

Pro Oracle SQL Development

Best Practices for Writing Advanced
Queries

Second Edition

Jon Heller

Apress®

**You are viewing a courtesy copy of chapter 16
from Jon Heller's book, *Pro Oracle SQL*
*Development***

Would you like a copy of the complete book?

Purchase now from Amazon:

**[https://www.amazon.com/Pro-Oracle-SQL-
Development-Practices/dp/1484288661](https://www.amazon.com/Pro-Oracle-SQL-Development-Practices/dp/1484288661)**

Or from Apress.com:

**[https://link.springer.com/book/10.1007/978-1-
4842-8867-2](https://link.springer.com/book/10.1007/978-1-4842-8867-2)**

Thank you for your support of the author and his work

CHAPTER 16

Understand SQL Performance with Algorithm Analysis

Solving Oracle SQL performance issues is the pinnacle of SQL development.

Performance tuning requires a combination of all the skills previously discussed in this book. We need to understand the development process (to know why problems happened and weren't caught sooner), advanced features (to find alternative ways to implement code), and programming styles (in order to understand the code and rewrite it into something better).

Programming is one of the few fields with order-of-magnitude difference in skill level between professionals. We've all had to deal with coworkers who are only one-tenth as productive as us, and we've all been humbled by developers whose code is ten times better than ours. With performance tuning these numbers get even higher, and the rewards are greater. It's exhilarating when we make a small tweak and something runs a million times faster.

But “a million times faster” is only the beginning of this story. All SQL tuning guides discuss run *times*, but none of them consider the run-time *complexity*. Database performance tuning requires more than an encyclopedic knowledge of obscure features and arcane settings. Performance tuning calls for a different mindset.

Algorithm analysis is the most underused approach to understanding Oracle performance problems. This chapter tells the story of Oracle performance through the lens of simple algorithm analysis. The sections are not listed in a traditional order, such as ordered by performance concepts and tuning tools. Instead, the sections are ordered by time complexity, from fastest to slowest.

Admittedly, algorithm analysis is not the most useful performance tuning technique. But algorithm analysis should not be relegated to the halls of academia – it should be a part of everyone’s SQL tuning toolkit. We may solve more problems with techniques like sampling and cardinality estimates, but we’ll never truly understand performance without understanding the algorithms.

Algorithm analysis can help us with both proactive and reactive tuning. For proactive tuning, we need to be aware of the advantages available if we create different data structures. For reactive tuning, we need to be able to measure the algorithms chosen by the optimizer and ensure Oracle made the right choices.

The two chapters after this one provide a more traditional approach to performance tuning; those chapters describe the different concepts and solutions used in Oracle SQL tuning. This chapter explains why the stakes are so high for the decisions Oracle must make. This material should be useful to SQL developers of any skill level.

Algorithm Analysis Introduction

With a few simple mathematical functions, we can gain a deeper understanding of the decisions and trade-offs involved in execution plan creation. Performance results are often given as simple numbers or ratios, such as “X runs in 5 seconds, and Y runs in 10 seconds.” The wall-clock time is important, but it’s more powerful to understand and explain our results with mathematical functions.

Algorithm analysis, also known as asymptotic analysis, finds a function that defines the boundary of the performance of something. This technique can apply to memory, storage, and other resources, but for our purposes we only need to consider the number of steps in an algorithm. The number of steps is correlated with run time and is an oversimplification of the overall resource utilization of a database system. This chapter ignores measuring different kinds of resource utilization and considers all database “work” to be equal.

A full explanation of algorithm analysis would require many precisely defined mathematical terms. Don’t worry – we don’t need a college course in computer science to use this approach. A simplified version of algorithm analysis is easily applied to many Oracle operations.

Let’s start with a simple, naïve way to search a database table. If we want to find a single value in a table, the simplest search technique would be to check every row. Let’s say the table has N rows. If we’re lucky, we’ll find the value after reading only 1 row. If we’re unlucky, we have to read N rows. On average, we will have to read $N/2$ rows. As the number of table rows grows, the average number of reads grows linearly.

Our naïve algorithm for searching a table has a best-case, an average-case, and a worst-case run time. If we plot the input size and the number of reads, we'll see that the worst-case performance is *bounded* by the function N . In practice, we only care about the upper bound. Worst-case run time can be labeled with the traditional Big O notation as $O(N)$. Although real-world solutions are full of constants and exceptions, we can ignore all those details and still meaningfully compare algorithms.

Figure 16-1 visualizes the worst case as a solid line, an asymptote that the real values can never exceed. Our not-so-smart search algorithm uses the dashed line, and it takes more or less steps¹ depending on exactly where the value is placed in the table. Our real-world results look like the messy dashed line, but to understand performance, we can use the simpler solid line.

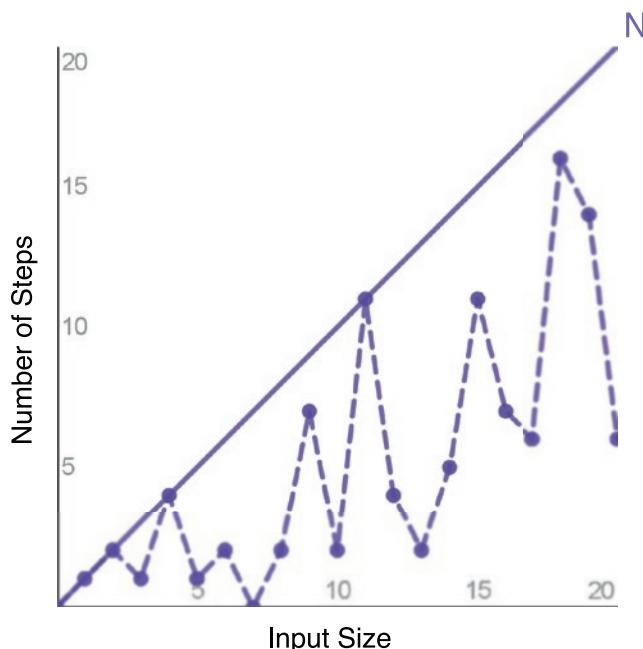


Figure 16-1. Number of steps vs. input size for linear search algorithm, including an asymptote

¹This book uses the word “steps” to refer to the work performed by algorithms. Traditionally, that value is called “operations.” But Oracle execution plans already prominently use the word “operation,” so it would be confusing if I used the standard name here. This chapter isn’t a mathematical proof, so it doesn’t matter if our terminology is nonstandard.

The functions by themselves are meaningless; it's the comparison between the functions that matters. We want Oracle to choose the functions that have the lowest number of steps on the Y axis, even as the data increases along the X axis.

If Oracle only had to compare straight lines that begin at the graph's origin, then the task would be trivial. The task becomes difficult when each algorithm's performance is represented by a different curve and the curves intersect at different points along the X axis. Stated as an abstract math problem, to know which curve has the lowest Y value, Oracle must determine the X value. Stated as a practical database problem, to know which algorithm is fastest, Oracle must accurately estimate the number of rows. Bad execution plans occur when Oracle doesn't have the right information to accurately estimate which operation is cheaper.

Figure 16-2 shows the most important functions discussed in this chapter. $O(1)$, $O(\infty)$, and Amdahl's law didn't fit together on the graph, but they are also discussed later. Luckily, the most important Oracle operations fall into a small number of categories. We don't need to look at the source code or write proofs, since most database operations are easy to classify. Spend a few moments looking at the lines and curves in Figure 16-2.

