Oracle SQL & PL/SQL Optimization for Developers Documentation

Release 2.1.1

Ian Hellström

Dec 29, 2017

Contents

CHAPTER 1

Introduction

SQL is a peculiar language. It is one of only a handful of fourth-generation programming languages (4GL) in general use today, it seems deceptively simple, and more often than not you have many quite disparate options at your disposal to get the results you want, but only few of the alternatives perform well in production environments. The simplicity of the language veils decades of research and development, and although the syntax feels (almost) immediately familiar, perhaps even natural, it is a language that you have to wrap your head around. People coming from imperative languages often think in terms of consecutive instructions: relational databases operate on sets not single entities. How the SQL optimizer decides to execute the query may not coincide with what you think it will (or ought to) do.

To the untrained eye a database developer can seem like a sorcerer, and it is often said that query tuning through interpreting execution plans is more art than science. This could not be further from the truth: a thorough understanding of the inner workings of any database is essential to squeeze out every last millisecond of performance. The problem is: the information is scattered all over the place, so finding exactly what you need when you need it can be a daunting task.

1.1 Why This Guide?

While it's easy to write bad code in any programming language, SQL — and to some extent PL/SQL too — is particularly susceptible. The reason is simple: because of its natural-looking syntax and the fact that a lot of technical 'stuff' goes on behind the scenes, some of which is not always obvious to all but seasoned developers and DBAs, people often forget that they are still dealing with a programming language and that SQL is not a silver bullet. Just because it looks easy does not mean that it is. We don't want to step on anyone's toes but frequently production SQL code (e.g. reports) is created not by developers but business users who often lack the know-how to create quality code. It's not necessarily their jobs to come up with efficient queries but it is not uncommon to hear their gripes afterwards, once they discover that a report takes 'too long' because the database is 'too slow'. The fact of the matter is that they have run into one of many traps, such as non-SARGable predicates, bad (or no) use of indexes, unnecessarily complicated (nested) subqueries that not even their creator can understand after a short lunch break but somehow magically deliver the correct, or rather desired, results. Other programming languages, especially the more common third-generation ones, do not have that problem: applications are mostly developed by professional developers who (should) know what they're doing.

There are many excellent references on the topic of SQL and PL/SQL optimization, most notably Oracle's own extensive documentation, Tom Kyte's Ask Tom Q&A website, entries by Burleson Consulting, Tim Hall's Oracle Base

pages, Steven Feuerstein's PL/SQL Obsession, Oracle Developer by Adrian Billington, books, and a wealth of blogs (e.g. by Oracle ACEs).

'So why this guide?' we hear you ask. Two reasons really:

1. It's a long and at times arduous journey to master Oracle databases, and it's one that never ends: Oracle continues to develop its flagship database product and it is doubtful that anyone knows everything about its internals. We hope that other developers will benefit from the experiences (and rare insights) chronicled here. These page are personal field notes too, so they evolve as we discover new avenues of investigation. The bulk of what is written here can of course be found elsewhere, and we have included references where appropriate or rather where we remembered the respective sources. Since it it is rarely straightforward to gather the information relevant to your particular use case, sift through it in a timely manner, and understand it well enough to use it, we hope to assist in that task. We have no intention or illusion of replacing the aforementioned resources or the coder's home base: Stack Overflow.
2. A lot of the documentation on tuning queries is geared towards DBAs. Often a developer needs to work closely with the DBA to discover what the source of some performance issue is, and quite frequently the DBA can help: if the database is tuned improperly, the best query in the world can be excruciatingly slow. Most of the time, though, the developer can tweak his or her queries without calling in the cavalry. These pages describe what a SQL and/or PL/SQL developer can do without the DBA. Furthermore, code optimization is often seen as the last step in the development process, and it is sometimes only then looked at seriously when the performance is abominable. We believe that much can be done before it's (almost) too late.

Note: We do not advocate premature optimization at all; code does not have to be as fast as technically possible, just as fast as necessary. Instead we want to stress that basic knowledge of the database engine and SQL optimizer can help you avoid common pitfalls. By doing so, you will not only learn to be more productive but also steer clear of many headaches and frowns from fellow developers along with their flippant references to RTFM, usually followed by a gesture to signal that you are to leave the office quickly.

1.2 System and User Requirements

There are many free (and paid) resources that you can use to learn more about SQL and Oracle. Before we can talk about optimization of SQL queries and PL/SQL scripts, however, you have to understand the basics of SQL. This manual is written for people with at least some experience with SQL or PL/SQL on Oracle databases in a production environment, in particular (aspiring) developers. Should you have only ever looked at the sample databases, you are not likely to gain much by reading these pages; we shall briefly cover a bunch of database basics but they are mainly meant to refresh people's memories.

It is recommended that you read the manual in its entirety, as we have tried our best to gradually go from easy to more advanced optimization techniques. Because this manual is an ongoing work in progress, more content will be added as time advances. Professional developers obviously do not need to read everything; we hope that there is something of interest to professional developers too, but there are no guarantees.

Should you wish to contribute to this project, head on over to the public repository to contact us. And if you ever have a question or comment? Just send us your query.

1.3 Notes

Some advice on these pages is generic in the sense that it applies to other RDBMSs too, but most of it is specific to Oracle. Whenever you see code you can be sure it has been tested on Oracle 11g R2 and/or 12c. Prior versions may not support all functionality assumed henceforth. All guidelines presented here can be considered best practices or good old-fashioned common sense.

Now, let's get cracking!

CHAPTER 2

SQL

Before Edgar F. Codd formulated the relational model *[Codd69] [Codd70]* for database management in 1969/1970 we had the Dark Ages of databases. Applications that required some form of stored data used a database unique to the application. Therefore, each development project had to reinvent the wheel again.

In the 1970s Donald Chamberlin and Raymond Boyce developed the Structured English Query Language (SEQUEL) at IBM to work with data stored in System R, IBM's database management system of the time. Because of trademark dispute they later changed the name to SQL, which stands for Structured Query Language. Because of that there are quite a few people who still pronounce SQL as 'sequel', whereas others say 'S-Q-L'. If you search the internet you are likely to find an equal amount of proponents of either school of pronunciation. Honestly, who gives a hoot!

While IBM were still tinkering with their prototype System R, Larry Ellison, Bob Miner and Ed Oates founded Software Development Laboratories (SDL) in 1977. The first version (1978) was never released officially but the code name 'Oracle', taken from the CIA database project all three founders had worked on while at Ampex Corporation, remains to this day. Another year later they - now calling themselves Relational Software Inc. (RSI) - released Oracle V2, the first commercially available implementation of SQL. It was not until 1979 that IBM had a commercial database product; IBM DB2 came out in 1983. A year before the release of DB2, Oracle decided they had enough of RSI and they became known as Oracle Systems Corporation, named after their primary product. In 1995 Oracle Systems Corporation was rechristened Oracle Corporation.

Oracle is still the leader when it comes to relational database management software, although the competition is getting stronger (especially on Windows platforms where Microsoft is in the lead with SQL Server) and more diverse: NoSQL (Cassandra, MongoDB, Neo4j, ...), NewSQL (Clustrix, H-Store), in-memory databases (IMDB) like SAP's HANA, highly distributed systems for cloud computing like Apache Hadoop, and so on.

A nice indicator of database popularity is DB engines. Their ranking does not compare to in-depth market research by Gartner and the likes but of course it's free and not that far from the truth.

Because of its ubiquity SQL became a standard of ANSI in 1986 and ISO one year later. Before you start shouting 'Halleluja!' from the rooftops, bear in mind that large parts of SQL are the same across the many database vendors but many are not. For example:

- `COALESCE(...)` is the ANSI SQL function that Oracle has implemented but most Oracle SQL developers use `NVL(...)` instead; in SQL Server there is `ISNULL(...)` even though `COALESCE(...)` works too.
- `SELECT 3.1415 AS pi` gives you a table with one row and one column in SQL Server, but all Oracle gives you is an `ORA-00923` error because it did not find the `FROM` keyword where it expected it. Oracle needs `FROM`

DUAL to make the query work.

- Returning ten rows from a result set can be accomplished with SELECT TOP 10 in SQL Server. Oracle requires you apply a filter like WHERE ROWNUM <= 10. To get the actual top-10 rows (based on some ordering), Oracle requires a subquery with an ORDER BY clause, whereas SQL Server allows you to simply issue ORDER BY in the same query. As of 12c it is possible to use the row limiting clause though: FETCH FIRST 10 ROWS ONLY, which comes after the ORDER BY clause.
- Window functions (i.e. aggregate functions in the SELECT clause that are accompanied by an OVER clause with PARTITION BY and/or ORDER BY) are another cause of portability headaches: the SQL:2011 (ISO/IEC 9075:2011) standard defines a window clause (WINDOW) that enables easy reuse (through an alias) of the same window but as of this writing no major RDBMS vendor, save for the open-source PostgreSQL project, has implemented the window clause.
- Hierarchical queries are done with the CONNECT BY clause in Oracle, whereas SQL Server requires recursive common table expressions. Only since 11g R2 does Oracle support recursive common table expressions.
- There are three inequality comparison operators in Oracle: <>, which is the standard operator, !=, which most databases accept too, and ^=, which happens to be supported by IBM DB2 but not by Microsoft SQL Server, for instance.

We could go on with our list but you probably get the message: even if you stick to the ANSI SQL standard as much as possible, you may not end up with portable code. Anyway, portability is by no means necessary: if you spend all your days working with Oracle databases, why care about NVL(...) not being understood by SQL Server? Joe Celko's books are great if you are interested in ANSI-compliant SQL tips and tricks. SQL Performance Explained by Markus Winand as well as his website Use the Index, Luke! come highly recommended for those who seek performance tweaking techniques for multiple database platforms.

As we have said before, we shall assume that you have knowledge of and considerable experience with SQL and Oracle. If not, you can read on but we do not recommend it.

2.1 SQL Basics

The histories of relational database management systems and SQL are inextricably linked. Very few top-notch RDBMSs use a non-SQL programming language as the primary data manipulation language, although a couple of alternatives have been spotted in the wild. The implementations of SQL vary from vendor to vendor, but most share roughly the same core feature set.

Relational database management systems, such as Oracle, are built on two pillars of mathematics: set theory and (first-order) predicate logic. In a database live objects and these objects have certain relations to one another. The objects have properties that are quantifiable and we can use these properties to compare objects.

All data is represented as *n*-ary relations. Each relation consists of both a heading and a body. A heading is a set of attributes, and a body of an *n*-ary relation is a set of *n*-tuples with no specific order of its elements. A sets is an *unordered* collection of *unique* elements: the sets {a,b,c} and {b,c,a,c,c} are equivalent. Whereas mathematicians generally use two-valued logic to reason about about data, databases use three-valued logic: true, false, and unknown (NULL).

Note that up to this point we have not talked about tables or columns at all. So far we have been looking at the high-level conceptual database model, which consists only of entities (e.g. Location, Department, Product, Supplier, Customer, Sales, ...) and relations. The conceptual data model describes *what* we want to model. When we move on to the logical data model, we need to add attributes and primary keys to our entities and relations. We are still independent of the particulars of a database management system but we have to define *how* we want to model our entities and relations. Common logical data models include the relational, network, hierarchical, flat, entity-relationship, and object-relational model. Since Oracle is a relational database management system (RDBMS) we shall focus on that one here.

Once we know what we want to model and how we intend to model our high-level entities and relations, we must *specify* our logical data model, that is, we define our physical data model, which is highly dependent on the RDBMS we use: tables, views, columns, data types, constraints, indexes, procedures, roles, and so on. Attributes are represented by columns, tuples by rows and relations by tables. A nice, brief overview of the three levels of data models is available on 1Keydata.

With the risk of sounding pedantic, we wish to emphasize that tables are logical beasts: they have logical rows and columns. Records and fields are their physical equivalents; fields are housed in the user interfaces of client applications, and records hang out in files and cursors. We shall try and not confuse them but we don't want to make promises we can't keep.

Important to note is that rows can appear more than once in relational databases, so the idea that we can have only distinct elements in sets does not strictly apply; with the `DISTINCT` clause you can again obtain all unique elements but that is not really the point. Multisets provide the appropriate generalization upon which RDBMSs are actually based, but even then SQL will deviate from the relational model. For instance, columns can be anonymous. Yes, we know: Oracle automatically assigns the expression of a column without an alias as the column name when outputting but that does not make it an actual attribute — try accessing it from outside a subquery or CTAS'ing into a new table without an `ORA-00998` error telling you to `name this expression with a column alias`. Anyway, we shall not dwell on any additional idiosyncrasies pertaining to the relational model. In case you do crave for more details on the relational model though, we recommend the book *SQL and Relational Theory* by Christopher J. Date.

2.1.1 Style Guide

Before we talk about the optimization of actual SQL queries in Oracle, we want to take a moment and discuss a few best practices regarding style. These recommendations do not improve the performance of your queries in any way, but they may well increase your productivity, especially when it comes to debugging your code. Other than that, your credibility as a developer might get a slight bump.

2.1.1.1 Conventions

Stick to existing rules regarding style, object nomenclature, comments, and documentation as much as possible. When it comes to object naming, be sure to follow whatever is generally accepted at your organization. For example, are underscores used (`FIRST_NAME`) instead of spaces or is it common to simply concatenate words (`FIRSTNAME`)? If there are no rules or guidelines yet, establish them with your team, write them down with plenty of examples so that they are clear to all, publish them where everyone can see them, and stick to your guns. Although it should be clear, we'll say it anyway: be consistent.

Use the ANSI-standard `JOIN` in `FROM` clauses rather than the deprecated versions with commas and the `(+)` operator for outer joins. It's deprecated, so leave it be.

2.1.1.2 Capitalization

Keywords, reserved words, reserved namespaces and objects (i.e. tables, columns, indexes, ...) are by default case-insensitive in Oracle, unless you have surrounded them by double quotes, like so: `SELECT 42 AS "THE AnsweR" FROM DUAL`. It is generally not recommended that you use case-sensitive object names or names with spaces. Translation of object names into more human-readable formats is something that should ultimately be handled by an application and not the database. Note, however, that strings can be case-sensitive: `SELECT last_name FROM people WHERE last_name = 'Jones'` is different from `SELECT last_name FROM people WHERE last_name = 'jones'`.

2.1.1.3 Semicolons

Sentences end with full stops, SQL statements with semicolons. Not all RDBMS clients require a semicolon to execute a single SQL statement, but you save yourself a lot of trouble if you just learn to finish each statement with a semicolon.

2.1.1.4 Asterisks

Never use `SELECT *` in production code. At some point, someone will come and modify the table or view you're querying from. If, on the one hand, the column you need in your application has been removed, you'll end up with an application that displays an error. Best case: you're alerted by an automated unit test that fails, so you branch off and fix the issue before merging it back into the main repository. Worst case: your client calls you and says that the application displays a runtime error, although the feedback is usually more along the lines of 'It does not work'. If, on the other hand, several columns have been added you grab more than you actually need, thereby generating unnecessary overhead in the database and additional network traffic, which bring us to the next point:

2.1.1.5 Thrift

Grab only the data you really need. If a table has a hundred columns and you only need three of them, do not select everything 'just in case'. You don't go to a car dealer and buy two cars just in case the first one breaks down, do you? Take what you need: no more, no less.

The same goes for subqueries: if you reuse a subquery multiple times in a larger query, don't copy-paste it. Instead use a subquery factor or common table expression (i.e. `WITH` clause). It makes your code easier to read, you don't have to update your subquery in several places if you ever need to make changes, and more importantly, Oracle can avoid doing the same thing multiple times. Oracle sometimes caches a subquery that appears repeatedly in your query, but there is no guarantee.

Factor your code in general. Portions of stored procedures (or user-defined functions) that you use frequently should become their own stored procedures (or functions). Whenever a (small) portion of a procedure or function needs to be modified, factored code can minimize the recompilation. Just because you are working on a database does not mean you can ignore good code design altogether.

When your result set needs to be sorted, use the `ORDER BY` clause, but do not force Oracle to sort data when you do not require it to be so. Oracle generally ignores irrelevant `ORDER BY` clauses in subqueries, but it's sloppy to leave them in your code, and it can have an adverse effect on performance in case Oracle does not decide to ignore it. Moreover, views with `ORDER BY` clauses cause multiple sorts to be performed whenever someone selects data from the view but in a different order.

Don't go nuts with minimalism though. Never use ordinals (a.k.a. the column position) to sort data in production code. Specify the column names (or aliases) in the `ORDER BY` clause and you won't have any problems when someone alters the code. The same applies to the column list in `INSERT` statements; never ever assume that the order in which the columns are provided matches the table you are adding data to, even though the data types happen to match, and that the order of both the source and the target will always stay the same.

2.1.1.6 Aliases

When you are dealing with more than one table (or view), use *meaningful* aliases. It reduces the amount of typing and it makes reading the query easier on the eyes. The adjective meaningful is there to remind you that `x` and `y` are probably not that revealing, and they do no nothing to aid the legibility of your code. Moreover, when defining column aliases, use `AS`. Its use is optional but sometimes it can be hard to figure out whether you missed a comma between two column names or whether the alias for one column is supposed to be the name of another.

2.1.1.7 Comments

Add meaningful comments to your code: either use `/* ... */` for (multiline) comment blocks or `--` for comments that do not extend to the next line. The key word here is *meaningful*. Trivial comments should not be added as they clutter your code and are immediately obvious to all but the brain-dead.

