EFFECTS OF PARTITIONING AND PHYSICAL ORDERING ON PERFORMANCE WITH MULTI-ORG ORACLE FINANCIALS

Mark W. Farnham
Rightsizing, Inc.

Introduction

Many financials reports need rows from only one of the possibly many organizations set up in the e-Business suite. By default a given database block may contain a relatively sparse set of rows with respect to any particular organization, so many more blocks are read than if each block only had rows you need. Since many of the tables contain only unique identifiers rather than the organization natural key, establishing the desired cohesion of organization rows to blocks is not trivial but is often worthwhile.

The heart of the matter is that Oracle stores rows in blocks. When you run a query, what you want are the right rows. The fewer blocks you need to retrieve a read consistent image of in order to get the columns you want of the rows you want, the less it costs per row.

If there is a dominant order in which you structure your financial reports, it can be profitable to maintain that physical order even if you do not have multiple organizations or charts of accounts.

If I write this paper well, then by the end of reading it you will:

1. Be aware that Oracle stores rows in blocks.
2. Be aware that the default setup of Multi-org tends to make desired rows for a particular organization sparse in blocks.
3. Understand the gl_code_combinations relationship to account ids.
4. Understand how the physical order of gl_code_combinations can affect cohesion of rows from each organization to blocks in gl_balances.
5. Take a glimpse at how the advent of “REF” partitioning in RDBMS 11g may finally allow a very useful partitioning of gl_balances and gl_je_lines.

Oracle Stores Rows in Blocks

Now In a logical sense the data in a relational table consists of rows containing values (or nulls) in one or more columns. By the definition of relational rules, there is no guarantee of the order rows exist in a table. When a small percentage of the rows in a table is to be returned based on a restriction in the values of one or more columns, it therefore makes sense at the logical level that work is saved if an index exists on the value restricted columns so that the desired rows can be found and delivered to the result set.

If you think through the logical steps done to get a row directly from the table, you get one step per row:

1. Read the row from the table.

If you think through the logical steps done to get a row via an index, you get two steps per row:

1. Read the location of the row from the index.
2. Read the row from the table using the location from the index.

When reading directly from the table, of course, you must read each row to determine whether you need to put the row into the result set. When reading via the index, some of the predicate restrictions on column values may be applied to reduce the number of row locations retrieved from the index. So at first blush and from a logical sense, you might think you would save work whenever the fraction of rows retrieved by going through the index is less than half the rows in the table.

But in any physical implementation of a relational database management system (RDBMS), such as Oracle, you would be wrong. Rows in a physical implementation are not disembodied tuples in the logical ether, but must be stored somewhere. Oracle stores each row of a table in one or more database blocks. Indexes are also stored physically with the branches and leaves each being stored in database blocks.

So my straw man depiction of the logical process above must be corrected significantly. In fact no one that I am aware of predicts such a generous row selectivity threshold for when it is likely to be good to use an index rather than a table scan. (Bitmap indexes are beyond the scope of this paper, and you should probably avoid embarking on applying bitmap indexes to Oracle Applications unless you really understand a particular case and the overall implications. In which case you should write a paper about it and let me know.)

The quick fix up to the straw man is that indexes have “height.”

The basic idea is that a flat index is like a phone book page (well okay, just the names and a row_id to the row holding the rest of the data, but keep this image in mind). This is called a leaf. When you get too much for a single leaf, you get another level, where the "branch" level gives you the high and low values and a block address for the leaves. When you get so many leaf references that one block on this next level can't hold them all, you get another level, and this continues on as the number of rows in a table grows. (Fortunately, the height grows very slowly, since you can fit a lot of high/low range entries with a block address per block.) How many references per leaf is proportional to the usable space per database block divided by the sum of the lengths of the indexed column values plus the length of the row_id. For tables with many, many rows, this is one of the major reasons why large block size is a usually a good thing for database performance. The bigger the block, the lower the minimum possible height for the same rows and index. But there is height, so at the conceptual row level you must penalize the index access path for doing “height” reads per reference to get the location of the row. So pretty quickly the conceptual row selectivity requirement to gain from reading via the index goes from 50% to 33% to 25% to 20% to 16% and so forth as the index height grows. So at second blush and from a logical sense, you might think you would save work whenever the fraction of rows retrieved by going through the index is less than one divided by (height + 1).