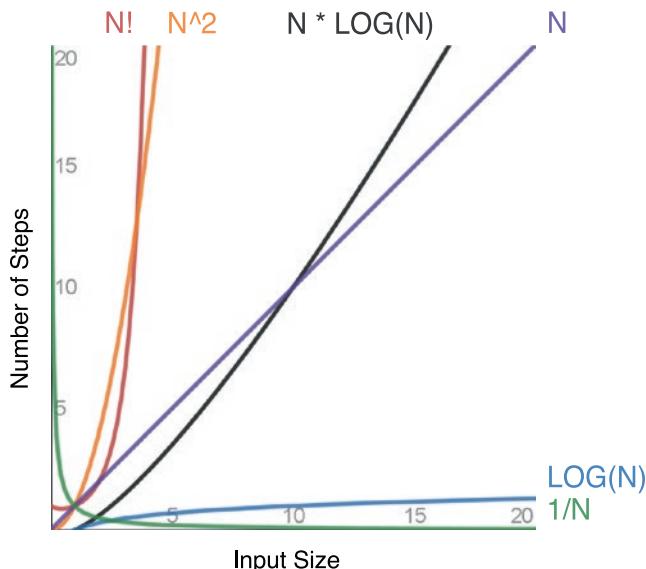


Figure 16-2. Functions that represent important Oracle operations

The next sections discuss each function, where we find them, and why they matter. Comparing the preceding shapes helps explain many Oracle performance problems.

0(1/N): Batching to Reduce Overhead

The harmonic progression $1/N$ perfectly describes the overhead of many Oracle operations. This function describes sequence caching, bulk collect limit, prefetch, arraysize, the number of subqueries in a UNION ALL, PL/SQL packages with bulk options like DBMS_OUTPUT and DBMS_SCHEDULER, and many other operations with a configurable batch size. Choosing the right batch size is vital to performance. For example, performance may be disastrous if there is a SQL-to-PL/SQL context switch for every row or if we have to wait for a network round trip for every row.

To reduce wasteful row-by-row overhead, we must combine the overhead for multiple rows. But how many rows do we combine? This is a difficult question with many trade-offs. Not combining *anything* is ridiculously slow, but combining *everything* will cause memory or parsing problems. Batching is one of the keys to good performance, so we need to think clearly about how much to batch.

Just pick 100 and stop worrying about it. We can be confident in choosing a value like 100 by understanding the following charts in Figure 16-3. The first thing to notice is that the theoretical results almost perfectly match the real-world results.

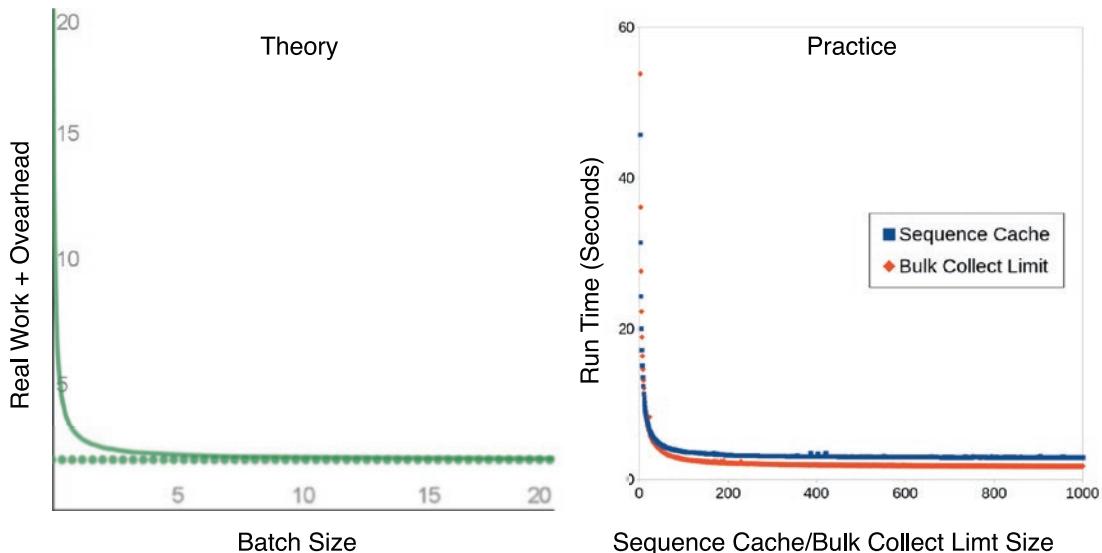


Figure 16-3. The effect of increased batch size on run time

For the theoretical chart on the left, the total run time is the dotted line of the “real work” plus the solid line of the overhead. Increasing the batch size rapidly reduces the overhead, but there is only so much overhead to reduce. No matter how much we increase the batch size, the total amount of work will never get to zero. Our processes are never purely overhead – there should always be a significant amount of real work to do.

The lines in the preceding graphs start near infinity and quickly plateau. As we increase the batch size, the overhead rapidly disappears. A batch size of 2 removes 50% of the overhead, a batch size of 10 removes 90%, and a batch size 100 removes 99%. A setting of 100 is already 99% optimized. The difference between a batch size of 100 and a batch size of one billion cannot theoretically be more than 1%.

The scripts I used to produce the real-world results can be found in the GitHub repository. There’s not enough space to list all the code here, but the code is worth briefly discussing. The scripts created two scenarios with a ridiculous amount of overhead: inserting rows where every value is a sequence and bulk collecting a small amount of data and doing nothing with the results. The test cases are almost completely full of useless overhead. If any test case was going to show a *meaningful* improvement by increasing batch size from 100 to 1000, this was it.

Table 16-1 describes three batching scenarios in detail. These rows are three completely unrelated tasks that can be perfectly explained with the same time complexity.

Table 16-1. *Different Tasks Where Performance Depends on Reducing Overhead*

Task	Real Work	Overhead	Configurable Parameter
Bulk collect	Selecting data	SQL-and-PL/SQL context switch	LIMIT
INSERT using sequence	Inserting data	Generating sequence numbers	CACHE SIZE
Application fetching rows	Selecting data	Network lag	Fetch size

There are many practical results we can derive from this theory. The default bulk collect size of 100 used by cursor FOR loops is good enough; there’s almost never any significant benefit to using a ridiculously high custom limit. The default sequence caching size of 20 is good enough; in extreme cases it might be worth increasing the

cache size slightly,² but not by a huge amount. Application prefetch is different for each application; we should aim for something close to 100. These rules apply to any overhead-reducing optimization.

If we have an extreme case or drill way down into the results, we can always find a minuscule difference with a larger batch size. But if we find ourselves in a situation where setting a huge batch size significantly helps, then we've already lost; we need to bring our algorithms to our data, not our data to our algorithms.

In practice, bringing algorithms to our data means we should put at least some of our logic in SQL, instead of loading billions of rows into a procedural language for a trivial amount of processing. If we foolishly load all the rows from a table into PL/SQL just to count them with `V_COUNT := V_COUNT+1`, then increasing the batch size will help. But the better solution would be to use `SELECT COUNT(*)` in a SQL statement. If we load a billion rows into PL/SQL and perform real work with those rows, the few seconds we save from a larger batch size will be irrelevant. There are always exceptions, like if we have to access lots of data over a database link with horrendous network lag, but we should not let those exceptions dictate our standards.

There are trade-offs between space and run time, but with a harmonic progression time complexity, we will quickly trade space for *no* run time. Developers waste a lot of effort debating and tweaking large numbers on the right side of the preceding graphs, but almost all the bang for our buck happens quickly on the left side of the graph. When we have a $1/N$ time complexity, the point of diminishing returns is reached very quickly. We should spend our time looking for opportunities to batch commands and reduce overhead, not worrying about the precise batch size.

0(1): Hashing and Other Operations

Constant-time access is ideal but often unrealistic. The operations that can work in constant time are mostly trivial, such as using a predicate like `ROWNUM = 1`. The constant-time function is simple, just a horizontal line, and is not worth showing on a graph. The most important Oracle operation that can run in constant time is hashing. Hashing is a core database operation and is worth discussing in detail.