Add meaningful comments to the data dictionary with the `COMMENT` statement. You can add comments to tables, (materialized) views, columns, operators and index types.

Note: You can automatically generate documentation (HTML, PDF, CHM, ...) from the metadata in the data dictionary (`SELECT * FROM dictionary`) with for instance the option to 'Generate DB Doc' from the connections window/tab in Oracle SQL Developer, Quest Toad's 'HTML Schema Doc Generator' in the Database > Report menu. Specialized tools to extract and display metadata from Oracle's data dictionary exist too: for example, the xSQL's excellent Database Documenter or the free SchemaSpy.

2.1.1.8 Constraints

We've said it before and we are going to say it again: be consistent. Especially when it comes to constraints that force user data into straitjackets. Constraints are imperative to databases. However, when you add `NOT NULL` constraints to columns that can have missing data (`NULL`), you force users to enter rubbish. As they will soon find out after receiving an error message: a blank space will often do the trick. Before you think about adding `TRIM(...)` or `REGEXP_LIKE(...)` checks to all data entered manually, think again: users will also quickly figure out that any random character (combination) will work and you cannot account for all possible situations. Prior to 11g you may have needed to convert `NULL` to `'N/A'` or something similar to allow indexing on missing values, but that is not necessary any longer. The link shows a function-based B-tree index that includes columns with `NULL`. By the way, bitmap indexes include rows with `NULL`; the default index is a B-tree index though.

2.1.1.9 Respect

No, you don't have to get all Aretha Franklin over your database, but you have to respect data types. Never rely on implicit data type conversions, and always convince yourself that the data type you think applies, really does apply. With a simple `DESC tab_name` you can remove all doubt.

If you're not convinced, please take a look at the following example, which shows you what you get when you sort numerical-looking data that is actually stored as a string.

```
WITH
  raw_data AS
  (
    SELECT 1 AS int_as_number, '1' AS int_as_varchar FROM dual
    UNION ALL
    SELECT 2 AS int_as_number, '2' AS int_as_varchar FROM dual
    UNION ALL
    SELECT 3 AS int_as_number, '3' AS int_as_varchar FROM dual
    UNION ALL
    SELECT 12 AS int_as_number, '12' AS int_as_varchar FROM dual
    UNION ALL
    SELECT 28 AS int_as_number, '28' AS int_as_varchar FROM dual
  )
SELECT * FROM raw_data ORDER BY int_as_varchar;
```

The moral: do not assume anything when it comes to data types. Just because something looks like a number does not mean that it is stored as a number.

2.1.1.10 Formatting

Format your SQL queries and format them consistently. Better yet, use either a built-in formatter or use an online formatter. Make sure you use the same formatting rules as your colleagues: it helps making sharing and analysing each other's code so much easier. It may come as a surprise but the actual format matters, even spaces! The result set that Oracle fetches for you does not depend on spaces but whether it needs to parse a statement with a single space extra. We shall talk more about (hard/soft) parsing of statements later when we discuss *execution plans*, but for now suffice to say that each query needs to be hashed and analysed by Oracle before it can execute it. If the query hashes are the same, which generally means that the query you have submitted is formatted identically as one in memory (the system global area (*SGA*) to be precise), Oracle can immediately execute it. If not, Oracle needs to analyse your query first. As said on DBA Oracle, the time Oracle needs to parse a statement is almost negligible, but when many users issue functionally and syntactically identical yet symbolically distinct statements, the small amounts of time can quickly add up.

Although there is no general consensus about good formatting rules, you can add line breaks in appropriate places, so you are able to comment or uncomment lines without having to manually reformat your code every time. This is particularly useful when you are debugging more complex queries. To do so, insert line breaks

- before and after `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `FROM`, `JOIN`, `ON WHERE`, `CONNECT BY`, `START WITH`, `GROUP BY`, `HAVING`, and `ORDER BY`
- before and after `DECLARE`, `BEGIN`, `END`, `LOOP`, `EXCEPTION` in PL/SQL blocks
- after `AS` or `IS` in `CREATE` statements
- before `WHEN`, `ELSE`, and `END` in `CASE` statements
- before `AND` and `OR`
- before commas
- before semicolons
- after the first, and before the last bracket of a large expression.

2.1.1.11 Coding Guidelines

We recommend that each organization define a programming standards document that clearly specifies how to write consistent and maintainable code. At the very least the coding standards should tell you how to name objects and format code. That includes, but is not limited to, standard prefixes for all database objects, notation standards (e.g. keywords in upper case, application-specific identifiers in lower case, underscores between words in identifiers), maximum line length, line break rules, indentation spaces for code blocks, and default headers. If your IDE supports IntelliSense or something similar, then Hungarian notation may be overkill, but for complex programs it may be beneficial to prefix the logical (Apps Hungarian) or the physical (Systems Hungarian) type to avoid collisions, although the former is often to be preferred to the latter.

An example of a comprehensive set of coding guidelines for both SQL and PL/SQL is by Ian Hellström. The document's source is in Markdown and publicly available in order to make it easy for you to adapt it to your (organization's) needs. Steven Feuerstein's and topcoder's best practices and programming standards focus mainly on PL/SQL.

2.1.2 Query Processing Order

Important to understand before we discuss execution plans is how Oracle processes queries logically. Let's look at the following query:

```
1 SELECT
2     f.product AS beer
3   , p.product AS crisps
```

```
FROM
  fridge f
CROSS JOIN
  pantry p
WHERE
  f.product         = 'Beer'
  AND f.temperature < 5
  AND f.size        = '50 fl oz'
  AND p.product     = 'Crisps'
  AND p.style       = 'Cream Cheese'
  AND p.size        = '250g'
ORDER BY
    crisps
  , beer
;
```

What does this query tell you other than that you're a tad peckish, extremely thirsty, and that the fridge and pantry seem to use different systems of measurement?

You may think that it reads the query in the way that we type it, but Oracle (and other RDBMSs too) does not read from top to bottom. It more or less reads our queries upside down. Not exactly, but we'll see what it does in a minute. Oracle is a fancy machine that translates our SQL statements into something it can understand and execute. In essence, it's a data robot that does exactly what we tell it to do. Now, suppose you have purchased a robot to help you around the house, and its first and foremost task is to assist you in quenching your thirst and sating your appetite. How would you tell it go fetch a beer and a packet of crisps?

Well, you'd probably tell it to go to the fridge, look for beer, grab a bottle (50 fl oz) with a temperature below 5 degrees Celsius, then go to the pantry and look for a 250g packet of cream cheese crisps. Once it's done, it should come back to you and place the items in front of you, sorted in the way you asked it to do. That's right, you first tell it to go to the place where the fridge and the pantry are located (probably the kitchen: `FROM`), then to look for everything that matches your criteria (`WHERE`), and finally to return the items (`SELECT`) sorted in the order you specified (`ORDER BY`).

That's pretty much what Oracle does too. The order in which clauses are logically processed by Oracle is as follows: `FROM -> CONNECT BY -> WHERE -> GROUP BY -> HAVING -> SELECT -> ORDER BY`.

Of course, your query does not have to have every clause, and some cannot even be used with/without others (e.g. `HAVING` can only be used when you use `GROUP BY`).

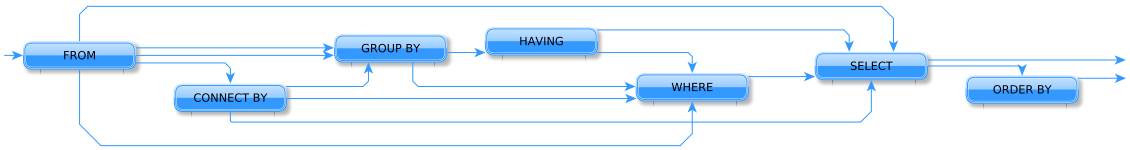


Fig. 2.1: Oracle's query processing order, including optional clauses.

The processing order is also the reason why the previous query worked like a charm and the following will result in an error:

```
SELECT
    product          AS item
  , MIN(temperature) AS min_temperature
  , COUNT(*)         AS num_products
FROM
  fridge
GROUP BY
```

```
  item
;
```

When Oracle processes the `GROUP BY` clause the alias `item` is not yet known, so you are greeted by an `ORA-00904: invalid identifier` error.

2.2 Execution Plans

What happens to your SQL statement when you hit execute?

First, Oracle checks you statement for any glaring errors. The syntax check verifies whether the language elements are sequenced correctly to form valid statements. If you have neither made any typos in keywords and the like nor sequenced the language elements improperly, you're good for the next round. Now Oracle moves on to evaluate the meaning of your syntactically legal code, which is known as semantic analysis. All references to database objects and host variables are scrutinized by Oracle to make sure they are valid. Oracle also inspects whether you are authorized to access the data. These checks expand views referenced by your statement into separate query blocks. For more details we refer you to the chapter *Syntactic and Semantic Checking* of the *Programmer's Guide to the Oracle Precompilers* for your database version.

Once your SQL statement has passed both checks with flying colours, your statement receives a SQL ID and (MD5) hash value. The hash value is based on the first few hundred characters of your statement, so hash collisions can occur, especially for long statements.

You can find out the SQL ID and hash value of your SQL statement by querying `v$sql`. To make life easier it is often best to add a comment unique to your statement, for instance `SELECT /* my_custom_comment */ select_list FROM tab_name`. Then you can simply look for your query from `v$sql`:

```
SELECT
    sql_id
  , hash_value
  , plan_hash_value
  , sql_text
FROM
  v$sql
WHERE
  sql_text LIKE 'SELECT /* my_custom_comment */%'
;
```

In case you happen to know the SQL ID already and would like to know the corresponding hash value, you can use the function `DBMS_UTILITY.SQLID_TO_SQLHASH`, which takes the sql_id (`VARCHAR2`) and returns a `NUMBER`.

Note: *All* characters affect the hash value, including spaces and line breaks as well as capitalization and of course comments.

The last stage of the parser is to look for possible shortcuts by sifting through the *shared pool*, which is a "portion of the SGA that contains shared memory constructs such as shared SQL areas", which hold the parse tree and execution plans for SQL statements; each unique statement has only one shared SQL area. We can distinguish two cases: hard and soft parses.

1. **Soft parse** (library cache hit): if the statement hashes to a value that is identical to one already present in the shared pool *and* the texts of the matching hash values are the same *and* its parsed representation can be shared, Oracle looks up the execution plan and executes the statement accordingly. Literals must also be the same for Oracle to be able to use the same shared SQL area. The exception is when `CURSOR_SHARING` is set to `FORCE`.

2. **Hard parse** (library cache miss): if the statement has a hash value different from the ones that are available in the SGA *or* its parsed representation cannot be shared, Oracle hands the code over to the query optimizer. The query optimizer then has to build an executable version from scratch.

Criteria for when a SQL statement or PL/SQL block can be shared are described in the *Oracle Database Performance Tuning Guide*, which can be found here. Basically, the statements' hashes and texts, all referenced objects, any bind variables (name, data type, and length), and the session environments have to match. PL/SQL blocks that do not use bind variables are said to be not re-entrant, and they are always hard-parsed. To find out why statements cannot be shared you can use the view `v$sql_shared_cursor`.

Perhaps you noticed that we had sneaked in the column `plan_hash_value` in the `v$sql` query above. SQL statements with different hash values can obviously have the same plan, which means that their plan hash values are equal.

Note: The plan hash value is merely an indicator of similar operations on database objects: filter and access predicates, which we shall discuss in more detail, are not part of the plan hash value calculation.

For hard parses, the next station on the *SQL compiler* line is the query optimizer. The query optimizer, or just optimizer, is the "built-in database software that determines the most efficient way to execute a SQL statement". The optimizer is also known as the cost-based optimizer (CBO), and it consists of the query transformer, the estimator, and the plan generator:

- The query transformer "decides whether to rewrite a user query to generate a better query plan, merges views, and performs subquery unnesting".
- The estimator "uses statistics [from the data dictionary] to estimate the selectivity, cardinality, and cost of execution plans. The main goal of the estimator is to estimate the overall cost of an execution plan".
- The plan generator "tries out different possible plans for a given query so that the query optimizer can choose the plan with the lowest cost. It explores different plans for a query block by trying out different *access paths*, join methods, and join orders". The optimizer also evaluates expressions, and it can convert correlated subqueries into equivalent join statements or vice versa.

What the optimizer does, in a nutshell, is apply fancy heuristics to figure out the best way to execute your query: it calculates alternate routes from your screen through the database back to your screen, and picks the best one. By default Oracle tries to minimize the estimated resource usage (i.e. maximize the throughput), which depends on I/O, CPU, memory, the number of rows returned, and the size of the initial data sets. The objective of the optimization can be altered by changing the value of `OPTIMIZER_MODE` parameter.

If you can recall our *example* of the robot, the beer, and the packet of crisps, you may remember that the robot had to check both the pantry and the fridge. If we equip our robot with Oracle's query optimizer, the robot will not simply walk to the kitchen, find the items by searching for them, and then return with our refreshments, but try to do it as efficiently as possible. It will modify our query without altering the query's function (i.e. fetch you some booze and a few nibbly bits), and explore its options when it comes to retrieving the items from the kitchen. For instance, if we happen to have a smart fridge with a display on the door that shows where all bottles are located, including the contents, temperature, and size of each bottle, the robot does not have to rummage through decaying fruit and vegetables at the bottom in the hope that a bottle is located somewhere underneath the rubbish. Instead it can look up the drinks (from an index) and fetch the bottles we want by removing them from the spots highlighted on the display (by ROWID). What the optimizer can also figure out is whether it is more advantageous to grab the beers and then check the pantry, or the other way round (join order). If the robot has to go down a few steps to obtain crisps while still holding the beer in its hands, the optimizer may decide that carrying the heavy and/or many beverages may be inefficient. Furthermore, it may decide to pick up a tray and place the products it has already extracted on it (temporary table) while continuing its search for the remaining items. Because the optimizer evaluates expressions it will know whether or not it has to do something: a predicate like `0 = 1` will be immediately understood by the optimizer, so that our friendly robot knows it has to do bugger all beforehand.

After the optimizer has given its blessings to the optimal execution plan, the row source generator is let loose on that

plan. The row source generator produces an iterative plan, which is known as the query plan. The query plan is a binary program that produces the result set when executed. It is structured as a row source tree, where a row source is the combination of a set of rows returned by a step in the execution plan and "a control structure that can iteratively process the rows", that is one row at a time. The row source tree shows an ordering of the tables, an access method for each table, a join method for tables affected by join operations, and data operations (filter, sort, aggregation, etc.)

During execution, the SQL engine executes each row source in the tree produced by the row source generator. This step is the only mandatory step in DML processing.

More information on the optimizer can be found on the Oracle Technology Network in the Database Concepts documentation under the the section *SQL* and the subsection *Overview of the Optimizer*.

2.2.1 Explain Plan

As a developer, the execution plan is probably the best tool at your disposal to discover why a query performs the way it does *and*, if necessary, figure out what you can do about its performance. A 2011 Oracle white paper called *Explain the Explain Plan*, which is available on the OTN, and the documentation on optimizer access paths form the basis for most of what is written in this section. Additionally, Jonathan Lewis has written an excellent series on the execution plan, the parts of which you can find here, here, and here.

So, how do you obtain the execution plan?

If you happen to work with Oracle SQL Developer you can simply press F10 to see the execution plan of the query selected. It does not get much easier than that. Similarly, you can execute `EXPLAIN PLAN FOR your_query;` and then run

```
SELECT
  *
FROM
  table
    (
      DBMS_XPLAN.DISPLAY
        (
           'plan_table'  -- default plan table name
         , NULL          -- NULL to show last statement inserted into plan table
         , 'all'         -- show all available metrics
        )
    );
```

What you end up with is a tabular representation of the execution plan; the table is a top-down, left-to-right traversal of the execution tree. Each operation is printed on a single line, together with more detailed information about that operation. The sequence and types of operations depend on what the query optimizer considers to be the best plan, that is the one with the lowest cost. The components that are factored into the cost of a query are

- Cardinality
- Access methods
- Join methods
- Join types
- Join orders
- Partition pruning
- Parallel execution

We shall now discuss each of these components separately.

2.2.1.1 Cardinality

No, cardinality has nothing to do with the clergy. What cardinality refers to is the uniqueness (or lack thereof) of data in a particular column of a database table. It is a measure of the number of distinct elements in a column. A low cardinality implies that a column has very few distinct values, and is thus not very selective.

In the context of execution plans, the cardinality shows the number of rows estimated to come out of each operation. The cardinality is computed from table and column statistics, if available.[1] If not, Oracle has to estimate the cardinality. For instance, suppose you have an equality predicate in a single-table query, for which there is no histogram (i.e. no statistics). The query optimizer will then assume a uniform distribution, so that the cardinality of each operation in the execution plan is calculated by dividing the total number of rows by the number of distinct values after the equality predicate has been applied. The number of rounded up and shown in the column `CARDINALITY`.

What can (negatively) impact the accuracy of the estimate and therefore the quality of the execution plan are i) data skew, ii) multiple single-column predicates on a single table, iii) function-wrapped columns in `WHERE` clause predicates, and iv) complex expressions.

Interestingly, you can see runtime cardinality information by using the `GATHER_PLAN_STATISTICS` hint in your query, after which you have to execute `SELECT * FROM table(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'))`. The result shows you the estimated number of rows (`E-Rows`) and the actual number of rows (`A-Rows`), which can of course be quite different because of data skew. Don't use this hint in a production environment as each query incurs some overhead.