But in any physical implementation of an RDBMS, such as Oracle, you would be wrong. Rows in a physical implementation are not disembodied tuples in the logical ether, but must be stored somewhere. Oracle stores each row of a table in one or more database blocks. Indexes are also stored physically with the branches and leaves each being stored in database blocks. Yes, I repeated myself. Furthermore, there can be a huge effect from buffer caches at several levels from the disk farm to the operating system to the Oracle system global area (SGA). So what is important is not row selectivity, but rather block selectivity. Quickly consider 100 rows in ten database blocks, and that you want to return 10 of these rows. Depending on which rows you want and in which blocks the rows are stored, your block selectivity can range from a single block to all the blocks. Now 100 rows in 10 blocks where you want 10 of the rows is a tiny example so you can picture it quickly. If your real problems are on that scale you have probably wasted time reading this far. You’ll get all the blocks in cache in a blink and they will probably stay in cache so how you get at them is pretty irrelevant. But you are probably reading this paper because you have one or both of the following challenges:

1. Limited total cache compared to the blocks needing to be read to service row requests from queries.
2. Large queries where the time to read the blocks is significant.

In the first case, you care about block selectivity because the fewer blocks you read, the fewer other blocks you have to kick out of the cache and re-read later when another query runs. So improving block selectivity improves the chance that queries

will run at cache speed rather than at physical device speed, or will reduce the amount of cache you need to buy to reach your desired service level.

In the second case the elapsed time of producing the result set of the query is likely to be dominated by the number of blocks you need to read, since you are likely to be reading many of them from physical devices. Even if the physical read times are negligible, the logical IO time to project a read consistent block from which to pluck the row you need must be considered.

So at third blush and in the case of real physical implementations, you might think you would save work whenever the number of database blocks read is minimized, and that is about the best you can do.

But we still have not gone far enough, since I have not given you a way to improve block selectivity, and there is an optimization beyond block selectivity that is often significant: Ordered block selectivity. When in case two above many of the rows referenced consecutively via the index are in the same block, the chances to read successive rows from a block that is in one of the various levels of cache rises dramatically. Some disk farm hardware reads and caches significantly more than the multiblock read request from Oracle. Likewise with the operating system level cache in some operating systems. And if you indeed have just read an Oracle database block containing the next several rows you are about to request via the index, the probability closely approaches 1 that the block will still be in the SGA block buffers when you ask for that row. Further, the closer in time this occurs, the more likely that the block will require no read consistency “fixups.”

What about Multi-org?

At most Oracle Applications sites, account code combinations get created over time with no correlation whatsoever between the order they are created and the segment values they contain. Since this means that the rows are inserted into whatever block is next up on the free list (or an ASSM insertion point), your chances for block selectivity using an index on GL_CODE_COMBINATIONS is at best random. Since a large percentage of the load on an Oracle Applications Financials database is often from Journal Import and from Financials Statement Generator (FSG) reports, this is usually a serious candidate for physical re-ordering. Even if the combinations table is relatively small, the effects on subsequent GL Open Period record creation can be significant. If your accounting team has a predominant order for presenting FSG or other Financial reports, then you can usually arrange for this to be a reasonable order to be used by the Journal Import Process. This can then make both Journal Import and Financials reporting run faster, because block selectivity on reports is improved, and Journal Import can end up being a single pass through GL_CODE_COMBINATIONS. Your mileage may vary. An interesting side note is that once reordered, subsequent GL Open Period operations apparently create the GL_BALANCES rows for the new period in the matching order, so any joins will require the minimum number of block reads from that point forward. But if you have Multi-org or even just multiple sets of books, just separating out the different organizations can be a tremendous win. In the example before with 100 rows in ten database blocks, consider that they represent 10 organizations, each the same size. If you don’t do anything about it the ten rows you are interested in and are allowed to see will be mixed up in blocks with 90 rows you are not allowed to see. The organization setting for your logon will correctly prevent you from seeing those rows, but in order to get the rows you want to see you will have to read many more blocks than if your organization’s rows were segregated into their own blocks.