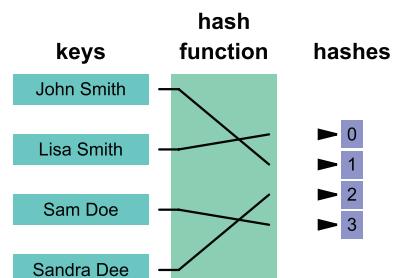
²Since 21c, Oracle may dynamically adjust sequence caching based on the sequence workload. Perhaps in the near future, we won't have to worry about sequence caching at all.

How Hashing Works

Hashing assigns a set of items into a set of hash buckets. Hash functions can assign items into a huge number of cryptographically random buckets, such as `SELECT STANDARD_HASH('some_string', 'SHA256') FROM DUAL`. Or hash functions can assign values into a small number of predefined buckets, such as `SELECT ORA_HASH('some_string', 4) FROM DUAL`. Hash functions can be designed with many different properties, and their design has important implications for how they are used. Figure 16-4 includes simple ways of describing hash functions.

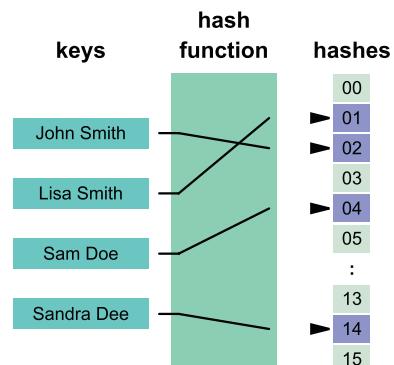
Minimal perfect hash

Every value maps to one unique hash, and there are no empty hash buckets



Perfect hash

Every value maps to one unique hash, but there are empty hash buckets



Typical Hash

Multiple values may map to the same hash, and there are empty hash buckets

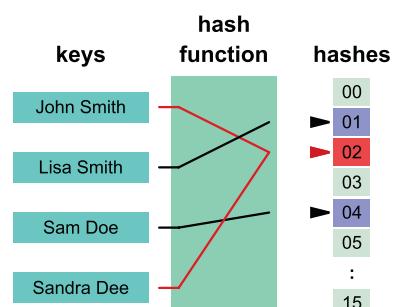


Figure 16-4. Description of different hash functions. Images were created by Jorge Stolfi and are in the public domain

There is a huge range of performance that depends on exactly how the hash works. With perfect hashing, access requires only a single read, and the operation is $O(1)$. With a broken hashing algorithm, where every value is mapped to the same hash, we just end up with a regular heap table stuck inside a hash bucket. In that worst case, we have to read the entire table to find a single value, which is $O(N)$.

When hashing, we need to be aware of space-time trade-offs. In practice we cannot achieve a minimal, perfect hash. Oracle's hash partitioning is minimal (there are no empty buckets), but far from perfect (there are many collisions); hash partitions don't provide instant access for one row, but they don't waste much space. Hash clusters can be perfect (no collisions) but are far from minimal (many empty buckets); hash clusters provide instant access for one row, but they waste a lot of space. Hash joins are somewhere in the middle. Those three types of hashing serve different purposes and are described in detail in the next sections.

Hash Partitioning

Hash partitioning splits a table of N rows into P partitions, based on the `ORA_HASH` of one or more columns. Hash partitioning will certainly not be a perfect hash, since each hash bucket is a segment meant to fit many rows. The number of hash partitions should be minimal, since we don't want to waste disk space by having extra segments.

The time to insert a row into a hash-partitioned table is still $O(1)$ – the `ORA_HASH` function can quickly determine which partition the row goes in. But the time to retrieve a row will be $O(N/P)$, or the number of rows divided by the number of partitions. That can be an important improvement for large data warehouse operations that read a large number of rows. But for reading a single row, that improvement is not nearly as good as what we can achieve with an $O(\log(N))$ B-tree index access. (The time complexity of B-tree indexes is described in a later section.) Don't try to replace indexes with partitions – they solve different problems.

We could theoretically build a hash-partitioned table with an amazing $O(1)$ read access by using a ridiculously large number of hashes. While that hash might be “perfect” in the sense that every row maps to a single segment, a ridiculous number of segments would waste a huge amount of space and would also cause problems with the data dictionary.

When building hash-partitioned tables, we must use the right columns for the partition key. If we use columns with a low cardinality or don't set the number of partitions to a power of two, the data will not be evenly distributed among the partitions. If all of the rows are stored in the same bucket (the same hash partition), then partitioning won't help us at all.

Hash partitioning is frequently overused in practice. To avoid abusing that feature, we need to understand the hashing trade-offs and choose a good partition key.

Hash Clusters

Hash clusters are designed to return a small number of rows, unlike hash partitions that are designed to return a large number of rows. An undocumented hash function tells Oracle exactly where to store and find each row – there's no need to walk an index tree and follow multiple pointers. In theory, hash clusters are even better than B-tree indexes for retrieving a small number of rows. In practice, hash clusters are rarely used.

The first problem with hash clusters is that we cannot add one to an existing table; we have to organize the table from the beginning to use a hash cluster. To get $O(1)$ read time on hash clusters, we need to create a near-perfect hash. But we have to worry about that space-time trade-off. There is no practical way to get perfect hashes without also creating a huge number of unused hash buckets. We can have a hash cluster with good performance or a hash cluster that uses a minimal amount of space; we can't have both.

When we create a hash cluster, we can specify the number of buckets with the `HASHKEYS` clause. In my experience, getting $O(1)$ read time requires so many hash buckets that the table size will triple.

Unfortunately, even after all that effort to get $O(1)$ access time, hash clusters still end up being slower than indexes. I can't explain why hash clusters are slower; this is a place where our theory breaks down in real-world applications.

Oracle measures the number of reads performed by a SQL statement using a statistic called “consistent gets.” It's possible to create test cases where hash lookups require only one consistent get, while index lookups on the same table require four consistent gets.³ But the index is still faster. For these operations, the $O(\log(N))$ of an index is less than the $O(1)$ of a hash cluster.

³ See this Stack Overflow answer, where I try and fail to create a useful constant-time index using hash clusters: <https://stackoverflow.com/questions/32071259/constant-time-index-for-string-column-on-oracle-database>

When comparing small numbers, such as 1 vs. 4, the constants become more important than the Big O analysis. Hash clusters are rarely used, and Oracle surely invests more time optimizing indexes than clusters. Perhaps those optimizations compensate for the small difference between the run-time complexities.

Algorithm analysis helps us drill down to the problem, but in this case the real performance difference is hidden by constants in closed source software. Perhaps this is an opportunity for a custom index type. Other databases have hash indexes that don't require reorganizing the entire table. Maybe someday Oracle will add that feature and make hash access work better. For now, we should ignore hash clusters.

Hash Joins

Hashing is a great way to join two tables when there's a large percentage of rows that match between them. A hash join starts by creating a temporary hash table based on the join column values from the first table. Next, Oracle uses the join column values from the second table and probes the hash table to see if there is a match.

The precise performance of hash joins is discussed later, but the main point to understand is that the first step of creating a temporary hash table requires a significant amount of time and space. That extra work can be a good investment because each probe in the second step can theoretically run in $O(1)$ time. But like with hash partitioning and hash clusters, the hash table is certainly not minimal and perfect. There will be some wasted space, and the probe run time ends up being something worse than $O(1)$. However, unlike hash clusters, the probe run time is significantly better than the $O(\log(N))$ we would get from a single index lookup. Determining when that up-front investment of building a hash table is worth the cost is one of the most important decisions the optimizer makes.

It's important for us to understand that hash joins can only be used for equality conditions. The way input values are mapped to fixed-length hash values preserves some equality properties, but hashing does not preserve any other relationships between the inputs. For example, A may be smaller than B, but that relationship is not necessarily true of their hash values.