2.2.1.2 Access Methods

The way in which Oracle accesses data is aptly called an access method. Oracle has a bunch of access methods in its arsenal:

- A **full table scan** reads all rows from a heap-organized table and filters out the ones that do not match the `WHERE` clause predicates. Each formatted block of data under the high water mark (HWM) is read only once and the blocks are read in sequence. A multi-block read can be performed to speed up the table scan: several adjacent blocks are combined into a single I/O call. The `DB_FILE_MULTIBLOCK_READ_COUNT` parameter in the init.ora file defines the number of blocks that can be lumped into one multi-block read.

  A full table scan is typically employed if

  - no index exists;
  - the index cannot be used because the query predicate has a function applied to an indexed column (in a non-function-based index);
  - the optimizer decides against an index skip scan because the query predicate does not include the leading edge of a (B-tree) index;
  - `SELECT COUNT(*)` is issued and the index contains nulls;
  - the table statistics are stale and the table has grown considerably since the statistics were last computed;
  - the query is not selective, so that a large portion of the rows must be accessed;
  - the cost of a full table scan is the lowest because the table is small, in particular the number of formatted blocks under the high water mark is smaller than `DB_FILE_MULTIBLOCK_READ_COUNT`;
  - the table has a high degree of parallelism, which makes the optimizer biased towards full table scans;
  - the query uses the `FULL` hint.

[1] By default, Oracle determines all columns that need histograms based on usage statistics and the presence of data skew. You can manually create histograms with the `DBMS_STATS.GATHER_TABLE_STATS` procedure.

- In a **table access by ROWID**, Oracle looks up each selected row of a heap-organized table based on its ROWID, which specifies the data file, the data block within that file, and the location of the row within that block. The ROWID is obtained either from the `WHERE` clause predicate or through an index scan. If the execution plan shows a line `TABLE ACCESS BY INDEX ROWID BATCHED` it means that Oracle retrieves a bunch of ROWIDs from the index and then tries to access rows in block order to reduce the number of times each block needs to be accessed.
- The `SAMPLE` and `SAMPLE_BLOCK` clauses (with a sample percentage below 100%) cause a **sample table scan**, which fetches a random sample of data from a heap-organized table. Note that block sampling is only possible during full table scans or index fast scans.
- For an **index unique scan** only one row of a (B-tree) index or index-organized table will be returned because of an equality predicate on a *unique* index or a primary key constraint; the database stops looking for more rows once it has found its match because there cannot be any more matches thanks to the `UNIQUE` constraint. Hence, *all* columns of a unique index must be referenced with equality operators. Please observe the index itself must be unique: creating an index on a column and subsequently adding a `UNIQUE` or `PRIMARY KEY` constraint to the table is not enough, as the index is not aware of uniqueness; you are liable to end up with an index range scan.
- An **index range scan** scans values in order. By default, Oracle stores and scans indexes and index-organized tables in ascending order. Oracle accesses adjacent index entries and uses the ROWID values to retrieve the rows in ascending order. If multiple index entries happen to have the same keys, Oracle returns the entries in ascending order by ROWID.

  The database chooses an index range scan if the leading columns of a *non-unique* index are specified in the `WHERE` clause or if the leading columns of a *unique* index have ranges rather than single values specified. Oracle navigates from the root blocks to the branch blocks where it reads the maximum values of the leading edge in the index for each leaf block that the branch blocks refer to. Because the database has no way of knowing in advance how many hits there are, it must visit each relevant leaf block and read each one until it does not find any more matches. The advantage of the index unique scan is that Oracle does not have to visit the leaf blocks at all, because once it has found its match it is done. Not so with the index range scan.

  A common gripe with the index range scan is in combination with the table access by ROWID method, especially if the index range scan includes filter predicates instead of mere access predicates. Filter predicates in conjunction with index range scans and tables access by ROWID can cause scalability issues, as the entire leaf node chain has to be accessed, read, and filtered. As more and more data is inserted, the time to access, read, and filter the data increases too.

  There is also an **index range scan descending**, which is basically the same beast; the only difference is that the rows are returned in descending order. The reason Oracle scans the index in descending rather than ascending order is because either the query has an `ORDER BY ... DESC` clause or the predicates on a key with a value less than instead of equal to or greater than a given value are specified. Another (obvious) cause is the `INDEX_DESC` hint.
- If the entire index is read in order, then we are dealing with an **full index scan**. A full index scan does not read every block in the index though. Instead it reads the root block and goes down the left-hand side (ascending scan) or right-hand side (descending scan) of the branch blocks until it reaches a leaf block. From there it reads the index on a block-by-block basis.

  The full index scan is used when one of the following situations arises:

  - a predicate references a (non-leading) column in an index;
  - no predicate is specified but all columns in the table as well as query are in the index, and at least one indexed column is not null;
  - the query has an `ORDER BY` clause on indexed non-null columns;
  - the query has a `GROUP BY` clause where all aggregation columns are present in the index.

The query optimizer estimates whether it is cheaper to scan the index (full index scan) or table itself (full table scan); for index-organized tables the table and index are of course one and the same.

- A **fast full index scan** reads index blocks as they exist on disk. The index (entries on the leaf blocks) rather than the table is read in multi-block I/O mode.

  Oracle looks at a this access path whenever a query only asks for attributes in the index. It's an alternative to a full table scan when the index contains all columns needed for the query, and at least one column in the index key has a `NOT NULL` constraint.

  Because the index is not read in order (because of the multi-block I/O), a sort operation cannot be eliminated as with the full index scan.

- For queries with predicates on all but the leading column of a composite index, an **index skip scan** is a possibility. The index skip scan is also an option if the leading column has few distinct values.

  The optimizer logically splits a composite index into smaller sub-indexes based on the number of distinct values in the leading column(s) of the index. The lower the number of distinct values, the fewer logical sub-indexes, the more efficient the scan is. Index blocks that do not meet the filter condition on the non-leading column are immediately skipped, hence the name. An index skip scan can of course only be efficient if the non-leading columns are highly selective.

- An **index join scan** is performed if all data can be retrieved from a combination of multiple indexes, which are hash-joined on the ROWIDs. Because all data is available in the indexes, no table access is needed.

  An index join is often more expensive than simply scanning the most selective index and subsequently probing the table. It cannot be used to eliminate a sort operation.

- Whereas in traditional B-tree indexes each entry point refers to exactly one row, a bitmap index's entry points refer to multiple rows. That is, each index key stores pointers to multiple rows. Each bit corresponds to a possible ROWID; if the bit is set, then the row with the corresponding ROWID contains the key's value. The bit position is converted to an actual ROWID by a mapping function. Internally, Oracle stores the bitmap index in a B-tree structure for quick searching.

  Bitmaps are frequently used in data warehousing (OLAP) environments, where ad hoc queries are commonplace. Typically, a bitmap index is recommended if the indexed columns have low cardinality and the indexed table is rarely modified or even read-only. Bitmap indexes are not appropriate for OLTP applications, though. If, for instance, the indexed column of a single row is updated, the database locks the index key entry, which points to many rows. Consequently, all these rows are locked.

  Back to business. If the predicate on a bitmap-indexed column contains an equality operator, the query optimizer considers the **bitmap index single value** access path to look up a single key value. A single bitmap is scanned for all positions containing a value of `1`. All matching values are converted into ROWIDs, which in turn are used to find the corresponding rows.

- A B-tree index can have an index range scan for ranges of values specified in the `WHERE` clause. Its counterpart for bitmap indexes is the **bitmap index range scan**.

- A **bitmap merge** is typically preferred by the optimizer when bitmaps generated from a bitmap index range scan are combined with an `OR` operation between two bitmaps.

- When a query accesses a table in an indexed cluster, the optimizer mulls over a **cluster scan**. Tables in table clusters have common columns and related data stored in the same blocks. The proximity of the physical location of these shared columns reduces disk I/O when joining clustered tables and accessing related data. Table clusters are appropriate for tables that are rarely modified and do not require full table scans as in the case of retrieval of a lot of rows.

  An index cluster is, as you might expect, a (B-tree) index on a cluster; the cluster index associates the cluster key with the database block address of the block with the relevant data. In a cluster scan, Oracle scans the index to obtain the database block addresses of all rows required. Because the index cluster is a separate entity, reading

or storing rows in a table cluster requires at least two I/Os: one (or more) to retrieve/store the key value from/in the index, and one to read/write the row from/to the cluster.

- If you create a table cluster with the `HASHKEYS` clause, you end up with a hash cluster. A hash cluster is similar to an indexed cluster, only the index key is replaced with a hash function. All rows with the same hash are stored in the same data block. Because no separate cluster index exists as in the case with an indexed cluster, I/O can be usually halved.

  A **hash scan** is used to locate rows based on a hash value of the key in a hash cluster. A disadvantage of hash clusters is the absence of range scans on cluster keys that are not in the index, in which case a full table scan must be performed.

2.2.1.3 Join Methods

Join methods refer to the way in which two row sources are joined with each other. The query optimizer designates one table the outer table and the other the inner table. If more than two tables need to be joined, the optimizer analyses all permutations to determine the optimal join sequence.

The join method dictates to a large degree the cost of joins:

- A **hash join** is usually preferred for (equi)joining large data sets. The query optimizer takes the smaller of two data sources to build a (deterministic) hash table in memory. It then scans the larger table and performs the same hashing on the join columns. For the in-memory hash table, the database probes each value and if it matches the predicate's conditions returns the corresponding row.

  If the smaller data set fits in memory, there is only the cost of a single read pass over both data sets. If the hash table does not fit in memory, then Oracle partitions it. The join is then performed partition by partition, which uses a lot of sort area memory and causes a lot of I/O to the temporary tablespace.

- A **nested loop join** is typically used when small subsets of tables are joined or if there is an efficient way of accessing the inner table, for example with an index lookup. For every row selected from the outer table, the database scans *all* rows of the inner table. If there is an index on the inner table, then it can be used to access the inner data by ROWID.

  The database can read several rows from the outer row source in a batch, which is typically part of an adaptive plan.

  It is not uncommon to see two nested loops in the execution plan (as of 11g) because Oracle batches multiple I/O requests and process these with a vector I/O, which means that a set of ROWIDs is sent to the requesting operating system in an array. What Oracle does with two nested loops is basically the following:

  1. Iterate through the inner nested loop to obtain the requested outer source rows.
  2. Cache the data in the PGA.
  3. Find the matching ROWID from the inner loop's inner row source.
  4. Cache the data in the PGA.
  5. Organize the ROWIDs for more efficient access in the cache.
  6. Iterate through the outer loop to retrieve the data based on the cached ROWIDs; the result set of the inner loop becomes the outer row source of the outer nested loop.

- A **sort-merge join** is done when join conditions are inequalities (i.e. not an equijoin). It is commonly chosen if there is an index on one of the tables that eliminates a sort operation. The sort operation on the first data set can be avoided if there is such an index. The second data set will always be sorted, regardless of any indexes. It generally performs better for large data sets than a nested loop join.

  A sort-merge join proceeds in two steps:

  1. Sort join: both tables are sorted on the join key. 1. Merge join: sorted tables are merged.

Oracle accesses rows in the PGA instead of the SGA with a sort-merge join, which reduces logical I/O because it avoids repeated latching blocks in the database buffer cache; a *latch* protects shared data structures in the SGA from simultaneous access.

- Whenever a table has no join conditions to any of the other tables in the statement, Oracle picks a **Cartesian join**, which is nothing but a Cartesian product of the tables. Because the number of rows of a Cartesian join of two tables is the product of the number of rows of the tables involved, it is generally used only if the tables are small. This join method is, however, very rare in production environments.

Partial join evaluation is available from Oracle Database 12c onwards. It allows joined rows that would otherwise have to be eliminated by a `SORT UNIQUE` operation to be removed during the execution of the join an inner join or semi-join instead. This optimization affects `DISTINCT`, `MIN()`, `MAX()`, `SUM(DISTINCT)`, `AVG(DISTINCT)`, `COUNT(DISTINCT)` , branches of `UNION`, `MINUS`, and `INTERSECT` operators, `[ NOT ] EXISTS` subqueries, and so on. For instance, a `HASH JOIN > SORT UNIQUE` is replaced by a `HASH JOIN SEMI > HASH UNIQUE` combo.

2.2.1.4 Join Types

Join types are easier to explain because they are determined by what you have typed. There are four categories of join types: i) inner joins, ii) outer joins (left, right, and full), iii) semi-joins, and iv) anti-joins. Even though semi-joins and anti-joins are syntactically subqueries, the optimizer beats them until they accept their fate as joins.

Semi-joins can occur when the statement contains an `IN` or `EXISTS` clause that is not contained in an `OR` branch. Analogously, anti-joins are considered if the statement contains a `NOT IN` or `NOT EXISTS` clause that is not contained in an `OR` branch. Moreover, an anti-join can be used the statement includes an outer join and has `IS NULL` applied to a join column.

2.2.1.5 Join Orders

So, how does Oracle determine the order of joins? The short answer is cost. Cardinality estimates and access paths largely determine the overall cost of joins:

- Whenever a particular join results in at most one row (e.g. because of `UNIQUE` or `PRIMARY KEY` constraints) it goes first.
- For outer joins, the row-preserving (outer) table comes after the other tables to ensure that all rows that do not satisfy the join condition can be added correctly.
- When Oracle converts a subquery into an anti- or semi-join, the subquery's table(s) come after tables in the outer (connected/correlated) query block. Hash anti-joins and semi-joins can sometimes override the ordering though.
- If view merging is impossible, then all tables in the view are joined before joining the view to the tables outside the view.

2.2.1.6 Partition Pruning

Partition pruning, which is also known as partition elimination, is used for partitioned tables when not all partitions need to be accessed.

Pruning is visible in the execution plan as integers in the columns `PSTART`, the number of the first partition, and `PSTOP`, the number of the last partition to be accessed.[2] If you do not see a number (e.g. `KEY`), don't worry! It means that Oracle was not able to determine the partition numbers at parse time but thinks that dynamic pruning (i.e.

[2] If a table has n partitions (range partition) and m sub-partitions per partition, then the numbering is from 1 to $n \cdot m$. The absolute partition numbers are the actual physical segments.

during execution) is likely to occur. This typically happens when there is an equality predicate on a partitioning key that contains a function, or if there is a join condition on a partitioning key column that, once joined with a partitioned table, is not expected to join with all partitions because of a filter predicate.

For hash-partitioned tables, there can only be pruning if the predicate on the partition key column is an equality or `IN`-list predicate. If the hash partition key involves more than one column, then all these columns must appear in the predicate for partition pruning to be able to occur.

2.2.1.7 Parallel Execution

Sometimes Oracle executes statements in parallel, which can significantly improve the runtime performance of a query. The query coordinator (QC) initiates the parallel execution of a SQL statement. The parallel server *processes* (PSPs) are responsible for the actual work that is done in parallel. The coordinator distributes the work to the PSPs, and may have to do some of the work that cannot be done in parallel. A classic example is the `SUM(...)` operation: the PSPs calculate the subtotals but the coordinator has to add the subtotals from each PSP to obtain the final tally.

All lines in the execution plan above `PX COORDINATOR` are taken care of by the query coordinator. Everything below is done by the PSPs.

A granule is the smallest unit of work a PSP can perform. Each PSP works exclusively on its own granule, and when it is done with the granule, it is given another one until all granules have been processed. The degree of parallelism (DOP) is typically much smaller than the total number of granules.

The most common granules are block-based. For block-based granules, the `PX BLOCK ITERATOR` iterates over all generated block range granules. However, it is sometimes advantageous to make use of pre-existing data structures, such as in the case of partitioned tables. For partition-based granules, each PSP will do work for all data in a single partition, which means that the number of sub-partitions that need to be accessed is typically larger than or equal to the degree of parallelism. If there is skew in the sizes of the (sub-)partitions, then the degree of parallelism is generally smaller than the number of (sub-)partitions. Partition-based granules show up in the execution plan as `PX PARTITION`. Whether Oracle uses block-based or partition-based granules cannot be influenced.

PSPs work together in pairs as producers and consumers (of rows). Producers are below the `PX SEND` operation in the execution plan. The line `PX RECEIVE` identifies consumers who eventually send their results to the query coordinator: `PX SEND QC`. Similar information is shown in the column `TQ` (table queue).

Occasionally data needs to be redistributed, especially in joins, which is shown in the columns `IN-OUT` and `PQ Distrib` (parallel query distribution) of the execution plan. Producers scan the data sources and apply `WHERE` clause predicates, after which they send the results to their partner consumers who complete the join. `IN-OUT` shows whether a parallel operation is followed by another parallel operation (`P->P`) or a serial operation (`P->S`). On the line with `PX SEND QC` you always encounter `P->S`, because the PSPs send their results to the QC, which is a serial process. If you happen to see `P->S` below that line it means that there is a serialization point. This may be due to not having set the parallel degree on one or more objects in the query. The flag `S->P` often indicates that there is a serial bottleneck because parallel processes have to wait for the serial process to finish.

You may also encounter `PCWP` and `PCWC`, or parallel combined with parent and parallel combined with child, respectively. This means that a particular step is executed in parallel including the parent or child step. An example is a parallel nested loop join, for which each PSP scans the outer (driving) row source but does the index lookups on the inner row source too. If such an operation carries the label `BUFFER(ED)`, it tells you that Oracle needs to temporarily hold data between parallel processes, because there is no (parent) PSP available that can accept the data.

