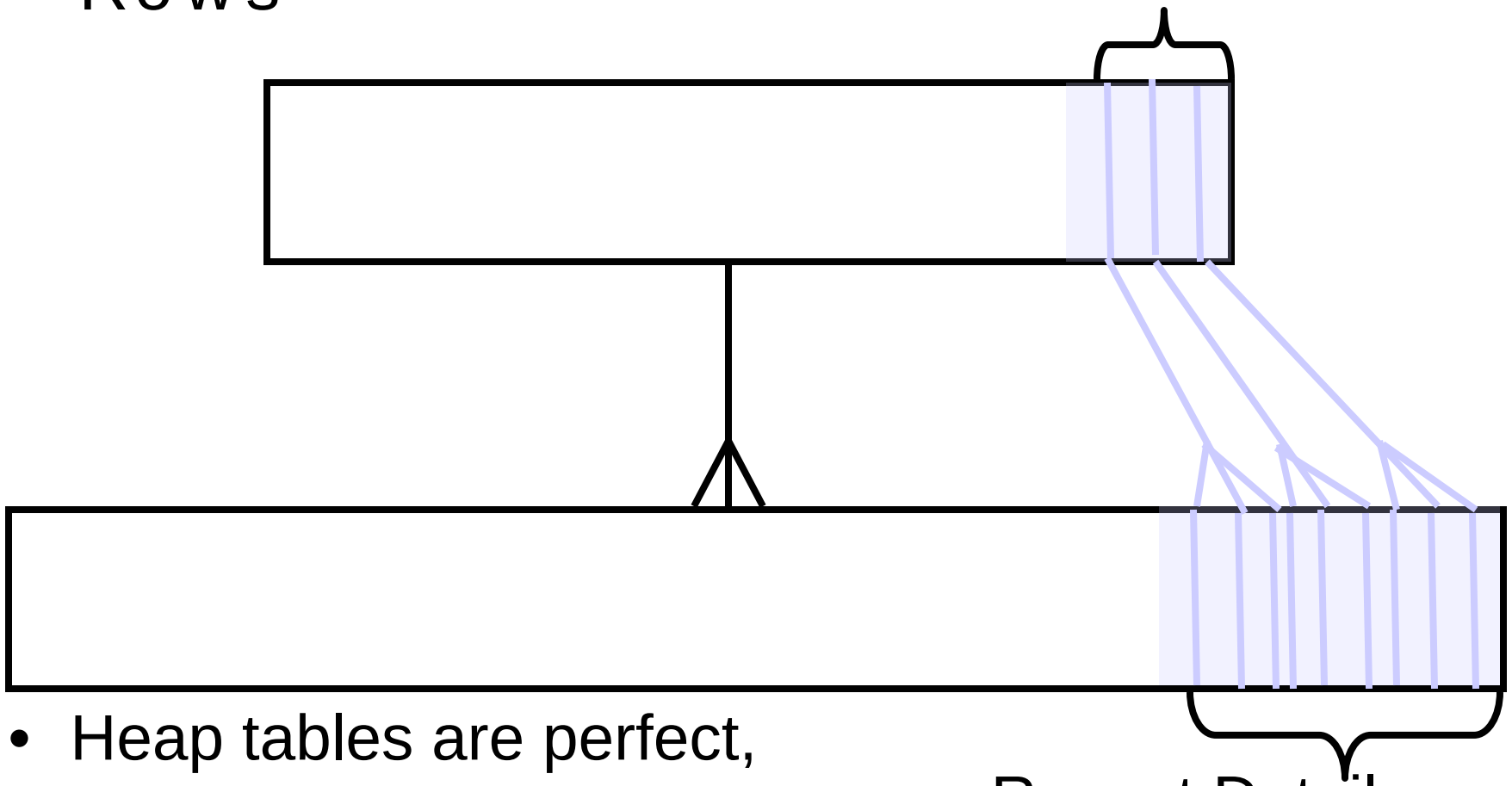
- Conditions defining data for a *single*, *specific* event.
  - E.g., “Data for a specific client visit for service”
  - These should normally drive from an indexed primary or foreign key that uniquely points to a row mapping to that event.
  - The details of that event will need to be reached through indexed foreign keys, with nested loops.
  - The “*recentness*” of the queried data may be non-obvious, but it is almost invariably the case that the specific event queried will be a recent one.