Hash joins are so useful that it is often worth going out of our way to enable them. We can enable hash joins by rewriting simple non-equality conditions into weird equality conditions. For example, `COLUMN1 = COLUMN2 OR (COLUMN1 IS NULL AND COLUMN2 IS NULL)` is a logical way of telling Oracle, "Either these columns are equal or both of them

are NULL." We may be able to significantly improve the performance of joining two large tables by rewriting the condition to `NVL(COLUMN1, 'fake value') = NVL(COLUMN2, 'fake value')`. Writing cryptic expressions is not ideal and may cause problems from bad cardinality estimates, but it's often worth the trouble if it enables faster join operations.

Other

Constant-time operations show up frequently in Oracle, like in any system. For example, inserting a row into a table, creating objects, and altering objects often take the same amount of time regardless of the input size.

On the other hand, all those operations also have nonconstant time versions. Inserting a row into a table takes a non-trivial amount of time if there are indexes that need to be maintained. Creating objects like indexes can take a huge amount of time to sort the data. Even altering tables may or may not take constant time, depending on the `ALTER` command. Adding a constraint usually requires validation against the data, which depends on the number of rows; but adding a default value can be done purely in metadata and can finish almost instantly.

We can't categorize operations based purely on their command type. To estimate the run time of any command, we always need to think about the algorithms, data structures, and size of the data.

O($\log(N)$): Index Access

$O(\log(N))$ is the worst-case run time for index access. Index reads are another core database operation. Indexes were described in detail in Chapter 9, but a brief summary of this important concept is included in the following paragraph.

When we search a binary tree, each step can eliminate half the rows of the index. Doubling the number of rows grows exponentially; conversely, halving the number of rows shrinks logarithmically. Index access on a simple binary tree is $O(\log_2(N))$. Oracle B-tree indexes store much more than one value per branch, so they have a worst-case access time of $O(\log(N))$. Figure 16-5 shows an example of a binary tree search.

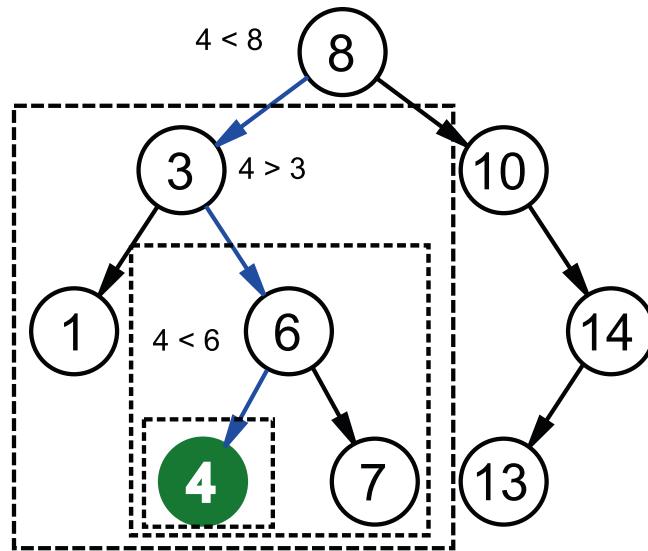


Figure 16-5. Binary search algorithm. Image was created by Chris Martin and is in the public domain

We know how indexes work, and we know that indexes are great for performance, but comparing the algorithms helps us understand precisely how awesome indexes can be. Figure 16-6 compares the fast $O(\log(N))$ of an index read with the slow $O(N)$ of a full table scan. This visualization is a powerful way of thinking about index performance.

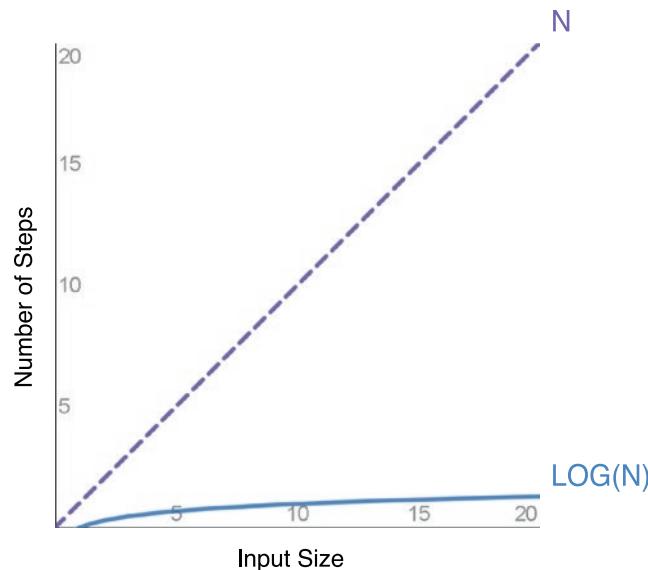


Figure 16-6. Compare $O(\log(N))$ index read vs. $O(N)$ full table scan

Most performance tests only compare results at a single point along a range of input sizes. Thinking about the lines and curves in the preceding visualizations helps us understand how our system's performance will change as the data grows. The amount of work required to read one row from an index rarely increases, no matter how large the table grows. In practice, our B-tree indexes will rarely even grow to a height of five levels, which means any indexed value is just a few reads away. Contrast that gradual run-time growth with the steeper growth of a full table scan, where every new row adds new work.

So far, we've mostly discussed small operations – one hash lookup, one index lookup, or one full table scan. The performance comparisons will soon become trickier as we start iterating the operations.

1/((1-P)+P/N): Amdahl's Law

For optimal parallel processing in a data warehouse, we have to worry about *all* operations. It is not enough to only focus on the biggest tasks. If our database has N cores and we want to run large jobs N times faster, we need to parallelize *everything*.

Amdahl's law is the mathematical version of the preceding paragraph. Amdahl's law is not a worst-case run-time complexity, but the function is important for understanding performance and is worth discussing here. The law can be expressed as the following equation in Figure 16-7.

$$\text{TotalSpeedup} = \frac{1}{(1 - \text{ParallelPortion}) + \frac{\text{ParallelPortion}}{\text{ParallelSpeedup}}}$$

Figure 16-7. Amdahl's law as an equation

We don't need to remember the equation, and it's only listed for completeness. But we do need to remember the lesson of Amdahl's law, which can be learned from the graph in Figure 16-8.