Note: The last operation before `PX SEND QC` always shows a buffered blocking operation.

If the amount of data that needs to be buffered is larger than what can reasonably fit into memory, the data needs to written to the temporary tablespace, after which the PSPs have to re-read it. You will thus incur significant penalties for having your queries executed in parallel.

There are several methods that can show up in the column `PQ Distrib` of the execution plan:

- `HASH`: a hash function is applied to the join columns, after which a particular consumer receives the row source based on the hash value. This is by far the most common distribution method.
- `BROADCAST`: the rows of the smaller of two data sources are sent to all consumers in order to guarantee that the individual server processes are able to complete the join.
- `RANGE`: each PSP works on a specific data range because of parallel sort operations, so that the coordinator merely has to present the PSPs' results in the correct order instead of a sort on the entire result set.
- `KEY`: only one side in a partial partition-wise join is redistributed.
- `ROUND ROBIN`: rows are distributed to PSPs one at a time in a circular fashion. This is either the final redistribution operation before sending data to the requesting process, or one of the earlier steps when no redistribution constraints are needed.
- `RANDOM`: rows are randomly assigned to PSPs.
- `PARTITION`: partial partition-wise join, for which the first row source is distributed in accordance with the partitions of the second row source. By the way, a full partition-wise join, that is for equi-partitioned row sources, does not require redistribution.

On real application cluster (RAC) databases the `LOCAL` suffix indicates that rows are distributed to consumers on the same RAC node.

Watch out for `PX COORDINATOR FORCED SERIAL`, which means that the plan may look like Oracle prefers a parallel execution of the SQL statement but it doesn't really; when push comes to shove, Oracle picks a serial execution.

What is important to understand is that each data flow operation (DFO) in the execution plan can have its own degree of parallelism. The degree of parallelism for each DFO at execution time may be quite different from the one determined by the optimizer because of a downgrade in the degree of parallelism. Common causes are that the amount of CPUs available at execution time is different from what the optimizer assumed would be available, or that Oracle is unable to use similar amounts of PGA memory for the SQL areas because only a specific amount is allocated per PSP.

Even if at runtime the degree of parallelism is as expected, the parallel execution can only be efficient if the work is distributed evenly across the PSPs. During development, the view `v$pq_tqstat` can help you with identifying skew across table queues. Data skew or unequal partitions are the usual suspects, depending on the `PX ITERATOR` chosen by the query optimizer. An insidious reason for distribution skew is the `HASH` distribution method. The hashes are sometimes not uniformly distributed, generally because of an outer join for which all nulls are hashed to the same value, which leads to some PSPs receiving a larger-than-average share of data. As a consequence, the remainder of PSPs are idle most of the time.

Notes

2.2.2 Adaptive Query Optimization

An important new feature in Oracle Database 12c is adaptive query optimization, which consists of two components: adaptive plans and adaptive statistics. The optimizer's statistics collector has the ability to detect whether its cardinality estimates are different from the actual number of rows seen by the individual operations in the plan. If the difference is significant, then the plan or a at least a portion of it can be modified on the fly to avoid suboptimal performance of the initial execution of a SQL statement. Plans are also automatically re-optimized, which means that *after* the initial execution of a SQL statement and *before* the next execution of the same statement, Oracle checks whether its estimates were off, and if so determines an alternative plan based on corrected, stored statistics.

Statistics feedback allows re-optimization based on erroneous cardinality estimates discovered during the execution. Tables without statistics, queries with multiple filter predicates on a table, and queries with predicates that include complex operators are candidates for statistics feedback; if multiple filters on columns that are correlated are issued,

then the combined data can become skewed as compared to the original data, which is something the optimizer is not aware of before the execution of the statement.

You can see whether a statement can be re-optimized by querying `v$sql`: the column `is_reoptimizable` holds the information you seek. The next time you execute the statement a new plan will be generated, for which the flag `is_reoptimizable` will be `N`. Similarly, joins can be adapted at runtime; only nested loops can be swapped for hash joins and vice versa.

Oracle Database 12c also introduced another distribution method for the parallel execution of queries: `HYBRID HASH`. This allows the optimizer to defer its distribution method until execution at which point more up-to-date information on the number of rows involved can be used. Immediately in front of the `PX SEND HYBRID HASH` operation there is the work of the `STATISTICS COLLECTOR`. If the statistics collector discovers that the actual number of rows buffered is less the threshold of 2·DOP, the distribution method will go from `HASH` to `BROADCAST`. If it finds out that the actual number of rows buffered is more than the threshold, the distribution method will always be `HASH`. Re-optimization of parallelizable SQL statements is done with the performance feedback mechanism, which allows the optimizer to improve the degree of parallelism for repeated SQL statements; AutoDOP has to be enabled though.

Something you can see in the notes section of execution plans is whether a SQL plan directive was used. A SQL plan directive is automatically created when automatic re-optimization of query expressions takes place. It instructs the optimizer what to do when a certain query expression is encountered, for example to employ `DYNAMIC_SAMPLING` because of cardinality misestimates. Information about the SQL plan directive can be obtained from `dba_sql_plan_directives` and `dba_sql_plan_dir_objects`. When a certain expression is encountered by the optimizer, the SQL plan directive type listed (e.g. `DYNAMIC_SAMPLING`) is employed.

As in previous versions, `EXPLAIN PLAN` only returns the execution plan preferred by the optimizer. The function `DBMS_XPLAN.DISPLAY_CURSOR` shows the plan actually used by the query executor. To see all information, please use the following statement:

```
SELECT
  *
FROM
  table ( DBMS_XPLAN.DISPLAY_CURSOR ( FORMAT => '+adaptive' ) )
;
```

Note: When dynamic statistics is enabled (`ALTER SESSION SET OPTMIZER_DYNAMIC_SAMPLING = 11`)), the time it takes to parse a statement will go up. More information can be found on the Oracle's web page for the query optimizer.

2.3 Indexes

Imagine you're feeling nostalgic one day and want to look for a definition in a printed dictionary; you want to know whether 'crapulent' really is as funny as you think it is. What makes the dictionary so practical when it comes to finding words and phrases? Well, entries are listed from A to Z. In other words, they are ordered. All you need is the knowledge of the order of the letters in the alphabet. That's all.

It would be very inefficient if you had to read the entire dictionary until you spotted the single item you're interested in. In the worst case you could end up reading the entire 20 volumes of the Oxford English Dictionary, at which point you probably don't care about the assumed funniness of crapulence any longer.

The same goes for databases. To fetch and retrieve rows in database tables it makes sense to have a simple and fast way to look them up. That is what an index does. It is similar to the index in old-fashioned books: the entries are listed in alphabetical order in the appendix, but the page number to which the entries refer does not generally have any structure. That is, the physical order (when an entry is mentioned in the book) is independent of the logical order

(when an entry is listed in the index, from A to Z). In databases we typically do that with the aid of a doubly linked list on the index leaf nodes, so that each node refers to its predecessor and is successor; the leaf nodes are stored in a database block or page, which smallest available storage unit in the database. This data structure makes it easy to run through the list in either direction.

The dictionary we mentioned earlier is an example of an index-organized table (IOT) in Oracle parlance; Microsoft SQL Server calls these objects clustered indexes. The entire table, or dictionary in our example, is ordered alphabetically. As you can imagine, index-organized tables can be useful for read-only lookup tables. For data sets that change frequently, the time needed to insert, update, and/or delete entries can be significant, so that IOTs are generally not recommended.

Where an index leaf node is stored is completely independent of its logical position in the index. Consequently, a database requires a second data structure to sift quickly through the garbled blocks: a balanced search tree, which is also known as a B-tree. The branch nodes of a B-tree correspond to the largest values of the leaf nodes.

When a database does an index lookup, this is what happens:

1. The B-tree is traversed from the root node to the branch (or header) nodes to find the pointers to relevant leaf node(s);
2. The leaf node chain is followed to obtain pointers to relevant source rows;
3. The data is retrieved from the table.

The first step, the tree traversal, has an upper bound, the index depth. All that is stored in the branch nodes are pointers to the leaf blocks and the index values stored in the leaf blocks. Databases can therefore support hundreds of leaves per branch node, making the B-tree traversal very efficient; the index depth is typically not larger than 5. Steps 2 and 3 may require the database to access many blocks. These steps can therefore take a considerable amount of time.

Oh, and in case you are still wondering: crapulent isn't that funny at all.

2.3.1 Developer or Admin?

Indexes can speed up lookups but having too many indexes causes serious performance degradations when inserting, updating, and deleting. The reason is simple: the database has to maintain the index and the data structures associated with it. As the index grows, branch and leaf nodes may have to be split, which obviously gobbles up valuable CPU time.

Horrible advice you'll sometimes encounter in your life as a database developer is that a DBA is responsible for indexes. Absolutely not! The performance of a `SELECT` depends on indexes, and the existence of indexes on a table affects `INSERT`, `UPDATE`, and `DELETE` statements. Only a developer knows what queries are typically run, how often a table's data is modified, how it is modified (i.e. single rows or in bulk, normal or direct-path inserts, ...) so only a developer can judge whether an index on a particular combination of columns makes sense.

Knowledge of indexes is a must for every database developer. A magnificent reference is Markus Winand's SQL Performance Explained. If it's not in your library, you're not yet serious about databases! For the more frugal among us, his website on indexes is also a valuable resource that covers a lot of what's in the book.

2.3.2 Access Paths and Indexes

Let's take another quick look at the access paths we talked about *earlier*.

The **index unique scan** only returns one row because of a `UNIQUE` constraint. Because of that, Oracle performs only the tree traversal: it goes from the branch node to the relevant leaf node and picks the source row it needs from the table.

For an **index range scan** the tree is traversed but Oracle also needs to follow the leaf node chain to find all remaining matching entries. It could be that the next leaf node contains more rows of interest, for instance if you require the

maximum value of the current leaf node; because only the maximum value of each leaf block is stored in the branch nodes, it is possible that the current leaf block's maximum index value 'spills over' to the next. A **table access by index ROWID** often follows an index range scan. When there are many rows to fetch this combination can become a performance bottleneck, especially when many database blocks are involved. The cost the optimizer calculates for the table access by index ROWID is strongly influenced by the row count estimates.

As the name suggest, a **full index scan** reads the entire index in order. Similarly, a **fast full index scan** reads the entire index as stored on disk. It is the counterpart of the **full table access scan**, and it too can benefit from multi-block reads.

2.3.3 Statistics

Contrary to what some people may have heard, a *balanced* search tree is, as the name suggests, *always* — read that again, please — always balanced. It is a myth that you have to rebuild the index whenever the performance of your queries is below par. There are extremely rare cases when Oracle recommends that you rebuild the indexes but in almost all cases you do not have to rebuild your indexes.[1]

Oracle nowadays automatically collects statistics, so once you create an index, Oracle takes care of the rest. You can see the schedule and some basic information about the statistics collection with the following statement:

```
SELECT
  *
FROM
  dba_autotask_client
WHERE
  client_name = 'auto optimizer stats collection'
;
```

For most databases the automatic statistics collection is sufficient. If, however, your database has tables that are being deleted and truncated between collection runs, then it can make sense to go gather optimizer statistics manually.

When you create an index, Oracle automatically gathers optimizer statistics because it needs to do a full scan anyway. As of Oracle Database 12c, the same piggybacking is done for the statistics collection of create-table-as-select (CTAS) and insert-as-select (IAS) operations, which is quite nifty; histograms require additional data scans, so these are not automatically gathered. The execution plans of CTAS and IAS statements show whether statistics are being collected at runtime: `OPTIMIZER STATISTICS GATHERING`, right below the `LOAD AS SELECT` operation.

If you change the definition of an index, you may want to update the statistics. Please coordinate with the DBA to avoid unwanted side effects, such as degrading the performance of all but your own queries because invalidation of execution plans; gathering statistics does not lock the table, it's like running multiple queries against it.[2]

Notes

2.3.4 Predicates: Equality before Inequality

An index can be beneficial to your queries' performance when there is some sort of filtering that can be handled efficiently by the index. The performance is intimately related to the `WHERE` clause and the existence (or absence) of indexes on columns specified in the `WHERE` clause. As such, the `INSERT` statement is the only one of the unholy

[1] The index clustering factor indicates the correlation between the index order and the table order; the optimizer takes the clustering factor into account for the `TABLE ACCESS BY INDEX ROWID` operation. A high ratio of leaf nodes marked for deletion to leaf nodes (> 0.20), a low value of percentage used (< 0.80), and a clustering factor (see `DBA_INDEXES`) close to the number of rows (instead of the number of blocks) in the table (see `DBA_SEGMENTS`) are indicators that your indexes may benefit from rebuilding. If the clustering index is close to the number of rows, then the rows are ordered randomly.

[2] The `DBMS_STATS.AUTO_INVALIDATE` option can be used to ensure that Oracle does not invalidate all cursors immediately, which can cause a significant CPU spike. Instead, Oracle uses a rolling cursor invalidation based on internal heuristics.

insert-update-delete (*IUD*) trinity that can never benefit from an index: it has no `WHERE` clause. With an `UPDATE` or `DELETE` you typically have predicates, so they can benefit from fast lookups, even though the maintenance of the index negatively affects the performance; it is a trade-off that has to be evaluated carefully by the developer. In data warehouse environments it is not uncommon to drop all indexes temporarily and re-create them once the data loaders have completed their business. Alternatively, you can make your index unusable (i.e. `ALTER INDEX index_name UNUSABLE`) and once you have imported the data, rebuild it: `ALTER INDEX index_name REBUILD`. Only for function-based indexes you can `DISABLE` and `ENABLE` the index.

Predicates show up in execution plans as access, index filter, or table-level filter predicates. An access predicate corresponds to the start and stop conditions of the leaf node traversal. During the leaf node traversal index filters can be applied. In case you filter on a column that is not in the index, the filter predicate is evaluated on the level of the table.

It is important that you understand the differences in predicates, because it is critical to the index range scan access method. Access predicates limit the range of the index to be scanned, whereas index filters are applied to the results of the scanned range. This is typically the performance bottleneck for index range scans; if the requested rows are all stored in the same block, then it may not be a problem because Oracle can use a single read operation to fetch the data. However, you want to keep the scanned index range as small as possible, because the leaf node traversal can take a while.

Because index maintenance affects the performance of insert, update, and delete (IUD) statements, it is important that you create enough indexes but no more than needed. The fewer indexes a table has, the better its IUD performance; the fewer indexes the execution of a query uses, the better its performance. The question thus is how to optimally define indexes, so that the performance of queries is acceptable while at the same time the performance of IUD statements is not abominable.

Primary key and unique constraints automatically generate database indexes. Sometimes these constraints are sufficient but sometimes you need more.

If you only have one column that you ever use to look for data, then that column is your index. Suppose your application requires an ISBN to return the title, the first and last name of the main author, purely as information and nothing else. It can then make sense to include these three columns in the index too, not because you filter on them — because we just said you don't — but to make sure that Oracle can avoid a trip to the table: Oracle can simply read the index and retrieve the information it needs to provide from it. Such retrieval logic is generally coded in PL/SQL functions that live inside the database. We shall talk more about functions when we talk about PL/SQL, but we want to mention two advantages of this approach to whet your appetite: when using *bind variables*, the execution plan is stored irrespective of the ISBN provided, so Oracle does not need to ask the optimizer again and again, and you can take advantage of the *result cache*, which means that information on popular books is cached and available without having to check the index or fetch data from the table.

Anyway, when you include multiple columns in an index, that is you create a compound index, the order of the columns is very important. The difference between a column being used as a filter rather than access predicate can be significant. Because of that, it is recommended to index for equality first and then for ranges, typically `DATE`-like columns, so that the extent of the leaf node traversal can be kept to a minimum.