Oh, and I better mention that it is possible to degrade the block selectivity of one index access order when you optimize another. So if one order of access dominates, or if the significant orders of access are not in strong negative correlation, you will not hurt anything. But if two or more index access paths are co-dominant and have a strong negative correlation, then it is possible to make one path slower than random when you optimize another path. In practice I have not seen this happen. Usually you get a significant improvement in block selectivity when using an index having columns matching from the left to the rebuild order and essentially no change from random with respect to other indexes. The most likely exception is disturbing useful ordering by time of creation. For example, if you reordered GL_JE_LINES in the same order as a dominant order for GL_CODE_COMBINATIONS and then ran a trial balance report on a single period you will likely have mixed up the rows you want with lots of rows for the same code combination from other periods. So reordering requires some thought and analysis. If you reordered GL_JE_LINES by period first and then in the predominant order for code combination access you would very likely have a big win in many cases and no significant change from random in the other cases.

Another thing that affects block selectivity is the dynamic tension between densely filled blocks (good) and row migration (bad). The smaller your pctfree for a table, the more likely it is that an update to a row which causes an increase in the length of the row will cause row migration. Row migration is to be avoided since you pretty much read an extra database block every time you get a migrated row via an index. Worse, the block location of the new version of the row has no correlation with the ordering you have done, so it is likely that you will be going to a physical device rather than cache to get the row. However, since much of the data in an Oracle Applications database is accessed and becomes final based on date of creation or accounting period, it is possible for many tables to optimize this relationship by breaking up the reordered reload into pieces by date or period. For example, perhaps you have five years of General Ledger data on line, but the three oldest years are absolutely frozen from change and last year is unlikely to change much. You might then consider reloading the oldest three years with an absolute minimum of pctfree, a very small amount of pctfree for last year, and the usual amount for the current year. By getting more rows per block on the rows that are unlikely to change, you not only save size, but likely increase block selectivity when reports on older data are made. But on the younger rows, where the length of the row may still be in flux, you reduce the tendency toward row migration. In some cases it is possible to determine that rows in a table cannot change in length. For such tables rebuilding with a minimum pctfree gets you a big win.

GL_CODE_COMBINATIONS and Accounts

For each set of books you define a certain number of flex fields are used to make up the account. A relationship is maintained in the table FND_ID_FLEX_SEGMENTS that relates your name for the segments of your account and the name of the column in gl_code_combinations that contains the data for each row for a given set of books. Each account, though, has a unique identifier “code_combinations_id” that is carried into all the tables that need to reference accounts. If the textual name of that account is required, then it is picked up by reference through the code_combinations_id and the flex segments definition for that set of books to get back the segments. Thus the big transaction and balance tables carry only the id and not the natural keys. Now prior to RDBMS release 11g, that poses a problem for trying to partition, since the keys you need are not in the data. In 11g, though, there is “REF” partitioning, so it is possible to partition the big tables by reference to a column in the relatively small gl_code_combinations table.

Prior to certification of the applications on 11g, though, the fundamental way to untangle the data is physical sorting. The good news is that you probably do not ever need to sort periods more than once. And you may be able to avoid sorting the big tables altogether.

As long as gl_code_combinations is ordered physically so that when it is scanned as a reference for which accounts to create in the balance tables for the new period, it gets one organization at a time (preferably in the dominantly used order for financials reports), then the new period will be in order. Since interactively entered transaction rows are usually entered one organization at a time, the row to block cohesion is usually good from that source, and you may elect to physically order journal batches before importing them from an external source.

In this way you may find that your reports run a bit faster while putting less stress on your hardware because fewer blocks have to be read to get the rows you need for each report in each organization.