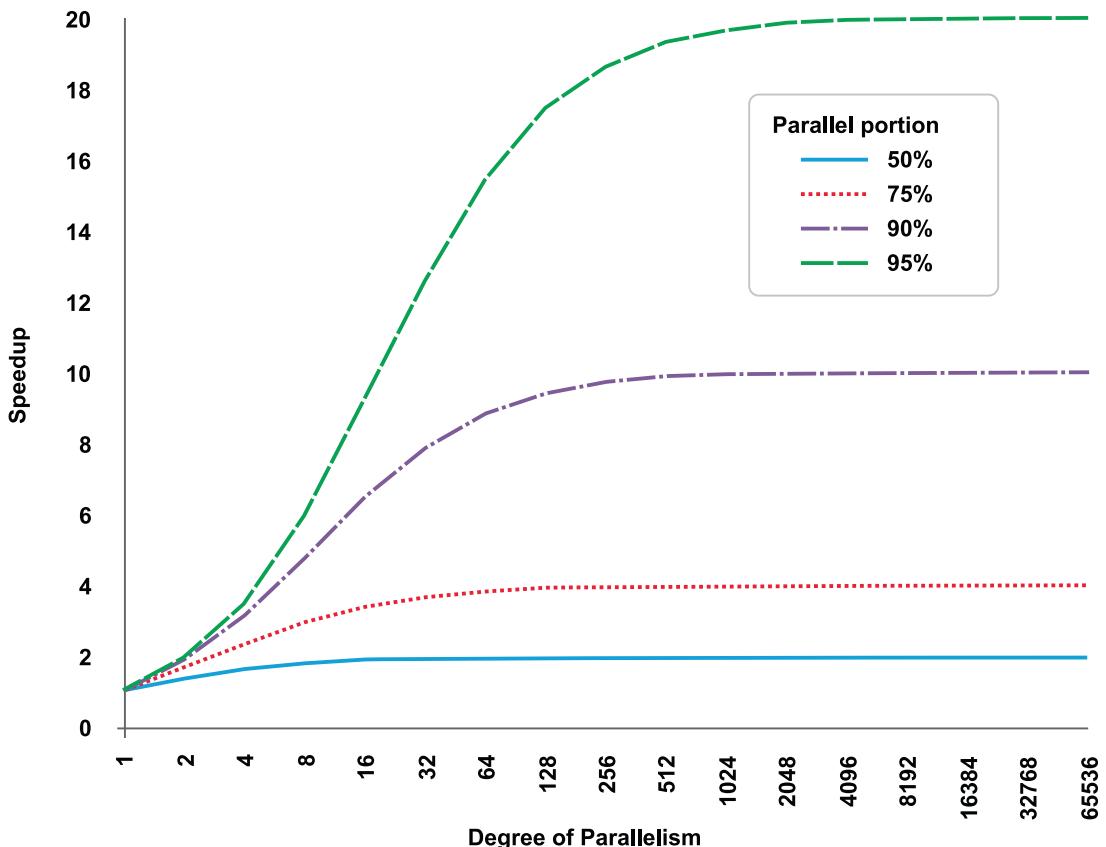


Figure 16-8. Amdahl's law as a graph. Based on "Amdahl's Law" by Daniels220, licensed under CC BY-SA

Notice that the preceding graph uses a logarithmic X axis and the numbers on the Y axis are comparatively small. The implications of this graph are depressing; even the tiniest amount of serialization will dash our dreams of running N times faster.

For example, let's say we've got a large data load and we've parallelized 95% of the process. Our machine has 128 cores, and (miraculously) the parallel portion runs 128 times faster. Yet the overall process is only 17.4 times faster. If we foolishly throw hardware at the problem and increase the cores from 128 to 256, the process will only run 18.6 times faster. These are disappointing diminishing returns.

Our time would be much better spent increasing the parallel portion from 95% to 96%, which would increase the performance from 17.4 times faster to 21 times faster. These surprising performance gains are why we need to spend so much effort finding exactly where our processes spend their time. We all like to tune with our gut feelings,

but our intuition can't tell the difference between 95% and 96%. That level of precision requires advanced tools and a deep analysis of the process activity.

For data warehouse operations, we cannot only worry about the longest-running tasks. It's not enough to parallelize only the `INSERT` or `CREATE TABLE` statement. To get the maximum speedup, we need to put a lot of effort into all the other data loading operations. We need to pay attention to operations like rebuilding indexes, re-enabling constraints, gathering statistics, etc. Luckily, Oracle provides ways to run all of those operations in parallel. Previous chapters included examples of rebuilding indexes and re-enabling constraints in parallel, and Chapter 18 shows how to easily gather statistics in parallel.

To fully optimize large data warehouse operations, it can help to create a graph of the system activity, with tools like Oracle Enterprise Manager or `DBMS_SQLTUNE.REPORT_SQL_MONITOR`. As discussed in Chapter 12, optimizing data warehouse operations requires worrying about even the tiniest amount of system inactivity.

O(N): Full Table Scans, Other Operations

$O(N)$ is a simple, linear performance growth. We find this run-time complexity all over the place: full table scans, fast full index scans (reading all the index leaves instead of traversing the tree), SQL parsing, basic compression, writing or changing data, etc. There's not much to see here, but things get interesting in the next section when we start comparing more functions.

Most Oracle operations fall into the $O(N)$ category. In practice, we spend most of our tuning time hoping for linear improvements. For example, direct-path writes can improve performance by reducing the amount of REDO and UNDO data. That's only a linear improvement, but it's a significant one.

O(N*LOG(N)): Full Table Scan vs. Index Access, Sorting, Joining, Global vs. Local Index, Gathering Statistics

$O(N*LOG(N))$ is the worst-case run time for realistic sorting algorithms. This run-time complexity shows up in many unexpected places, like constraint validation. Previous sections discussed data structures and algorithms that mostly focused on finding a single

value, and now it's time to start iterating those algorithms to find multiple values. For finding a single value, an $O(\log(N))$ index access is obviously faster than an $O(N)$ full table scan, but what happens when we're looking up more than a single row?

Figure 16-9 compares the functions discussed in this section: N^2 , variations of $N \cdot \log(N)$, and N .

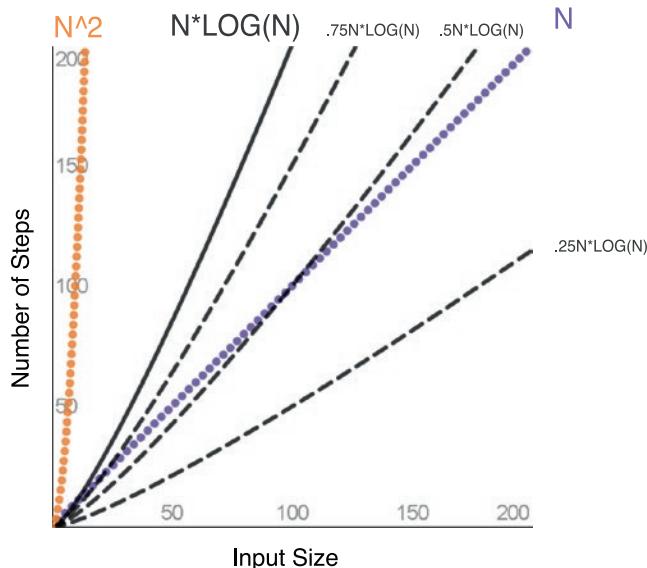


Figure 16-9. Comparing N^2 , $N \cdot \log(N)$, and N

The preceding lines and curves can be used to understand full table scans vs. index access, sorting, joining, global vs. local indexes, gathering statistics, and many other operations.

Full Table Scan vs. Index Access

Indexes are awesome for looking up a single row in a table – it's hard to beat $O(\log(N))$ read time. But other than primary key or unique key indexes, most index access will retrieve more than one row. To retrieve multiple rows, Oracle must repeat $\log(N)$ multiple times.

If an index scan is used to look up every row in the table, Oracle has to walk the tree N times, which leads to a $N \cdot \log(N)$ run time. That run time is clearly much worse than just reading the entire table one row at a time. The performance difference is obvious

if you look at Figure 16-9 and compare the slower, solid line of $N \cdot \log(N)$ against the faster, dotted line of N . There are many times when a full table scan is faster than index access.

The performance difference is more complicated when Oracle reads a percentage of rows other than 1% or 100%. Notice the dashed lines in Figure 16-9; they represent repeating the $\log(N)$ access for 25%, 50%, or 75% of the rows in the table. As the size of the table grows, each curve will eventually overtake the linear line for N . We cannot simply say that one algorithm is faster than another. The fastest algorithm depends on the size of the table and the percentage of rows accessed. An index might be a good idea today for reading 25% of a table, but a bad idea tomorrow if the table has grown.

There are other factors that can significantly change the balance. Oracle can use multi-block reads for full table scans, as opposed to single-block reads for index access. Using multi-block reads, Oracle can read several blocks from a table in the same time it takes to read one block from an index. And if the index clustering factor is high (if the index is not ordered by the value being searched for), an index lookup on a small number of rows may end up reading all of the blocks of the table anyway.

The theory tells us the *shape* of the lines, but only practice can tell us the actual values. I can't give you an exact number for your system, table, and workload. But there is a number – a point where a full table scan becomes cheaper than an index. Finding that number is not trivial, and it should not surprise us that Oracle doesn't always make the right decision.

When Oracle fails to make the right choice, we shouldn't throw out the entire decision-making process by using an index hint. Instead, we should look for ways to provide more accurate information, to help Oracle make better choices. Helping Oracle is usually done by gathering optimizer statistics.