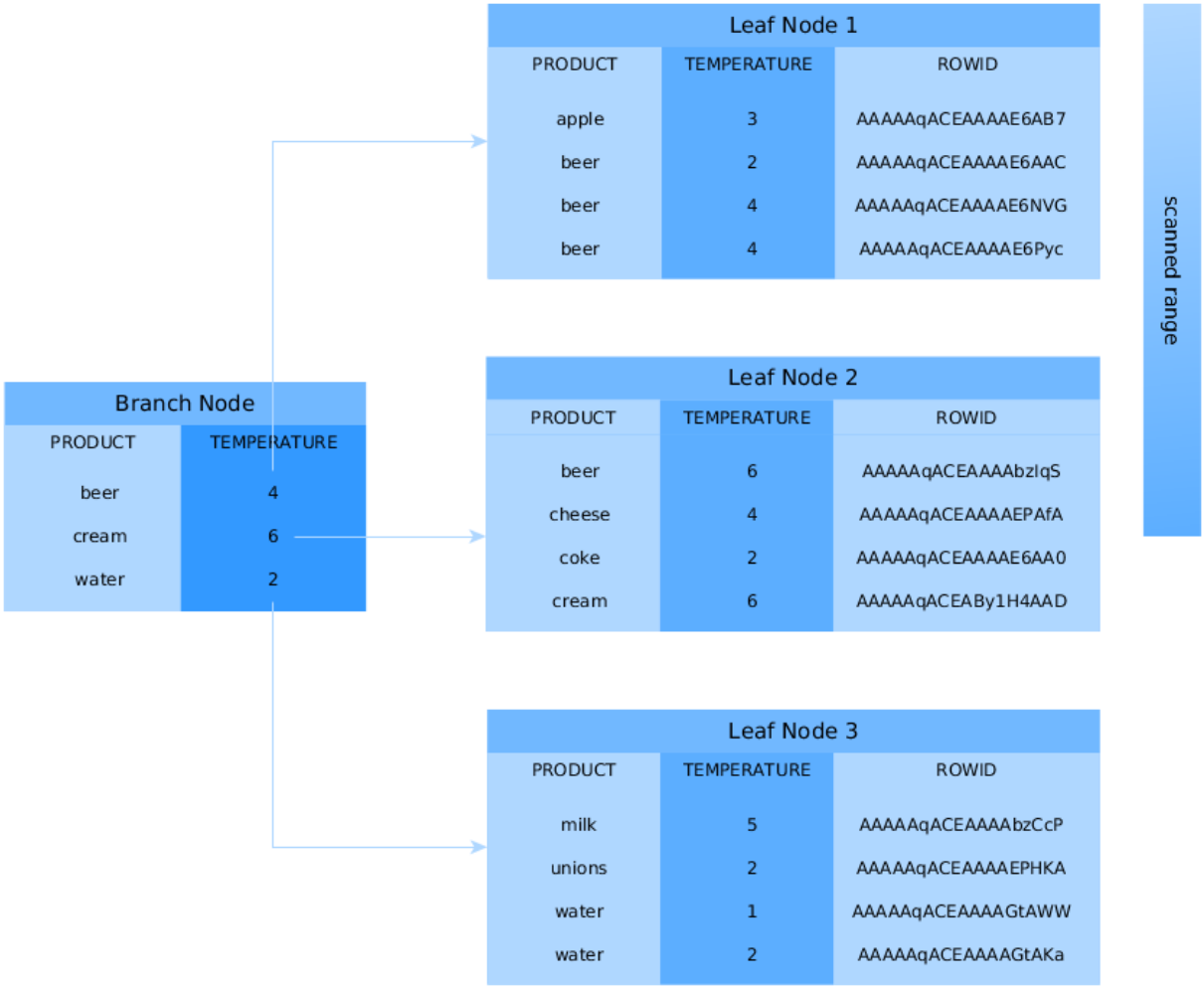
Let's take another look at our *friendly household robot*. We wanted beer chilled below five degrees. The time we have to wait for the robot to fetch the items from the fridge depends on how fast our electronic companion can locate the items requested. An index is of course beneficial but the question remains what we want to index first: temperature or product. Because we rarely ever need an item at an exact temperature, that is we want items in a particular temperature range, it makes sense to index on product first and then on temperature.

'What's the difference?' you ask? Well, let's find out.

When the index is on temperature first and then product, the robot checks one of the branch nodes and sees that there are *at least* two leaf nodes with the correct temperature; the temperature is the access predicate. It follows the pointers to these leaf nodes and uses the product column as a filter predicate. Because one of the leaf nodes' maximum value for temperature is 4, we have to follow the leaf node chain because the next leaf node may contain more products with a temperature below five degrees. And yes, there is one more item; it's not what we were looking for but our robot


Leaf Node 1
TEMPERATURE
PRODUCT
ROWID
1
water
AAAAAqACEAAAAGtAWW
2
beer
AAAAAqACEAAAAE6AAC
2
coke
AAAAAqACEAAAAE6AA0
2
water
AAAAAqACEAAAAGtAKa
Leaf Node 2
TEMPERATURE
PRODUCT
ROWID
2
unions
AAAAAqACEAAAAEPHKA
3
apple
AAAAAqACEAAAAE6AB7
4
beer
AAAAAqACEAAAAE6Pyc
4
beer
AAAAAqACEAAAAE6NVG
Leaf Node 3
TEMPERATURE
PRODUCT
ROWID
4
cheese
AAAAAqACEAAAAEPAfA
5
milk
AAAAAqACEAAAAbzCcP
6
beer
AAAAAqACEAAAAbzIqS
6
cream
AAAAAqACEABy1H4AAD
Branch Node
TEMPERATURE
PRODUCT
2
water
4
beer
6
cream
scanned range

could not know that when browsing the index.



Assume the product is index first and the temperature second, as recommended. What's the advantage? Well, the index tells us exactly in which leaf nodes to look for beer: it's in the ones before cream because `'apple' < 'beer' < 'cream'`. We still have the case that we have to follow the leaf node chain because `'beer'` happens to be the last product in one of the index leaf nodes.

Note that the advice about equality-range indexes is not the same as saying that the most selective index in our case should go first. Suppose, for the sake of argument, that each product in our table has a different temperature. Depending on the temperature of the warmest item, we could have to scan the entire table, if all products have been in the fridge at least a few hours, so that they are all below five degrees, or we could have no items to consider at all because someone forgot to close the fridge door and everything is almost at room temperature. Sure, in the latter case an index on the temperature first would yield the best performance. But: in the former case the performance could potentially be horrible, especially if the number of products in the fridge were to increase significantly.

Another scenario to ponder about: all products in the fridge are unique. For the product-temperature index we would look up the product and then verify whether its temperature is in the specified range. Simple. Quick. For the temperature-product index, however, we could potentially have to scan everything and then filter out the items that are not beer.

Yet another scenario: all products in the fridge are beer — man, you're thirsty! The product-temperature index requires

us to do a full fridge scan and take only the bottles below five degrees. The temperature-product index is obviously more efficient because it can use the temperature as an access predicate; the filter predicate on `'beer'` is pretty much useless, as is the index.

As you can see, the performance of an equality-range index is more consistent and thus more production-suitable than the range-equality index. There are of course cases when the range-equality index is more appropriate: skewed data, where you *know* it is more advantageous, but you have to be absolutely sure the skewness stays the same. For most databases that is a bit too iffy to be useful advice.

Another reason why the equality-range index is a good rule of thumb is that whatever is searched for with an equality predicate is something that is pretty much standard to all your queries: you primarily want certain stuff from the fridge, where the temperature is only secondary. "I'm starving for some 7-degrees' produce," is not something you often hear people say when they're thirsty but have no cool liquids available; you might want to gobble up some cool broccoli instead but I doubt it.

If our fridge table is equipped with expiry dates, that column would also be included as a second or third column. We're typically interested in items that have not yet expired (`expiry_date <= SYSDATE`), or, if we want to have the robot clean up the fridge, all items that have already expired. Whether the temperature or expiry date should go first *after* the product depends a bit on the situation: do you search more frequently for the expiry date or the temperature of items in the fridge?

Anyway, when you need an index on additional columns, add these to the index you already have or redefine it. An extra index may not provide you with the benefits you expect: the optimizer has to combine two indexes when executing your queries, and the database has to maintain two indexes. The fewer indexes the optimizer has to use, the better the performance of your queries. More than 5 indexes is usually not recommended, but the exact number may well depend on the specifics of your environment. Nevertheless, if you are really looking at five or more indexes for a particular table, you have to think about why you need so many separate indexes, and document your reasons carefully.

With regard to SQL statements, always be as specific as possible. Suppose you go to the trouble of adding manufacturers of products in your fridge, you create a compound manufacturer-product index, and let the legged circuit board look for some `'Coke Zero'` by `'The Coca-Cola Company'`. Sure, `'Coke Zero'` is only made by one company, but today you're too tired, so you simply write `WHERE product = 'Coke Zero'`. If you're lucky, the robot decides to do a skip scan on the leading edge of the index; if you are less fortunate, and your fortune depends mainly on the histogram of the leading index column (i.e. the manufacturer column) , your robot may decide on a full fridge scan. Oracle does not know about correlations in your data, so if you want Oracle to come back with your rows as quickly as possible, provide all the details possible that aid it in its search for your data. If at all possible, always include your leading index column in all your queries' predicates. It is advice given to mathematicians and, likewise, applies to (database) developers: do not assume anything unless stated otherwise.

2.3.5 Predicates: LHS vs RHS

They way you *syntactically* write predicates matters, even though *logically* various forms of predicates are equal. The difference between the left-hand side (LHS) and the right-hand side (RHS) of equality and inequality predicates is significant. Oracle transforms a predicate such as $A \star B$, with $\star$ any valid operator (e.g. `=`, `>`, or `LIKE`) to a canonical form, where typically the LHS contains only the (indexed) column name. The most this transformation does is swap the operands. The exception to this rule is when both sides of the operator are non-standard, for instance a function is applied or numbers are added. That Oracle transforms the predicate `SYSDATE - 1 <= col_name` to `col_name >= SYSDATE - 1` can easily be verified by looking at the execution plan. When it encounters something like `col_name + 2 = 4*another_col_name`, it is not smart enough to rewrite this as `col_name = 4*another_col_name - 2`, so you have to do it for Oracle.

What about predicates that emulate full-text searches like `WHERE col_name LIKE '%something interesting%'`? Short answer: you're pretty much screwed. Standard indexes are not designed to meet that requirement. It's like asking you to search for a book with an ISBN that has 123 somewhere in it. Good luck! Long answer: Oracle Text. Yes, it's the long answer, even though it's only two words, because it requires you to do some

digging. Oracle Text comes with all editions of the database but it's beyond our scope. With it you can use SQL to do full-text searches, which is especially interesting you need to mine texts; it's overkill for occasional queries with a non-leading `LIKE` though.

2.3.6 Function-Based Indexes and NULLs

By default Oracle does not store null rows in a (B-tree) index. You can add them with a simple trick:

```
1 CREATE INDEX index_name
2   ON tab_name ( nullable_col_name, 1 );
```

The 'trick' is of course nothing but a function-based index. By adding nulls to your (function-based) index, you ensure that Oracle is able to avoid full table scans when you ask for `col_name IS NULL`. Alternatively, you can use `NVL` as a function in your index if you want to; you have to remember that your index can only be used if you use the same function in your filter predicates.

That is a common thread in function-based indexes though: you have to have the exact same predicate as used in the index for the index to be used. Oracle has no compiler that evaluates and simplifies (mathematical) expressions, so a `WHERE` clause like `WHERE col_a + col_b = 42` does not use an index on `col_a` because the lef-hand side also includes `col_b`. To use an index on `col_a` , you have to rewrite the predicate as `WHERE col_a = 42 - col_b`. Similarly, `LN ( EXP(col_real) )` is not simplified to `col_real` for `col_real` a column of real-valued numbers. Oracle is smart but you cannot expect it to do everything for you: not even state-of-the-art computer algebra systems like Mathematica and Maple can simplify all crazy expressions you can think of.

The power of function-based indexes lies in the fact that often your applications have to filter for bits and pieces of data that are already available but normal indexes cannot be used, which often happens because of conversion, mathematical, and string-manipulation functions, in particular `SUBSTR()` and `LOWER()` or `UPPER()`. Suppose you have a sudden, inexplicable urge to behave like a business analyst and you want to generate a report of the average temperature of all products with an expiry date of products in your fridge for a particular ISO workweek; if you think this is an odd request then please replace temperature with amount, expiry date with the date of the transaction, and the fridge with a sales table.

You create the following function-based index: `CREATE INDEX ix_workweek ON fridge ( TO_CHAR(expiry_date, 'IW') )`. If you now use a clause like `WHERE TO_CHAR(expiry_date, 'IW') = '20'`, you can see all products with an expiry date in workweek twenty using the index `ix_workweek`; the single quotes in the `WHERE` clause are included because the resulting expression is of type `VARCHAR2`. Avoid implicit conversions as much as possible; not because of the almost negligible conversion performance penalty but because you rely on Oracle to do it right in the background *and* it is considered bad style!

Imagine you have created a function-based index on a certain concatenation of columns, for instance `manufacturer || '''s ' || product`, then you can use that exact expression in the `WHERE` clause. This is, however, an extremely fragile and overly complex solution that does not really belong in your queries. Such logic is usually application-specific, so it should either be handled by the application itself or a layer of PL/SQL between the database and your user interface that extracts the data and then with the aid of an auxiliary function formats it correctly: it is always a good idea to separate the data-access layer from the application layer, as it minimizes the number of places you have to search and replace something whenever a change requests ends up on your desk.

Why? What if next week someone decides to search for `WHERE product || ' by ' || manufacturer = ...` instead? You then need to change not only your query but also your index. Worse still, what if you want to list only stuff from one manufacturer? You can't even use the index! Why make life hard when you can just add both `manufacturer` and `product` to your index and search for each one individually, separated by a beautiful `AND`?! If at this point you think that no one is that thick, then I'll just say that if I'd have had a dollar for each time I saw something similar (usually with first and last names), I'd be rich. And if I'd have had an extra dollar for each time people complained about shitty performance because of such an abomination of a predicate and demanded an index to solve it, I'd be stinking rich.

By the way, we're not done with nulls yet. Queries can sometimes run without utilizing an index because a `NOT NULL` constraint is absent. Constrains are thus not only important to enforce consistency but also to ensure consistent performance of your queries. Furthermore, functions on columns *with* `NOT NULL` constraints can lead to the same (unwanted) behaviour. The reason is that Oracle does not know whether a function preserves the `NOT NULL` constraint of the column. For some internal functions, though, Oracle knows that `NOT NULL` is preserved, which means that it can still use any available and relevant indexes. Examples of such internal functions are `LOWER()` and `UPPER()`.

User-defined functions are black boxes as far as the optimizer is concerned. As of Oracle Database 11g, virtual columns can be used to circumvent the issue. Virtual columns are not stored and they are derived (or computed) from other columns in the same table. They are created like normal columns but with the syntax of `col_name [ data_type ] [ GENERATED ALWAYS ] AS ( expression ) [ VIRTUAL ]`, where the entries between square brackets are optional although highly recommended to indicate that the column in question is merely virtual. An index on a virtual column is like a function-based index on a normal column, but it has the benefit that you can add the `NOT NULL` constraint to it. Hence, the optimizer can treat the expression as a `NOT NULL`-preserving function. Sweet!

2.3.7 Predicates: The `WHERE` Clause

The `WHERE` clause is the one that determines whether or not indexes can be used efficiently. One side of each predicate must be as specified in the index(es) for Oracle to be able to use any index. Whether it is the left-hand side or the right-hand side is irrelevant, although typically it is the left-hand side because SQL is written from the left to the right. Note that the order sometimes matters though: `col_name LIKE 'ABC%'` is not the same as `'ABC%' LIKE col_name`. The former searches for `col_name` entries that begin with `ABC`, whereas the latter is the same as the filter `col_name = 'ABC%'`, that is the `%` is not interpreted as a wild card at all.

Indexes can only be used when predicates are *sargable*, or search-argument-able, which admittedly is a horrible phrase. Functions on columns in the index can prohibit index use, particularly when the index is not a function-based index. Apart from that, some operators are sargable and optimizable (i.e. allow the use of an index): `=`, `<`, `>`, `>= IS NULL`; some operators are sargable yet not optimizable: `<>` and its equivalents (i.e. `!=` and `^=`) and `NOT`; and `LIKE` with a leading wild card is not sargable and hence not optimizable. Sargable predicates can be pushed down, which means that a predicate in a statement that references a view or derived table can be 'pushed down' to the view or derived table itself, which avoids having to scan the entire underlying data structure only to filter out a few relevant rows later. Sargable, non-optimizable predicates can still benefit from the optimizer's efforts; non-sargable predicates cannot though.

A SQL statement that links several sargable predicates with an `OR` cannot be optimized when the predicates involve different columns. If, however, the predicates can be rewritten as an equivalent `IN`-list, which Oracle does internally as a part of its predicate transformations, then Oracle can indeed optimize the statement and therefore utilize existing indexes.

Important is, as always, that the data type of each search term matches the data type of the indexed column or expression; it is best that you convert search terms on the right-hand side if necessary but leave the left-hand side as is. Unnecessary use of `TO_CHAR()` and `TO_NUMBER()` (on the left-hand side) is not only sloppy but it can hamper index use. The same goes for `NVL()` and the like.

If you often encounter fixed expressions or formulas in your predicates, you can create function-based indexes on these expressions. Make sure that the columns referenced appear in the index in exactly the same way as they appear in the predicates, *and* make sure that the right-hand side does not contain the columns from the index: Oracle does not solve your equations for you.

Predicates that are often badly coded include operations on dates. Yes, it is possible to create a function-based index on `TRUNC ( expiry_date )` and use same expression in the database. However, *all* predicates on the column `expiry_date` *must* include `TRUNC()` for Oracle to be able to use the index in all cases. A simple and elegant solution is to provide ranges, either with `>= TO_DATE(...)` and `<= TO_DATE(...)` or with `BETWEEN TO_DATE(...) AND TO_DATE`, which is inclusive. Should you not want it to be inclusive subtract a minimal interval like so: `TO_DATE(...) - INTERVAL '1' SECOND'`.

Why not the literal `1/86400` or `1/(24*60*60)`? Well, it may be easy for you to understand something like that because you wrote it (and perhaps added a comment), but it is not always easy to fathom such literals, especially if developers simplify their fractions as in `7/10800`, which is 56 seconds by the way. The index may not care about how you write your literals but the other developers in the team do care. Let the code speak for itself!

Since we're on the topic of dates: *never* write `TO_CHAR ( expiry_date, 'YYYY-MM-DD' ) = '2014-01-01'`. Leave the `DATE` column as is and write `expiry_date >= TO_DATE ( '2014-01-01', 'YYYY-MM-DD' )` and `expiry_date < TO_DATE ( '2014-01-01','YYYY-MM-DD' ) + INTERVAL '1' DAY` instead.[1] Yes, it's a bit more typing, but that way an index range scan can be performed and you do not need a function-based index.