The Advantage of Driving to Recent Rows



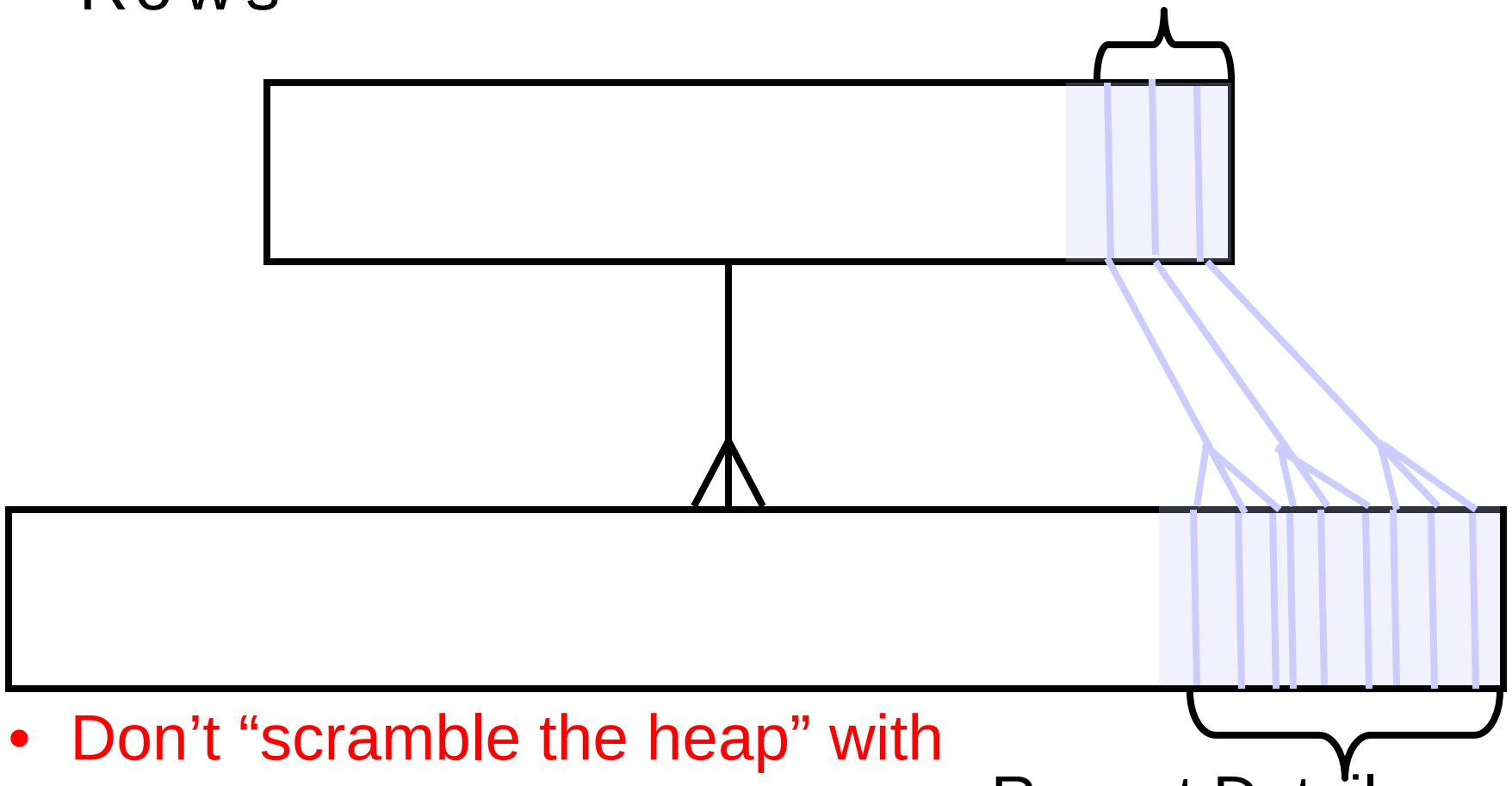
- Recent Rows cluster well and are well-cached

The Advantage of Driving to Recent Rows



- Heap tables are perfect, here!

The Advantage of Driving to Recent Rows



- Don't "scramble the heap" with (for example) a parallel rebuild!

"Good-Citizen" Queries

- Queries driving first to recent-events data benefit most from the tendency for this data to be cached.
- Queries driving first to recent-events data reinforce the tendency for this data to be cached.
- Queries reaching old data, on the other hand, tend to *flush* reusable data from the cache – these are *bad citizens* in the "query community."

A Few *Good* Reasons to Read or Modify 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.

A Few *Good* Reasons to Read or Modify Old Data

- 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).

Summarizing Principles

- 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.

Conclusions

- 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.

Conclusions

- 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.

Conclusions

- 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*.

Conclusions

- 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.

Conclusions

- 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.

Conclusions

- 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, other DBA tasks easier and faster, though.)

Conclusions

- *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…”

Conclusions

- 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!)

Questions?