Sorting

$O(N \cdot \log(N))$ is the worst-case run time for popular sorting. We always hope our processes will scale linearly, but that is often not the case. In practice, we have to deal with algorithms that get slower faster than we anticipate. Sorting is a central part of any database and affects ORDER BY, analytic functions, joining, grouping, finding distinct values, set operations, etc. We need to learn how to deal with these slow algorithms and avoid them when possible.

For planning, we need to be aware of how performance will change over time. If sorting one million rows takes 1 second today, we cannot assume that sorting two million rows will take 2 seconds tomorrow.

We also need to be aware of how the space requirements for sorting will change with the input size. Luckily, the amount of space required grows linearly. If the number of rows doubles, the amount of PGA memory or temporary tablespace required to sort will also double.

There are times when the sorting is already done for us. A B-tree index has already been sorted – the work was done during the original INSERT or UPDATE. Adding a row to a table is $O(1)$ – the process only requires adding a row to the end of a dumb heap. Adding a row to an index is $O(\log(N))$ – the process needs to walk the tree to find where to add or update the value.

Oracle can use an index full scan to read from the index, in order, without having to do any sorting. The work has already been done; Oracle just needs to read the index from left to right or right to left.

Oracle can also use a min/max read to quickly find the minimum or maximum value. The minimum or maximum value in a B-tree will be either all the way on the left or all the way on the right. Once again, the data is already sorted, so finding the top or bottom result is a trivial $O(\log(N))$ operation.

Oddly, there's a missing feature where Oracle can't find *both* the min and max using a simple min/max read.⁴ But the following code shows a simple workaround to this problem: break the problem into two separate queries, and then combine the results. Writing an extra subquery is annoying, but it's a small price to pay for a huge improvement in run-time complexity – $O(2*\log(N))$ is much better than $O(N)$. When we understand the algorithms and data structures used by Oracle, we know what to expect and when to look for a faster workaround:

```
--Create a table and query for min and max values.
create table min_max(a number primary key);

--Full table scan or index fast full scan - O(N).
select min(a), max(a) from min_max;
```

⁴ See my Stack Overflow answer here for more details: <https://stackoverflow.com/q/43131204/409172>

--Two min/max index accesses - $O(2 \cdot \log(N))$.

select

```
(select min(a) from min_max) min,  
(select max(a) from min_max) max  
from dual;
```

The set operations INTERSECT, MINUS/EXCEPT, and UNION require sorting. We should use UNION ALL when possible because it is the only set operation that does not need to sort the results.

Sorting and joining seem to go together, but in practice they are a bad combination. The next section discusses why we don't want to use sorting for joins.

Joining

Hopefully you remember the Venn diagrams and join diagrams from Chapter 1, which explained how joins logically work. Unfortunately, understanding how joins physically work is more complicated. Figure 16-10 visualizes the main join algorithms and their run-time complexity. The diagram visualizes joins as the process of matching rows between two unordered lists. Each join algorithm is also described in a separate paragraph after the diagram. You may need to flip back and forth a few times to understand the algorithms.

Join Algorithms and Time Complexity

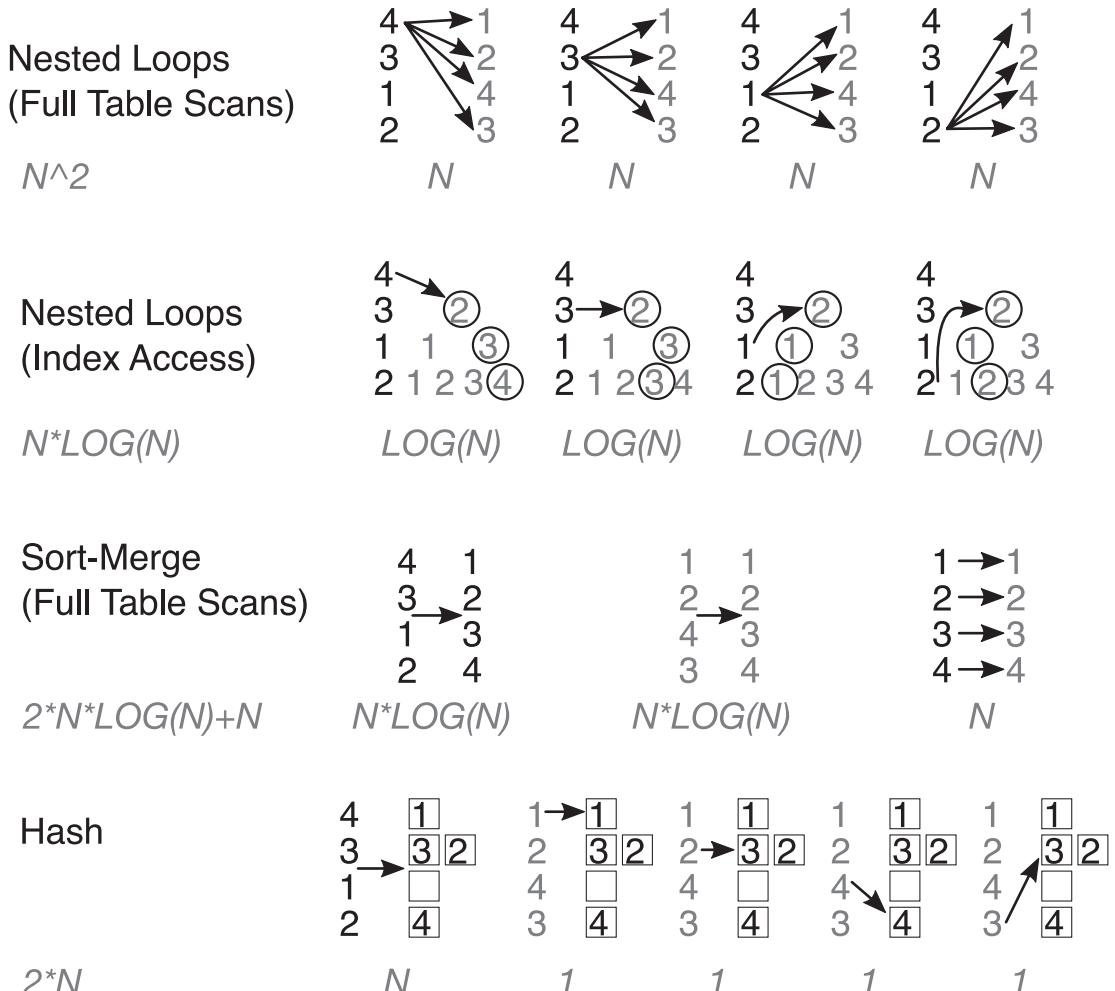


Figure 16-10. Visualization of join algorithms and time complexity

A **nested loop with full table scans (N^2)** is conceptually simple. Start with the first table and compare each row in that table with every row in the second table. We can think of this algorithm as two nested FOR loops. Each of the loops runs N time, so the run-time complexity is a horribly slow $O(N^2)$. Recall from Figure 16-9 that the line for N^2 looks almost completely vertical. This join algorithm should only be used when the tables are trivially small, the optimizer statistics are bad, or important indexes are missing.

A **nested loop with index access ($N * \log(N)$)** is often a faster choice for joins. Instead of searching an entire table for each row, we can search an index. Reducing $O(N^2)$ to $O(N * \log(N))$ is a huge improvement.

But to get the most out of this algorithm, it helps to be more specific with our variables. Instead of using N to mean the number of rows in either table, let's say the first table that we read every row has A rows and the second table that we read using an index has B rows. If we think about $O(A * \log(B))$, the number of steps mostly depends on A instead of B . The smaller table should be first, and the larger table should be second. For example, if one table has a hundred rows and another table has a million rows, $100 * \log(1,000,000) = 600$ is much smaller than $1,000,000 * \log(100) = 2,000,000$. It's not enough that Oracle chooses the right algorithm – it must also know how to use the tables in the algorithm.