'But what if I need only products from the fridge that expire in February?' Since repetition is the mother of learning, here comes: specify ranges from the first day of February to the last day of February.

'But I want to show the total number of products by the year and month of the expiry date.' You could use the `EXTRACT ( YEAR FROM expiry_date )` and similarly for the month, `TRUNC( expiry_date, 'MM' )` or `TO_CHAR ( expiry_date, 'YYYY-MM' )`. However, since you are pulling in all data from the table, a full table scan really is your best option. Yes, you read that right: a full table scan is the best alternative; we'll say more about full table scans in a few moments. Furthermore, if you already have an index on `expiry_date` and it is stored in order (i.e. it is not a `HASH` index on a partitioned table), then the `GROUP BY` can make use of the index without any additional function-based indexes.

The `LIKE` comparison operator is also often a cause for performance problems because applications tend to allow wild cards in strings, which means that a search condition à la `WHERE col_name LIKE '%SOMETHING%'` is not uncommon. Obviously, you cannot create a sensible index for a predicate like that. It is tantamount to asking a dictionary to provide you with a list of all possible sequences of characters in any position.

The `INDEX` hint, as described by Laurent Schneider, is — contrary to what is claimed by the said author — *not* always beneficial for predicates with leading and trailing wild cards, so be sure to try it out. An index is, however, used when such a predicate is specified with bind variables:

```
1 VARIABLE l_like VARCHAR2(20);
2 EXEC :l_like := '%SOMETHING%';
3
4 SELECT
5   *
6 FROM
7   tab_name
8 WHERE
9   col_name LIKE :l_like;
```

If you always look for things *ending* with a series of characters, such as `LIKE '%ABC'` you *can* use an index. Just create the index on `REVERSE ( col_name )` and reverse the string you are looking for itself, and voilà, it works: `WHERE REVERSE ( col_name ) LIKE 'CBA%'`.

To search in a case-insensitive manner you have to create a function-based index, say, `UPPER(col_name)`. You could have gone with `LOWER(col_name)` and whatever you prefer is really up to you. All that matters is that you are thrifty and consistent: switching back and forth between `UPPER()` and `LOWER()` is a bad idea because the database has to maintain two indexes instead of one, and you really only need one index. Which function you choose for case-insensitive searches is irrelevant but document whichever you choose, so it is clear to all developers on the team.

In an international setting you may want to use `NLS_UPPER( col_name, 'NLS_SORT = ...' )`, so that for instance — for `... = XGERMAN` — ß and `ss` are seen as equivalent. The parameters `NLS_SORT` and `NLS_COMP`

[1] The `INTERVAL` function has one major disadvantage: `SELECT TO_DATE ( '2014-01-31', 'YYYY-MM-DD' ) + INTERVAL '1' MONTH FROM dual` leads `ORA-01839: date not valid for month specified` error. The function `ADD_MONTHS()` solves that problem.

can be made case- or accent-insensitive by appending `_CI` or `_AI` to their sort name values respectively. The `NLS_SORT` parameter can be used to alter a session or the entire system.

For purely linguistic rather than binary searches of text, you can set the system's or session's `NLS_COMP = LINGUISTIC`. The performance of linguistic indexes can thus be improved: `CREATE INDEX ix_col_name_ling on tab_name ( NLSSORT( col_name, 'NLS_SORT = FRENCH' ) )`, for French for example.

We have already seen that with function-based indexes it is important to have the exact same expression save for irrelevant spaces. A functionally equivalent expression that is syntactically different prohibits the use of an index, so writing `REGEXP_LIKE()` in your `WHERE` clause when you have used `INSTR()` in the index means that the optimizer will ignore the index.

For Oracle Database 11g there is a good book on expert indexing, if you want to learn more about indexes.

Notes

2.3.8 Full Table Scans

Full table scans are often seen as a database's last resort: you only do them if you absolutely have to. That reputation of full table scans is not entirely warranted though.

For small tables it often does not make sense for Oracle to read the associated index, search for the relevant ROWIDs, and then fetch the data from the database tables when it can just as easily do a single round trip to the table. Thanks to multi-block I/O in full table scans a couple of parallel round trips are also possible to speed up the process.

Analogously, when the database has to return a sizeable portion of all the rows from a table, the index lookup is an overhead that does not always pay off. It can even make the database jump back and forth between blocks.

Full table scans frequently indicate that there is optimization potential but remember, as originally noted by Tom Kyte: "full table scans are not evil, indexes are not good".

2.3.9 Top-N Queries and Pagination

Top-N and pagination queries frequently pop up in applications: a user is only shown the top-N entries or allowed to flip back and forth between pages of data. Prior to Oracle Database 12c there were a couple of roundabout methods available to do pagination: offset, seek, window or analytical functions.

The `OFFSET/FETCH` or row-limiting clause has greatly simplified life for developers:

```
SELECT
   manufacturer
 , product
 , temperature
 , expiry_date
FROM
   fridge
ORDER BY
   expiry_date
OFFSET 5 ROWS
FETCH NEXT 10 [ PERCENT ] ROWS ONLY
;
```

An issue that is often overlooked when it comes to the row-limiting clause is explained on Markus Winand's Use The Index, Luke page. We'll briefly cover the salient details, as it affects application and database performance. Suppose your users flip through pages of data and are allowed to insert rows at any position. The `OFFSET` clause can cause rows to show up twice: once on the previous page *before* the row was inserted and once on the current page *after* the

row was inserted (on the previous page). Furthermore, `OFFSET` is implemented in a way that data below the `OFFSET` line needs to be fetched and sorted anyway.

The solution to this conundrum is quite simple: keyset pagination: use the `FETCH` clause as before but replace the `OFFSET` clause with a `WHERE` clause that limits the result set to all rows whose key is before or after the identifier (key) of the row previously displayed. Whether you have to take > or < depends on how you sort and what direction the pagination runs in of course. An index on the columns in your `WHERE` clause, including the key, to aid the `ORDER BY` means that browsing back to previous pages does not slow your users down.

With that in mind we can rewrite our query:

```
SELECT
   manufacturer
 , product
 , temperature
 , expiry_date
FROM
   fridge
WHERE
   expiry_date < last_expiry_date_of_previous_page
ORDER BY
   expiry_date
FETCH NEXT 10 [ PERCENT ] ROWS ONLY
;
```

Two major bummers of keyset pagination are that 1) you cannot jump to arbitrary pages because you need the values from the previous page and 2) no convenient bidirectional navigation is available because that would require you to reverse the `ORDER BY` and key comparison operator.

2.3.10 Index-Organized Tables

Index-organized tables are generally narrow lookup tables. They have to be narrow because all columns of the table are in the index. In fact, the index is the table itself.

It is technically possible to add additional indexes to an index-organized table. However, accessing an index-organized table via a secondary index is very inefficient. The reason is that the secondary index cannot have pointers to rows in the table because that would require the data to stay where it is. Forever. Because the data is organized by the primary index in an index-organized table, it can move around whenever data in modified. Secondary indexes store logical instead of physical ROWIDs; logical ROWIDs (`UROWID`) contain physical guesses, which identify the block of the row at the time when the secondary index was created or rebuilt. Standard heap tables are generally best for tables that require multiple indexes.

Index-organized tables can be beneficial to OLTP applications where fast primary-key access is essential; inserts typically take longer for index-organized tables. Because the table is sorted by the primary key, duplication of data structures is avoided, reducing storage requirements. Key compression, which breaks an index key into a prefix and suffix, of which the former can be shared among the suffix entries within an index block, reduces disk storage requirements even further.

Index-organized tables cannot contain virtual columns.

2.3.11 Beyond B-Trees: Bitmap Indexes

For columns with low cardinality the classical B-tree index is not an optimal solution, at least not in DSS or OLAP environments. Bitmap indexes to the rescue!

Bitmap indexes use compression techniques, which means that many ROWIDs can be generated with very little I/O. As argued by Vivek Sharma and Richard Foote, a bitmap index is not only your go-to index for low-cardinality columns

but also for any data that does not change frequently, for instance fact tables in data warehouses. Nevertheless, concurrent IUD operations clearly tip the scales in favour of standard B-tree indexes; bitmap indexes are problematic for online applications with many concurrent DML statements because of deadlocks and the overhead to maintain bitmap indexes.

Ad hoc queries are also generally handled better by bitmap than B-tree indexes. Queries with `AND` and `OR` can be executed efficiently because bitmap indexes on non-selective columns can be combined easily; `COUNT` queries are handled particularly efficiently by bitmap indexes. If users query many different combinations of columns on a particular table, a B-tree index has no real candidate for the leading index column. A bitmap index on all columns typically queried by analysts allows the index to be used for all these queries. It does not matter whether your business users use only one, two, three, or all columns from the index in their queries. In addition, nulls are included in the bitmap index, so you don't have to resort to function-based indexes.

By the way, you *can* create bitmap indexes on index-organized tables. More information on default B-tree and other indexes is of course provided by Oracle.

2.4 Subqueries

Subqueries come in different shapes and forms: scalar subqueries, nested (single- or multi-row) subqueries (in the `WHERE` clause), correlated subqueries, inline views (in the `FROM` clause), and factored subqueries (in the `WITH` clause), which are also known as common table expressions (CTEs) outside Oracleland. They are pretty versatile constructs as they can appear in almost any clause of DML statements with the exception of the `GROUP BY` clause.

Many of the subquery types are interchangeable. Inline subqueries can be rewritten as factored subqueries, and factored subqueries can often be rewritten as inline subqueries; recursive factored subqueries are the exception to this rule. Recursive factored subqueries can, nevertheless, typically be written as hierarchical queries using the `CONNECT BY ... START WITH ...` syntax. Similarly, you can write a correlated subquery as a join, but not every join can become a correlated subquery.

2.4.1 Scalar Subqueries

Let's start with the conceptually easiest type of the subqueries: scalar subqueries. Usually a scalar subquery in the `SELECT` needs to be evaluated for each row of the outer query (i.e. the part without the scalar subquery). If the scalar subquery calculates, say, a sum of a certain values from a large table, this means that the sum has to be calculated many times. Since it makes no sense to scan the same table over and over again for a value that really only needs to be computed once, Oracle Database 12c has the ability to unnest the scalar subquery, which means that it can convert the scalar subquery into, for instance, a `GROUP BY` view that is (outer-)joined to the table without the scalar subquery.

Tanel Poder has written about this feature, and his advice on the matter is spot-on too: you rarely need scalar subqueries in the `SELECT` clause and rewriting your scalar subqueries as joins opens up a whole collection of optimization options that Oracle can tinker with in the background. Oracle has had the ability to unnest scalar subqueries prior to 12c, but there was still a bug associated with scalar subquery unnesting until 11.2.

2.4.2 Nested and Correlated Subqueries

Nested and correlated subqueries show up in the `WHERE` clause of a SQL statement. Whereas a scalar subquery returns one row and one column, a single-row subquery returns one row but multiple columns, and a multi-row subquery returns multiple rows and multiple columns. Whenever the subquery does not reference columns from the outer query, we speak of a nested subquery, otherwise it is called a correlated subquery.

For multi-row nested subqueries it is important to note that the `ANY`, `ALL`, and `SOME` operators can sometimes be equivalent to `IN`-lists, which is why they do not often show up in production code even though Oracle loves them at certification exams. For instance, `WHERE col_name = ANY ( ... )` is equivalent to `WHERE col_name IN`

`( ... )`, and `WHERE col_name <> ALL ( ... )` is exactly the same as `WHERE col_name NOT IN ( ... )`, where the ellipsis indicates any valid, nested multi-row subquery. In fact, Oracle already does some of these transformations (and more) automatically.

Indexes are primarily used in the filters of the `WHERE` clause, as we have discussed before. This includes predicates with nested or correlated subqueries too. As such it is often advantageous to rewrite `NOT EXISTS` (anti-join) as `EXISTS` (semi-join) because it allows Oracle to use an index.

Related to the topic of semi-joins is whether there is any difference among the following three options that are commonly found in code:

- `WHERE EXISTS (SELECT * FROM tab_name ...)`,
- `WHERE EXISTS (SELECT 1 FROM tab_name ...)`,
- `WHERE EXISTS (SELECT col_name FROM tab_name ...)`.

The short answer: no.

The long answer is that the cardinality estimates may be slightly different, but in general the optimizer still chooses the same execution plan, for these differences are rarely the cause for the optimizer to think differently. The cost *is* affected by the cardinality estimates, but these are likely to be close together if the statistics for `tab_name` are representative. Oracle stops processing as soon as it obtains the first matching entry from `tab_name`. This question basically boils down to the age-old question whether `COUNT(*)` is the same as `COUNT(1)`, and the answer is affirmative.

2.4.3 Subquery Unnesting

The Oracle query optimizer has basically two types of transformations at its disposal:

1. Cost-based transformations, which are performed only when the optimizer believes it will improve the performance;
2. Heuristic transformations, which are generally applied regardless of what the optimizer believes.

Some heuristic transformations are really no-brainers because they cannot impact the performance negatively:

- `COUNT` conversion: whenever a `COUNT(*)` is equivalent to a `COUNT(col_not_null)` Oracle can use a `BITMAP CONVERSION COUNT` to speed up the SQL statement.
- `DISTINCT` is eliminated whenever Oracle knows the operation is redundant, for instance when all columns of a primary key are involved.
- Columns in a `SELECT`-list of a subquery are removed whenever they are not referenced, which is known as select-list pruning.
- Filters can be pushed down into subqueries when appropriate, so that the amount of data is minimized as soon as possible.
- Predicates can also be moved around from one subquery to another whenever an inner or natural join of two (or more) query blocks is performed where only one of the blocks has a particular predicate, but transitive closure guarantees that is also applies to the other subqueries. Aggregations and analytic functions are known to throw a spanner in the works for filter push-downs and predicate move-arounds.

Subquery unnesting is an interesting beast in the sense that is always applied irrespective of the impact on the performance. As such, it should be classified as a heuristic transformation were it not for the fact that it can be disabled with a hint.

Oracle automatically unnests subqueries (in the `WHERE` clause) when possible and merges the body of the subquery into the body of the statement or rewrites the nested subquery as an inline view, which of course opens up new optimization avenues, unless the nested subquery:

- contains the `ROWNUM` pseudocolumn;

- contains a set operator;
- contains an aggregate function or `GROUP BY` clause;
- contains a correlated reference to a query block that is not the immediate outer query block of the subquery;
- is a hierarchical subquery.

Why is unnesting important? Without it, Oracle needs to evaluate a (correlated) subquery *for each row of the outer table*. With it, it has a whole array of access paths and join options it can go through to improve the execution.

When you execute `SELECT * FROM dual WHERE 0 NOT IN (NULL)` you will receive no rows, as expected. After all, null may be 0, it may not be. Before Oracle Database 11g, a column that could be null would prevent Oracle from unnesting the subquery. With the null-aware anti-join in 11g and above this is no longer the case, and Oracle can unnest such subqueries.

As of Oracle Database 12c there is the so-called null-accepting semi-join, which extends the semi-join algorithm, indicated by `SEMI NA` in the execution plan. This is relevant for correlated subqueries that have a related `IS NULL` predicate, like so: `WHERE col_name IS NULL OR EXISTS ( SELECT 1 FROM ... )` The null-accepting semi-join checks for null columns in the join column of the table on the left-hand side of the join. If it is null, the corresponding row is returned, otherwise a semi-join is performed.

So, you may be wondering, 'If Oracle already unnests correlated subqueries, is there any reason to use correlated subqueries instead of joins?'

A correlated subquery is perfectly acceptable when your outer query already filters heavily and the correlated subquery is used to find corresponding matches. This often happens when you do a simple lookup, typically in a PL/SQL (table) function in an API.

Beware of nulls in the subquery of anti-joins though: whenever one or more rows return a null, you won't see any results. A predicate such as `col_name NOT IN (NULL, ...)` always evaluates to null. Analogously, it is important that you inform Oracle of nulls, or the absence thereof, in case you decide to explicitly rewrite a nested or correlated subquery as a join, as it may assist Oracle in determining a better execution plan. Remember: the more information the optimizer has, the better its decisions.

2.4.4 Combined Nested Subqueries

Sometimes you need to filter for two different (aggregated) values from a subquery. Basically, you have two options.

Option 1:

```
SELECT
  ...
FROM
  tab_name
WHERE
  col_name = ( SELECT ... FROM sub_tab_name ... )
AND another_col_name = ( SELECT ... FROM sub_tab_name ... );
```

Option 2:

```
SELECT
  ...
FROM
  tab_name
WHERE
  ( col_name, another_col_name ) =
  (
    SELECT aggregation(...), another_aggregation(...) FROM sub_tab_name ...
  );
```

The second option is to be preferred because the number of lookups in `sub_tab_name` is minimized: `col_name` and `another_col_name` are retrieved in the same round trip, potentially for each relevant row of `tab_name`.

2.4.5 Subqueries with `DISTINCT`

Let's take a look at two queries:

```
SELECT
  DISTINCT
  some_fancy_function(col_name) AS col_alias
FROM
  tab_name;
```

```
SELECT
  some_fancy_function(col_name) AS col_alias
FROM
  (
    SELECT DISTINCT col_name FROM tab_name
  );
```

Which one will run faster?

Well, in the first case, a full-table scan is done to fetch the columns, after which the function `some_function` is applied to each column, and finally Oracle looks for distinct values. In the second case, Oracle scans the table `tab_name`, returns only distinct values for `col_name`, and then applies the function to the results of the inline view. The function is invoked for every row of `tab_name` in the former query, whereas in the latter it is only called for every distinct `col_name`. Therefore, the bottom query will have better runtime performance.

Important to note is that the result sets of both may not be the same though. Suppose `col_name` contains the following *distinct* entries: 0, 1, and 2. Let's also assume that the function we want to apply is `SIN(col_name*gc_pi)`, where `gc_pi` can be a global (PL/SQL) constant defined in a package as `ACOS(-1)`. In case you have already forgotten geometric functions from basic calculus: the sine function is zero at all multiples of π. The former query will therefore return only one row with `col_alias` equal to zero, whereas the latter will return three rows, all zero.

Functions that lead to the same result set in both cases are known as bijective maps in mathematical circles. They map distinct input (domain) to distinct output (range); there is a one-to-one correspondence between the domain and the range of the function. A non-mathematical example that shows similar behaviour as our sine function is `SUBSTR(col_name, 1, 1)`. It takes the first character of each `col_name`, which means that `'Jack'` and `'Jill'` are both mapped to `'J'`.

So, when you *know* that the function you apply is a bijection, then you can rewrite your original query in the format that typically runs faster.

Sometimes you can even avoid a `DISTINCT` (with the associated costly sort operation) in a main query's `SELECT`-list altogether by opting for a semi-join (i.e. `EXISTS`) instead. This is common when you want unique entries from the main table but only when there is a match in another table for which there are multiple rows for one original row, that is, there is a one-to-many relationship from the main to the other (subquery) table.

2.4.6 Inline Views and Factored Subqueries

We tend to agree wholeheartedly with Tony Hasler in Oracle Expert SQL (pp. 9-16) when it comes to the question whether to prefer inline views to factored subqueries or the other way round. Some organizations have rules that instruct developers to use factored subqueries only when they are re-used in the same statement. When a subquery is read multiple times, such as in a recursive common table expression, factored subqueries can improve the performance

of your SQL statements, especially with the materialize or cache hint. There are, however, no performance problems associated with factored subqueries when they are queried only once, so it's more a matter of style than performance in these cases. Whenever it is more advantageous to (temporarily) materialize the factored subquery, Oracle will automatically do so. Of course, this does not always work, especially when statistics are unavailable or not representative of the current situation.

Interestingly, recursive factored subqueries can sometimes perform better than traditional solutions, especially for hierarchical queries. A detailed example is provided by Ian Hellström on Databaseline for the multiplication across the branches of a hierarchy, where an approach with a recursive factored subquery is shown to outperform the standard Oracle solution with `CONNECT BY` by several orders of magnitude.

Before the advent of factored subqueries, developers were often told that global temporary tables were the cure for bad subquery performance. That is no longer the case because either Oracle already materializes the factored subquery or you can force Oracle do to so with `/*+ materialize */`. Similarly, you can provide the hint `/*+ CACHE */`, so that Oracle caches the factored subquery, which can improve performance when the SQL statement accesses the factored subquery more than once. As of Oracle Database 12c, there is a session variable `temp_undo_enabled` that allows you to use the TEMP rather than the UNDO tablespace for temporary tables, materializations, and factored subqueries.

The only reason you may *not always* want to use factored subqueries is that in certain DML statements only inline views are permitted. Factored subqueries are easier to read and debug, hands down, and the performance is often superior too. So, unless you have a compelling reason, for instance syntax restrictions or performance, although the latter is rarely the case, stay away from inline views and go for glory with factored subqueries. For recursive subqueries and subqueries that need to be accessed multiple times in the same SQL statement, factored subqueries are pretty much your only option.

What is important, though, is – and this is by no means restricted to inline views vs factored subqueries – that you give your subquery factors meaningful names: `q1`, `a`, or `xx` do *not* qualify as meaningful names.

There is one instance, and one instance only, where an `ORDER BY` clause in an inline view or factored subquery is acceptable: top-*N* queries or pagination. If you only want the top-*N* rows based on some ordering, you simply need to sort the data. In all other cases, an intermediate sort does not make sense and may negatively affect the runtime performance. If the query blocks that work with the data from the sorted subquery perform several joins or sorts of their own, the effort of the initial sort is gone, including the time it took. When data needs to be sorted you do that in the outer query (for an inline view) or final `SELECT` statement (for a factored subquery). Often such unnecessary `ORDER BY` clauses are remnants of the development phase, so please make sure that you clean up your code afterwards.

Don't believe it? Take a look at the following SQL statement:

```
WITH
   raw_data AS
   (
     SELECT ROWNUM AS rn FROM dual CONNECT BY ROWNUM <= 1000 --ORDER BY rn DESC
   )
 SELECT * FROM raw_data ORDER BY rn;
```

The execution plan for the statement with the `ORDER BY` in the subquery factor reads (on 12c):

```
 ------------------------------------------------------------------------------------
↪----
 | Id  | Operation                       | Name | Rows  | Bytes | Cost (%CPU)| Time ␣
↪    |
 ------------------------------------------------------------------------------------
↪----
 |   0 | SELECT STATEMENT                |      |     1 |    13 |     4  (50)|␣
↪00:00:01 |
 |   1 |  SORT ORDER BY                  |      |     1 |    13 |     4  (50)|␣
↪00:00:01 |
 |   2 |   VIEW                          |      |     1 |    13 |     3  (34)|␣
↪00:00:01 |
```

```
 |   3 |    SORT ORDER BY                  |      |     1 |       |     3   (34)|␣
↪00:00:01 |
 |   4 |     COUNT                         |      |       |       |             |
↪    |
 |   5 |      CONNECT BY WITHOUT FILTERING|      |       |       |             |
↪    |
 |   6 |       FAST DUAL                   |      |     1 |       |     2    (0)|␣
↪00:00:01 |
 ---------------------------------------------------------------------------------------
↪----
```

For the statement *without* the unnecessary sorting it is (again on 12c):

```
-------------------------------------------------------------------------------------------
↪--
| Id  | Operation                         | Name | Rows  | Bytes | Cost (%CPU)| Time    ␣
↪ |
-------------------------------------------------------------------------------------------
↪--
|   0 | SELECT STATEMENT                  |      |     1 |    13 |     3   (34)|␣
↪00:00:01 |
|   1 |  SORT ORDER BY                    |      |     1 |    13 |     3   (34)|␣
↪00:00:01 |
|   2 |   VIEW                            |      |     1 |    13 |     2    (0)|␣
↪00:00:01 |
|   3 |    COUNT                          |      |       |       |             |
↪ |
|   4 |     CONNECT BY WITHOUT FILTERING|      |       |       |             |
↪ |
|   5 |      FAST DUAL                    |      |     1 |       |     2    (0)|␣
↪00:00:01 |
-------------------------------------------------------------------------------------------
↪--
```

The difference is the additional `SORT ORDER BY` operation with Id = 3.

Oracle does have a so-called `ORDER BY` elimination that removes unnecessary sorting operations, such as in subqueries. Such an elimination typically occurs when Oracle detects post-sorting joins or aggregations that would mess up the order anyway. Important to note is that said elimination procedure does *not* apply to factored subqueries, which is why the `SORT ORDER BY` operation shows up in the execution plan above!

You can have fun with the order-by-elimination by enabling/disabling it with the hints `ELIMINATE_OBY`/`NO_ELIMINATE_OBY`. Again, please observe that this fiddling around with these hints only applies to inline views. Similarly, you can use the `NO_QUERY_TRANSFORMATION` hint to disable *all* query transformations, as described by the authors in Pro Oracle SQL (pp. 45-46).

2.5 Joins

Probably the most used operation in a relational database is the infamous join. Apart from the semi- and antijoin that are basically subqueries, which we have already seen, there are roughly two types of joins: the inner and the outer join. The inner join encompasses the `[ INNER ] JOIN ... ON ...` and `NATURAL JOIN` syntax alternatives. For the outer join we have `LEFT [ OUTER ] JOIN ... ON ...`, `RIGHT [ OUTER ] JOIN ... ON ...`, `FULL [ OUTER ] JOIN ... ON ...`, but also some more exotic options that were introduced in 12c and are mainly interesting for people migrating from Microsoft SQL Server: `CROSS APPLY ( ... )`, `OUTER APPLY ( ... )`. The former is a variation on the `CROSS JOIN`, whereas the latter is a variation on the `LEFT JOIN`. [

`FULL ] JOIN ... USING ( ... )` can be used as both an inner and a full join.

As of 12c there is also a left lateral join, which can be employed with the `LATERAL ( ... )` syntax. A lateral view is "an inline view that contains correlation referring to other tables that precede it in the `FROM` clause". The `CROSS APPLY` is the equivalent of an inner lateral join, and `OUTER APPLY` does the same for outer lateral joins. A Cartesian product of two sets (i.e. tables) is achieved with the `CROSS JOIN`.

Oracle still supports the traditional syntax whereby the tables to be joined are all in the `FROM` clause, separated by commas, and join conditions are specified in the `WHERE` clause, either with or without the `(+)` notation. This syntax is generally not recommended any longer, as it has some limitations that the ANSI standard syntax does not have. Beware that there have been some bugs and performance with the ANSI syntax, although their number has decreased dramatically in more recent Oracle Database versions. For a discussion on this topic we refer to this thread and the links provided therein.

Internally Oracle translates these various joins into join methods to access the data. The options Oracle has at its disposal are:

- nested loops;
- hash join;
- sort-merge join, which includes the Cartesian join as it is a simple version of the standard sort-merge join.

Joining can, for obvious reasons, be a tad heavy on the RAM. What databases do to reduce memory usage is pipelining. It is effectively the same as what pipelined table function (in PL/SQL) do, but we should not get ahead of ourselves. Pipelining in joins means that intermediate results are immediately pipelined (i.e. sent) to the next join operation, thus avoiding the need to store the intermediate result set, which would have increased memory usage. OK, we can't help ourselves and jump the PL/SQL queue a bit: in a table function the result set is stored in a collection before it is returned. In a *pipelined* table function we pipe rows, which means that we do not store the result set in a collection but return each row as soon as it is fetched. In ETL situations, where lots of merges and transformations are typically done, pipelining can improve the performance significantly because the memory usage is reduced and rows can be loaded as soon as the database has them ready; there is no need to wait for all rows to be computed.

There are a couple of things developers can do to optimize the performance of SQL statements with joins and the main thing is pick the right one, syntactically: if you really only need an inner join, don't specify a full join 'just in case'. The more data Oracle has to fetch, the more I/O there is, and by taking data you may not need it is possible that Oracle chooses a join method that is not perhaps the best for your business case. Another tool in a developer's toolbox to boost Oracle's performance when it has to perform complex joins is understanding the various join methods and whether your situation may warrant a method that is not chosen – or even considered – by the optimizer. In such cases hints are invaluable.

Similarly, it is important that developers understand the difference between single-column predicates in the `ON` clause and the same predicates in the `WHERE` clause. Here's an example:

Query 1a:

```
SELECT
  *
FROM
  departments dept
INNER JOIN
  employees emp
ON
  dept.department_id = emp.department_id
WHERE
  emp.last_name LIKE 'X%'
;
```

Query 1b:

```
SELECT
  *
FROM
  departments dept
INNER JOIN
  employees emp
ON
  dept.department_id = emp.department_id
AND emp.last_name LIKE 'X%'
;
```

Query 2:

```
SELECT
  *
FROM
  departments dept
LEFT JOIN
  employees emp
ON
  dept.department_od = emp.department_id
WHERE
  emp.last_name LIKE 'X%'
;
```

Query 3:

```
SELECT
  *
FROM
  departments dept
LEFT JOIN
  employees emp
ON
  dept.department_id = emp.department_id
AND emp.last_name LIKE 'X%'
;
```

For inner joins the only difference is when the clauses are evaluated: the `ON` clause is used to join tables in the `FROM` clause and thus comes first – remember the query processing order from *before*? `WHERE`-clause predicates are logically applied afterwards. Nevertheless, Oracle can typically use the `WHERE` clause already when performing the join, in particular to filter rows from the join of the driving row source.

But we have not explicitly specified the driving row source. Is there thus a difference in the results from Queries 1a and 1b?

And the answer is... (cue drum roll): no!

Query 1a, on the one hand, looks at all departments, looks for employees in the departments, and finally removes any matching rows from both tables that do not have an employee with a last name that begins with an 'X'. Query 1b, on the other hand, takes the departments and returns rows when it finds a department that has an employee with a surname starting with an 'X'. Both queries do exactly the same, so for inner joins there is no *logical* difference. Personally, we would prefer Query 1a's syntax to 1b's, because the `WHERE` clause is unambiguous: it filters rows from the join. A single-column predicate in the `ON` clause of an inner join is murky at best, and should be avoided, because its intentions are not as clear as in the case of the `WHERE` clause.

For the outer joins, the difference is very real. Query 2 looks at all departments and joins the employees table. If a department happens to have no employees, the department in question is still listed. However, because of the `WHERE`

clause only rows (i.e. departments and employees) with the column `last_name` beginning with an 'X' are returned. So, even if a department has plenty of employees but none of them has a last name that starts with an 'X', no row for that department will be returned because logically the `WHERE` clause is applied to the result set of the join.

If we place the predicate in the `ON` clause, as in Query 3, we make it part of the outer join clause and thus allow rows to be returned from the left table (`departments`) even if there is no match from the right table (`employees`). The situation for `last_name` is the same as for `department_id`: if a department has no employees *or* a department has no employees with a surname that starts with an 'X', the department still shows up but with `NULL` for every column of `employees` because there are no employees that match the join criterion.

Anyway, we have already talked about joins methods *before*, but it may be beneficial to take another look at the various methods and when Oracle decides to pick one and not the others.

2.5.1 Nested Loops

Whenever you have correlated row sources for a left lateral join, Oracle uses nested loops to perform the join. Nested loops can, however, be used for uncorrelated row sources too, although that often requires some hint trickery, but more on that later when hints are in our focus.

Nested loops work by fetching the result from the driving row source and querying the probe row source for each row from the driving row source. It's basically a nested `FOR` loop, hence the name. The driving row source is sometimes also referred to as the leading row source, which is reflected in the hint `/*+ leading(...) */` that can be used to specify the leading object.

Nested loops scale linearly: if the row sources double in size, it takes roughly twice as much time to do the join with nested loops, provided that there is a relevant index on the probe row source. If there is no such index, we lose the linear scalability, because Oracle has to visit each row in the probe row source, which means that in that case nested loops scale quadratically. Appropriate indexes can often be tricky when the probe source is an inline view; Oracle typically chooses the table, if there is any, as the probe row source. A somewhat related problem is that the same blocks in the table being probed may be visited many times because different rows are looked at each time. For small driving row sources the nested loop join is often the best option.

Since 11g Oracle can prefetch nested loops, which shows up in the execution plan as the join operation being a child of a table access operation. This enables the database to first think about what it does to the ROWIDs it obtains from the nested loops. For instance, if the ROWIDs are all consecutive but not in the buffer cache, the table access operation can benefit from a multi-block read.

Multiple nested loops operations can occasionally show up in the execution plan for just one join, which indicates that Oracle used the nested-loop batching optimization technique. What this method does is transform a single join of two row sources into a join of the driving row source to one copy of the probe row source that is joined to a replica of itself on ROWID; since we now have three row sources, we need at least two nested loops. The probe row source copy that is used to perform a self join on ROWID is used to filter rows, so it will have a corresponding `TABLE ACCESS BY ... ROWID` entry in the execution plan. This cost-based optimization can often reduce I/O although the execution plan may not transparently display the benefits.

Whatever is specified in the `WHERE` clause that is exclusively applicable to the driving row source is used by Oracle to filter rows as soon as possible, even though semantically the filter comes after the `JOIN` clause.

Oracle always uses nested loops for left lateral joins. What makes lateral joins useful is that predicates derived from columns in the driving row source (i.e. the row source specified *before* the `LATERAL` keyword) can be used in the probe row source (i.e. the inline view that follows `LATERAL`).

Beware of the cardinality estimates when you use the `gather_optimizer_statistics` hint: for nested loops the estimated row count is *per iteration*, whereas the actual row count is for *all iterations*, as mentioned by Tony Hasler in Expert Oracle SQL (p. 266).

2.5.2 Hash Join

In a hash join, Oracle hashes the join key of the 'driving' row source in memory, after which it runs through the 'probe' row source and applies the hash to obtain the matches. We have placed the words 'driving' and 'probe' in quotes to indicate that the nomenclature is slightly different for hash joins though still applicable. Because of the hashing it is clear that an index on the probe row source will not improve the performance of the hash join. The only indexes that are beneficial in a hash join are indexes on predicates in the `WHERE` clause, but that is — as we have said — not specific to hash joins at all. Moreover, when the probe row source is table, a hash join does not visit blocks multiple times, since the database goes through the probe row source once. In fact, if Oracle decides to do a full table scan on the probe row source it may also decide to do multi-block reads to bump its retrieval efficiency.

It's not all peachy though. Suppose that the probe row source is huge but only very few rows match the join clause and that we have no predicate or one that is barely selective. With a hash join he database visits many blocks that contain no data we're interested in, because we cannot retrieve the rows through an index. Whether the balance is tipped in favour of nested loops with an index lookup of the probe row source, if of course available, that perhaps visits blocks over and over again, or a hash join with a single scan of each block depends mainly on the selectivity of the join condition and the clustering factor of the index. In these cases it is often advantageous to fix the execution plan with hints once you have discovered and argued that one consistently outperforms the other.