If both tables are large, this join algorithm is not efficient. Nested loops work best when the number of rows is small for one of the tables and there is an index access for the other table.

A **sort-merge join with full table scans ($2 * N * \log(N) + N$)** operates by first sorting both tables, which is expensive, and then matching the sorted rows, which is cheap. In practice we don't see sort-merge joins very often – nested loops with index access are a better choice for small joins, and hash join is a better choice for large joins. But if there are no relevant indexes (ruling out nested loops) and there is no equality condition (ruling out a hash join), then sort-merge is the best remaining option. And if one of the tables is already presorted by an index, then half of the sorting work is already done.

A **hash join ($2 * N$)** has two phases. The smaller table is read and built into a hash table, ideally in memory. Then the larger table is scanned, and the hash table is probed for each row in the larger table. With a perfect hash, writing and reading from a hash table only takes one step, and the run time is only $O(2 * N)$. But in practice there will be collisions, and multiple rows will be stored in the same hash bucket, so the time to perform hash joins does not grow linearly; hash joins are somewhere between $O(2 * N)$ and $O(N * \log(N))$.⁵

Hash joins have a great run-time complexity, but that doesn't mean we always want to use them. Hash joins read all of the rows of both tables, even if we are only going to match a few rows. Hash joins require a lot of memory or temporary tablespace – roughly equal to the size of the smaller input table. A slow algorithm in memory might be better

⁵ See my answer here for tests comparing sort-merge with hash: <https://stackoverflow.com/a/8548454/409172>. When available, hash joins are always faster.

than a fast algorithm on disk. And hash joins are not always available, because they only work with equality conditions.

There are many variations of the preceding join algorithms. Cross joins, also known as Cartesian products, are similar to nested loops with full table scans. Parallelism and partitioning can always complicate the algorithms. Hash joins can use bloom filters to eliminate values without fully comparing all of them. Joining is the most important operation in the database, and there's not enough space here to describe all of the features for joining tables.

Join performance is all about Oracle correctly estimating the size of tables. The table size determines which algorithms are used and how the tables are used in the algorithms. For example, hashing and sorting are bad ideas if one of the tables returns no rows – there's no need for all that prep work when the intermediate data structures won't be used. And a nested loop is a horrible idea for two huge tables that have a lot of matching rows. Bad execution plans happen when Oracle thinks a small table is large or a large table is small. How that mistake happens, and how to fix it, is discussed in Chapters 17 and 18.

Global vs. Local Index

The advantages and disadvantages of partitioning are different for tables and indexes. Partitioning a table is often a good choice, since there are many potential performance improvements and few costs. Indexes can be either global, one index for the whole table, or local, one index per partition. Compared with table partitioning, index partitioning benefits are smaller, and the costs are greater.

Reading from a single table partition, instead of a full table scan, significantly decreases the run-time complexity from $O(N)$ to $O(N/P)$, where P is the number of partitions. But reading from a local index, instead of a global index, only changes the run-time complexity from $O(\log(N))$ to $O(\log(N/P))$. That index access improvement is barely noticeable.

The cost of reading an entire partitioned table, without partition pruning, is still a normal $O(N)$. But reading from a local index without partition pruning is much more expensive than a global index read. Reading from one large index is $O(\log(N))$. Reading from many small indexes is $O(P * \log(N/P))$, a significant increase. Walking one big tree is much faster than walking many small trees. We need to think carefully when we partition, and we should not expect table and index partitioning to work the same way.

Gathering Optimizer Statistics

Gathering optimizer statistics is an essential task for achieving good SQL performance. Statistics are used to estimate cardinality, the number of rows returned by an operation. Cardinality is synonymous with “input size,” the X axis on most of the graphs in this chapter. Cardinality is vital information for making execution plan decisions, such as knowing when to use a nested loop or a hash join.

To get good execution plans, we need to gather statistics periodically and after any significant data changes. But gathering statistics itself can be slow. Understanding the different algorithms and data structures for gathering statistics can help us avoid performance problems.

Finding the cardinality isn’t just about counting the number of rows in each table. Oracle also needs to measure the distinctness of columns and specific column values. For example, an equality condition on a primary key column will never return more than one row, so it is a good candidate for index access. But an equality condition on a repetitive status column is more complicated; some statuses may be rare and benefit from index access; other statuses may be common and work best with a full table scan. Optimizer statistics gathering needs to generate information that can be used to estimate the cardinality for many complicated scenarios.

Counting distinct items is similar to sorting, which is a slow operation. To make things even worse, Oracle needs statistics for all the columns in a table, which may require multiple passes. A naïve algorithm to count distinct values would first sort those values; it’s easy to measure distinctness if the values are in order. But that naïve approach would take $O(N \cdot \log(N))$ time. A better algorithm would use hashing, and that’s exactly what recent versions of Oracle can do with the HASH GROUP BY and HASH UNIQUE operations. As with joining, the time to hash is somewhere between $O(N)$ and $O(N \cdot \log(N))$.

Luckily, when gathering optimizer statistics, we can trade accuracy for time. The default Oracle statistics gathering algorithm performs an *approximate* count with a single pass of the table. The single-pass algorithm runs in $O(N)$ time and generates numbers that are accurate but not perfect.

Developers rightfully worry about imperfect values. A single wrong bit can break everything. But in this case an approximate value is good enough. The optimizer doesn’t need to know the *exact* value. Oracle only needs to know if the algorithm needs to be optimized for “large” or “small” queries.

The following example shows how close the approximation is, using the function APPROX_COUNT_DISTINCT, which uses the same algorithm as statistics gathering:

```
--Compare APPROX_COUNT_DISTINCT with a regular COUNT.
select
    approx_count_distinct(launch_date) approx_distinct,
    count(distinct launch_date) exact_distinct
from launch;
```

APPROX_DISTINCT	EXACT_DISTINCT
61745	60401

The fast approximation algorithm⁶ only works if we read the entire table. If we try to sample a small part of the table, the approximation algorithm no longer applies, and Oracle has to use a different approach. We may think we're clever setting a small estimate percentage, such as DBMS_STATS.GATHER_TABLE_STATS(..., ESTIMATE_PERCENT => 50). But in this case, a slow algorithm reading 50% of the table is slower and less accurate than the fast algorithm reading 100% of the table. We should rarely, if ever, change the ESTIMATE_PERCENT parameter.

These distinct counting tricks can also apply to partition statistics, using a feature called incremental statistics. Normally, partitioned tables require reading the table data twice to gather statistics: one pass for each partition and another pass for the entire table. The double read may seem excessive at first, but consider that we cannot simply add distinct counts together.

But incremental statistics uses an approximation algorithm that does enable adding distinct counts. Incremental statistics creates small data structures called synopses, which contain information about the distinctness within a partition. After gathering statistics for each partition, the global statistics can be inferred by merging those small synopses. This algorithm improves statistics gathering from $O(2^*N)$ to $O(N)$. That decrease may not sound impressive, but remember that partitioned tables are often huge. Saving an extra full table scan on our largest tables can be a big deal.

This algorithm analysis is starting to get recursively ridiculous. We're discussing algorithms that help Oracle determine which algorithms to use. And the problem goes deeper – in rare cases, statistics gathering is slow because the optimizer chooses a bad

⁶Oracle uses an algorithm called HyperLogLog for distinct count approximations.

plan for the statistics gathering query. For those rare cases, we may need to prime the pump; we can use DBMS_STATS.SET_TABLE_STATS to create initial, fake statistics, to help us gather the real statistics.

Many organizations consider statistics gathering a boring maintenance task that is best handled by the DBAs. But that metadata is at the heart of Oracle performance, and it should be understood by anyone who is responsible for our database program's performance.

O(N^2): Cross Joins, Nested Loops, Other Operations

$O(N^2)$ algorithms are getting into the ridiculously slow territory. Unless we have special conditions, such as an input size close to 0, we should avoid these run times at all costs. This run-time complexity happens from cross joins, nested FOR loops, nested loop joins with full table scans, and some other operations. As a quick reminder of their poor performance, Figure 16-11 shows $N!$ and N^2 .

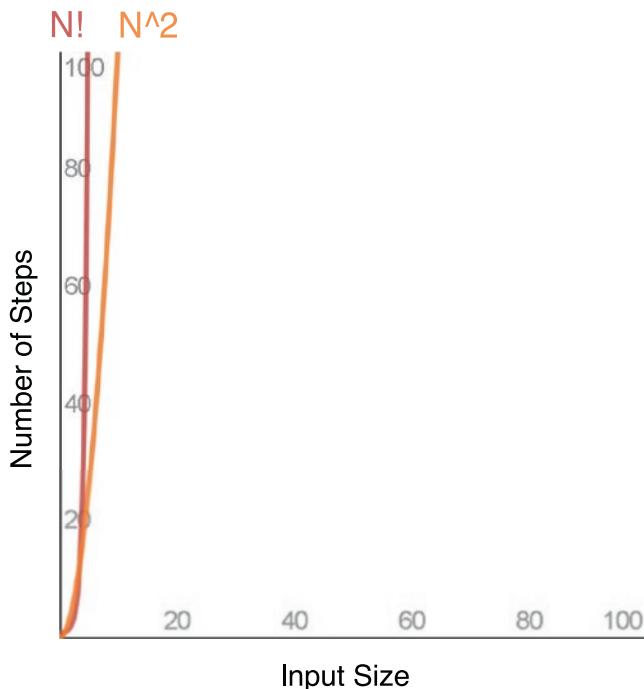


Figure 16-11. Compare $N!$ with N^2

Cross joins are not always bad. Sometimes it is necessary to mindlessly combine every row in one table with every row in another table. And cross joins can be fast if the number of rows is close to zero. But cross joins that happen because of incorrectly joined tables, or because of poor cardinality estimates, are horrible for performance. Accidental cross joins are so bad that they can effectively bring down an entire database; Oracle may need an unlimited amount of temporary tablespace to build the result set, depriving other processes of space for sorting or hashing.

FOR loops inside FOR loops are the most common way to generate $O(N^2)$ run times in procedural code. If we add *another* FOR loop, the run-time complexity becomes $O(N^3)$, then $O(N^4)$, etc. Nested FOR loops can easily happen in PL/SQL, and they are all too common when using explicit cursors. Avoiding that horrible performance is an important reason we should always try to replace nested cursors with a single SQL statement.

Most developers write procedural code as nested FOR loops because loops are the easiest way to think about joins. But Oracle cannot change the algorithms used by imperative programs – Oracle is stuck doing exactly what we ask for. The advantage of switching to declarative code is that we don't have to worry about these algorithms as much. When we use SQL, we let Oracle decide which join algorithms to use.

It is possible that after switching to declarative SQL the optimizer will still choose an $O(N^2)$ algorithm. As we saw in Figure 16-10 earlier in the chapter, a nested loop join with two full table scans is a terrible way to join tables. But that worst case should only happen if we have missing statistics, missing indexes, or weird join conditions.

$O(N^2)$ shows up in other unexpected places. In extreme cases, SQL parse time grows exponentially. For example, the parse time becomes ridiculously bad if we combine thousands of subqueries with UNION ALL. (The code measuring parse time is too large to include here, but it can be found in the repository.) Writing large SQL statements is a good thing, but writing Brobdingnagian SQL statements is not.

The MODEL clause brings procedural logic to SQL statements. MODEL is a neat trick and is explained briefly in Chapter 19. That clause gives us the ability to turn our data into a spreadsheet and apply FOR loops to our rows and columns. But just like with PL/SQL, we can easily find ourselves in $O(N^2)$ performance territory.

A high run-time complexity isn't always avoidable or necessarily bad. But we should think carefully when we see cross joins or nested operations.

O(N!): Join Order

$O(N!)$ is as bad as Oracle's algorithms get, but luckily, it's also rare. As discussed in Chapter 6, tables have to be joined in a specific order, and there are many ways to order a set of tables. The problem of ordering tables must be solved by the optimizer when building execution plans and in our minds when trying to understand queries.

The optimizer is capable of handling a huge number of tables without serious problems. But there are always unexpected problems when we push the system to the limits. For example, if we nest over a dozen common table expressions, the parse time appears to grow like $O(N!)$. (See the GitHub repository for a code demonstration.) In practice, this problem will only happen because of a rare bug or because we're doing something we shouldn't.

Our minds can only hold a small, ordered list in short-term memory. If we build our SQL statements with giant comma-separated lists, we'll never be able to understand them. Building small inline views and combining them with the ANSI join syntax will vastly simplify our SQL. There's no precise equation for SQL complexity, but we can think of the complexity comparison like this: $(1/2N)! + (1/2N)! \ll N!$

For performance we want to batch things together, to reduce overhead. But to make our code readable, we want to split everything up into manageable pieces. That is precisely what SQL excels at; we can logically divide our queries into inline views to make them understandable, and then the optimizer can efficiently put the code back together.

O(∞): The Optimizer

The Oracle optimizer builds SQL execution plans. Building optimal execution plans is impossible, so we could say this task is $O(\infty)$. That run-time complexity sounds wrong at first; Oracle obviously builds at least *some* optimal execution plans. But it's helpful to think of the optimizer as doing an impossible task.

Building the best execution plan means the optimizer must compare algorithms and data structures and determine which one runs fastest. But it is literally impossible to determine if a generic program will even *finish*, much less how long it will run. Determining if a program will finish is called the halting problem. Before computers were even invented, Alan Turing proved that it is impossible for computers to solve the

halting problem. Luckily, this is a problem that's theoretically impossible to solve in all cases, but it is practically feasible to solve in almost all cases.

To add another wrinkle to the problem, Oracle has to generate the execution plan incredibly fast. A better name for the optimizer would be the satisficer. Satisficing is solving an optimization problem while also taking into account the time necessary to solve the problem.

It's important we understand how difficult the optimizer's job is. We shouldn't be surprised when Oracle occasionally generates a bad execution plan. The optimizer isn't as bad as we think – it's merely trying to solve an unsolvable problem. When the optimizer is having trouble, we shouldn't abandon it; we should try to work with it.

When a bad execution plan is generated, our first instinct shouldn't be, "How can I work around this bad plan?" Our first instinct should be, "How can I provide better information so Oracle can make better decisions?" That information is almost always in the form of optimizer statistics, which are described in Chapters 17 and 18.

We need to resist the urge to quickly use hints and change system parameters. Think of the optimizer as a complex forecasting system. We should never change a system parameter such as `OPTIMIZER_INDEX_COST_ADJ` just because it helps with one query. That would be like adding 10 degrees to every weather forecast because the meteorologist was once wrong by 10 degrees.

Summary

Many performance problems fall into a small number of run-time complexities. Knowing which functions represent our problems can help us understand why Oracle is behaving a certain way and how to find a solution. Practical algorithm analysis is simply matching our problems with predefined classes of problems. We may not use this approach often, but without it we'll never be able to truly understand database performance.

Most of our performance problems are related to implementation details and those pesky constants we've been conveniently ignoring. The next chapter looks at a more traditional list of Oracle performance concepts.

**You are viewing a courtesy copy of chapter 16
from Jon Heller's book, *Pro Oracle SQL*
*Development***

Would you like a copy of the complete book?

Purchase now from Amazon:

**[https://www.amazon.com/Pro-Oracle-SQL-
Development-Practices/dp/1484288661](https://www.amazon.com/Pro-Oracle-SQL-Development-Practices/dp/1484288661)**

Or from Apress.com:

**[https://link.springer.com/book/10.1007/978-1-
4842-8867-2](https://link.springer.com/book/10.1007/978-1-4842-8867-2)**

Thank you for your support of the author and his work