Hash joins scale linearly too, but there is one caveat — isn't there always? The entire hash table has to fit in memory. If it does not, the hash table will spill onto disk, which ruins the linear scalability to the ground. As always, select only the columns you need as the size of the hash table may increase dramatically and thus ruin the performance benefit when Oracle runs out of memory.

Another gotcha with hash joins is that they can only be used with equality join conditions.

2.5.2.1 Join Orders and Join Trees

When you hash-join several row sources with an inner join, Oracle can in principle swap the order without affecting the result set. Why would it want to do that? Well, the optimizer may discover that one particular join order is better than all the others. For an inner join of n tables, there are $n!$ possible join orders. For four tables, we have $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$ possibilities. So the chances are that there is at least one that is significantly better and one that is the absolute worst.

Let's take four tables: T1, T2, T3, and T4. A so-called left-deep join tree is obtained in the following way:

1. Place T1's hash cluster in a workarea.
2. *Join T1 and T2. Call the intermediate result set J12.*
3. Place J12's hash cluster in a workarea.
4. Drop T1's workarea.
5. *Join J12 and T3. Call the intermediate result set J123.*
6. Place J123's hash cluster in a workarea.
7. Drop J12's workarea.
8. *Join J123 and T4.*
9. Drop J123's workarea.

The italicized items are the actual logical steps in the join order. The left row source is always the driving row source in our notation, and the right row source is always the probe row source. We can also write this succinctly as ((T1 $\rightarrow$ T2) $\rightarrow$ T3) $\rightarrow$ T4.

For a right-deep join tree we have the following steps:

1. Place T4's hash cluster in a workarea.

2. Place T3's hash cluster in a workarea.
3. Place T2's hash cluster in a workarea.
4. *Join T2 and T1. Call the intermediate result set J21.*
5. Place J21's hash cluster in a workarea.
6. Drop T2's workarea.
7. *Join T3 and J21. Call the intermediate result set J321.*
8. Place J321's hash cluster in a workarea.
9. Drop T3's workarea.
10. *Join T4 and J321.*
11. Drop all remaining workareas.

We can write this as T4 → (T3 → (T2 → T1)).

What is hopefully clear from these sequences is that a left-deep join tree requires two concurrent workareas, whereas a right-deep join tree has as many workareas as row row sources. So, why on earth do we ever want a right-deep join tree?

Suppose for a second that T1 is enormous and the remaining tables are relatively small, which often happens in data warehouses. Just think of T1 as being the fact table (e.g. sales) and T2, T3, and T4 dimension tables (e.g. products, customers, and suppliers). In a left-deep join tree we would create a large workarea with T1, and potentially do a couple of Cartesian joins on the dimension tables as these often do not have join conditions with one another. This would leave us with a monstrous hash cluster for T1 that will likely not fit in memory. Moreover, the hash clusters of the Cartesian joins of the dimension tables may also easily be more than Oracle can handle. The right-deep join tree places the smaller tables in workareas and finally scans the large table T1 instead. In doing so, we have more workareas but they are all likely to fit in memory, thus allowing us to feel the wind of linear scalability in our hair as we speed through the joins.

Let's not get carried away now. How do we obtain a right-deep from a left-deep join tree? We can go from a left-deep join tree to a right-deep join tree in the following manner:

1. Swap T4: T4 → ((T1 → T2) → T3).
2. Swap T3: T4 → (T3 → (T1 → T2)).
3. Swap T2: T4 → (T3 → (T2 → T1)).

Notice that in the first swap we have obtained an intermediate result set as a probe row source.

The corresponding (simplified) execution plans would look something like the ones shown below. In particular, for the left-deep join tree we have:

```
-------------------------------------
| Id | Operation             | Name |
-------------------------------------
| 0  | SELECT STATEMENT      |      |
| 1  |  HASH JOIN            |      |
| 2  |   HASH JOIN           |      |
| 3  |    HASH JOIN          |      |
| 4  |     TABLE ACCESS FULL |  T1  |
| 5  |     TABLE ACCESS FULL |  T2  |
| 6  |    TABLE ACCESS FULL  |  T3  |
| 7  |   TABLE ACCESS FULL   |  T4  |
```

And for the right-deep join tree we see:

These are of course not all of Oracle's options. Bushy joins (yes, they are really called that) or zigzag join trees have some of the row sources swapped but not all as in the case of left-deep and right-deep join trees. An example of such a zigzag tree would be the following: (T1 → T2) → (T3 → T4). To be specific, we obtain that particular join order as indicated:

1. Join T1 and T2. Call the intermediate result set J12.
2. Join T3 and T4. Call the intermediate result set J34.
3. Join J12 and J34.

Interestingly, bushy joins are *never* considered by the optimizer. Hence, if you believe a bushy join to be the best join order possible, you have to force Oracle with the `leading` hint.

2.5.2.2 Partitioned Hash Joins

For two tables that are equijoined and both partitioned identically, Oracle does a full partition-wise join, which shows up as a `PARTITION HASH` parent operation to the `HASH JOIN` in the execution plan. Similarly it can pop up in a parallel SQL statement as `PX PARTITION HASH`. Each parallel query server reads data from a particular partition of the first table and joins it with the appropriate rows from the corresponding partition of the second table. Query servers have no need to communicate to one another, which is ideal. The only downside is if there is at least one partition that is significantly larger than all the others, as this may affect the balancing of the load.

When only one table is partitioned, Oracle can go with a (parallel) partial partition-wise join. It (re)partitions the other table on the fly based on the partitioning scheme of the reference table. Once the partitioning is out of the way, the database proceeds as it does with a full partition-wise join.

It is generally recommended to use hash instead of range partitioning for partition-wise joins to be effective, mainly because of possible data skew that leads to some partitions being larger than others. Furthermore, the number of partitions in relation to the DOP is relevant to the performance. Ideally, the number of partitions is a multiple of the number of query servers.

Both hash and sort-merge joins are possible for full partition-wise joins. More details can be found in the excellent book Expert Oracle SQL.

2.5.3 Sort-Merge Join

The sort-merge or simply merge join is actually rarely used by Oracle. It requires both row sources to be sorted by the join columns from the get-go. For equality join conditions, the sort-merge join combines both row sources like a zipper: nicely in-sync. When dealing with ranged-based join predicates, that is everything except <>, Oracle sometimes has to jump a bit back and forth in the probe row source as it strolls through the driving row source, but it pretty much does what a nested loop does: for each row from the driving row source pick the matches from the probe row source.

A Cartesian join is basically a sort-merge join, and it shows up as `MERGE JOIN CARTESIAN` in the execution plan. It is Oracle's fall-back plan: if there is no join predicate, then it has no alternative as every row in the driving row source matches each and every row in the probe row source. What is slightly different for the Cartesian join is that no actual sorting takes place even though the execution plan informs us of a `BUFFER SORT` operation. This operation merely buffers, it does *not* sort. When the Cartesian product of two row sources is relatively small, the performance should not be too horrendous.

A sort-merge join may be performed when there is no index on the join columns, the selectivity of the join columns is low, or the clustering factor is high (i.e. near the number of rows rather than the number of blocks, so the rows are ordered randomly rather than stored in order), so that nested loops are not really an attractive alternative any longer. Similarly, a sort-merge join may be done instead of a hash join when the hashed row sources are too big to fit in memory.

Note: A sort-merge join may spill onto disk too, although that typically is not as bad to performance as with a hash join.

When one row source is already sorted and Oracle decides to go ahead with a sort-merge join, the other row source will *always* be sorted even when it is already sorted.

The symmetry of the sort-merge join is unique. In fact, the join order does not make a difference, not even to the performance.

2.5.4 Join Performance: `ON` vs `WHERE`

Now that we are equipped with a better appreciation and understanding of the intricacies of the various join methods, let's revisit the queries from the introduction.

Queries 1a and 1b are logically the same and Oracle will treat them that way. First, let's assume there there is an index on `department_id` in both tables. Such an index is only beneficial to nested loops because that particular column is in the join clause. As such, the `employees` table is likely to become the driving row source, for a filter like `LIKE last_name = 'X%'` is probably very selective in many instances, which means that the number of iterations will be relatively low. While accessing the `employees` table, Oracle will apply the filter because it knows that single-column join conditions in the `ON` clause of inner joins are the same as predicates in the `WHERE` clause. The database will do so either with a lookup if the relevant index on `employees` is selective enough or by means of a full table scan if it is not highly selective. It will then use the index on `departments` to access its data by ROWID, thereby joining it to the data from the leading row source. When the filter on `last_name` is not as selective, especially when the cardinality of the `departments` table is lower than the cardinality of the `employees` table *after* the filter has been applied, the roles of driving and probe row sources are reversed.

If no such indexes exist at all, then a hash join seems logical. Whether the `departments` or `employees` table is used to generate an in-memory hash cluster depends on what table Oracle believes will be best based on the cardinality estimates available. We basically have the same logic as before: for highly selective filters, Oracle will use the `employees` table as the driving row source, otherwise it will pick (on) the `departments` table to take the lead.

Queries 2 and 3 yield different result sets, so it's more or less comparing apples and oranges. Nevertheless, with an appropriate, selective index on `last_name` Oracle will probably settle for nested loops for Query 2 (i.e. the one with the `WHERE` clause), and a hash join for Query 3 (i.e. the one with the `ON` clause). If the index on `last_name` is not selective at all and its clustering factor is closer to the number of rows than the number of blocks, then Query 2 may also be executed with a hash join, as we have discussed earlier. Should the SQL engine decide on nested loops for Query 3, it is to be expected that the `departments` table be promoted to the position of driving row source because Oracle can use the single-column join condition on `last_name` as an access predicate.

Please note that a sort-merge join is possible in all instances. The sort-merge join is rarely Oracle's first choice when faced with equality join conditions, especially when the tables involved are not sorted to start with.

So, what you should take away from this section is that even though the `WHERE` clause is technically a *post*-join filter, it can be and often is used by Oracle when it fetches the data of the leading row source, analogously to single-column predicates specified in the `ON` clause, thereby reducing the number of main iterations (i.e. over the driving row source) or the number of index lookups (in the probe row source) for nested loops, or the size of the in-memory hash cluster for a hash join. For sort-merge joins these predicates can be used to minimize the size of the tables to be sorted, if one or both tables require reordering.

2.6 Hints

According to the Oxford Dictionary, a hint is "a slight or indirect indication or suggestion". Oracle optimizer hints can be broadly categorized in two classes: 1) real hints, and 2) instructions. The former, real hints, match the dictionary

definition: they provide Oracle with pertinent information that it may not have when it executes a particular statement. The information provided manually, for instance a cardinality estimate of an object, may aid in the search for an optimal execution plan. The latter are really best called instructions or directives as they tell the optimizer what (not) to do.

Hints are actually comments in SQL statements that are read by the optimizer. They are indicated by the plus sign in `/*+ SOME_HINT */`. Without the plus sign the comment would be just that: a comment. And the optimizer does not care about the comments you add to increase the quality or legibility of your code. With the plus sign, the optimizer uses it do determine the execution plan for the statement.

As always, the hint may be written in any case (UPPER, lower, or miXEd) but it must be valid for the optimizer to be able to do anything with it. Whether you add a space after the '+' and/or before the closing '*/' is up to you; Oracle does not care either way.

It is customary to place the hint directly after the SQL verb (e.g. `SELECT`) but it is not necessary to do so. However, the hint must follow the plus sign for the optimizer to pick it up, so do not place any comments in front of the hint. In fact, we recommend that you do not mix comments and hints anyway. Several hints may be placed in the same 'comment'; they have to be separated by (at least) one space, and they may be on different lines.

Mind you, the syntax checker does *not* tell you whether a hint is written correctly and thus available to the optimizer! You're on your own there. If it's not valid, it's simply ignored.

We shall henceforth use upper-case hints, lower-case parameters that have to be provided by the developer, and separate all parameters by two spaces to make it easier for you to read the hints. Some people prefer to use commas between parameter values but that is not strictly necessary, so we shall leave out the commas.

2.6.1 When To Use Hints

Oracle recommends that "hints [...] be used sparingly, and only *after you have collected statistics on the relevant tables and evaluated the optimizer plan without hints* using the `EXPLAIN PLAN` statement." We have added the emphasis because that specific phrase is critical: don't add hints because you *think* you know better than the optimizer. Unless you are an Oracle virtuoso you probably do not understand the fine print of each hint and its impact on the performance of your statements – we certainly don't.

Still, you may have good reasons to add hints to your statements. We have listed some of these reasons below.

- For demonstrative purposes.
- To try out different execution plans in the development phase.
- To provide Oracle with pertinent information when no such information is available in statistics.
- To fix the execution plan when you are absolutely sure that the hints lead to a significant performance boost.

There may be more, but these four sum up the whys and wherefores pretty well.

2.6.2 When Not To Use Hints

Even though Oracle – ahem – hints us to use hints as a last resort, Oracle whizz Jonathan Lewis goes even further and pretty much has one simple rule for hints: don't.

We do not take such a grim view of the world of hints but do wish to point out that there are actually very good reasons not to use hints at all.

A common use case for hints is when statistics are out of date. In such cases hints can indeed be useful, but an approach where representative statistics are locked, such as the TSTATS approach, may be warranted and in fact more predictable. Hints are basically static comments that fiddle around with the internals of the optimizer. Dynamically

generated hints (i.e. hints generated on-the-fly in dynamic SQL) are *extremely* rare, and come to think of it, we have never seen, heard of, or read about them anywhere. Full stop.[1]

Because hints are static they are in some ways the same as locked statistics. The only difference is that once you have seen (in development) that different statistics are more appropriate and you need to do a refresh, you can release the statistics into production by copying the statistics from development to production. Oracle will take care of the rest, and the optimizer can figure out the new best execution plan on its own, which will be the same as in development because the statistics are identical. You really don't want to run into performance surprises in production, especially if they suck the life out of your statements.

With hints, you have to check the performance against both the development and the production instance(s), as the statistics may very well be different. What you lose with hints is the predictability of locked statistics. You also need to regularly verify that the hints still perform as initially intended. And typically that is not something you want to do in a live system.

Moreover, when Oracle releases a new version of its flagship database, it usually comes packed with lots of improvements to the optimizer too. If you have placed hints inside your code, the optimizer does what you tell it to do, and you are unlikely to benefit from any new features. Some of these features may not always be helpful but in many cases they really are.

This brings us to another important point. Hints should be revisited regularly and documented properly. Without documentation no one except you during a period of a few hours or perhaps days, depending on nimbleness of your short-term memory and the amount of beer and pizza consumed in the time afterwards, will know why the hints were required at all, and why that particular combination of hints was chosen.

In summary, don't use hints when

- what the hint does is poorly understood, which is of course not limited to the (ab)use of hints;
- you have not looked at the root cause of bad SQL code and thus not yet tapped into the vast expertise and experience of your DBA in tuning the database;
- your statistics are out of date, and you can refresh the statistics more frequently or even fix the statistics to a representative state;
- you do not intend to check the correctness of the hints in your statements on a regular basis, which means that, when statistics change, the hint may be woefully inadequate;
- you have no intention of documenting the use of hints anyway.

Notes

2.6.3 Named Query Blocks

You may have already seen object aliases in the execution plan. An object alias is a concatenation of the name of an object or its alias and the name of the query block it appears in. A query block is any inline view or subquery of a SQL statement.

Object aliases typically look something like `tab_name@SEL$1`, `tab_nameINS$2`, `tab_nameUPD$3`, `tab_name@DEL$4`, or `tab_name@MISC$5` . These automatically generated names are hardly insightful, which is why you are allowed to name query blocks yourself.

[1] Even though we have never observed dynamically generated hints in the wild we can still perform a Gedankenexperiment to see why they seem like an odd idea anyway. Suppose you want to provide cardinality estimates with the undocumented `CARDINALITY` hint based on the parameter values of a subprogram, for instance the parameter of a table function. You may think this is a great idea because you already know about skew in your data, and you want to provide estimates based on your experience. Fine. Unfortunately, you cannot bind the estimate itself, which means that Oracle requires a hard parse, as the hint is simply a literal. This is tantamount to hard-coding the hint and choosing the statement to run with branches of a conditional statement, which sort of defeats the purpose of generating the estimate dynamically. Creating several alternatives based on parameter values may, however, be useful and beneficial to the performance, especially in cases of severe data skew.

You name query blocks with `/*+ QB_NAME( your_custom_qb_name ) */`. Afterwards you can reference objects from that named query block using `@your_custom_qb_name tab_name_or_alias`. The optimizer will use the custom name instead of `SEL$1` or whatever is applicable, so you can more easily understand the execution plan's details.

Note: The optimizer ignores any hints that reference different query blocks with the same name.

Should you name all query blocks?

Hell no! Only use the query block name hint when your statements are complex and you need to reference objects from various query blocks in your hints. When would you want to do that? When you use global hints.

2.6.4 Global Hints

Hints are commonly embedded in the statement that references the objects listed in the hints. For hints on tables that appear inside views Oracle recommends using global hints. These hints are not embedded in the view itself but rather in the queries that run off the view, which means that the view is free of any hints that pertain to retrieving data from the view itself.

We shall presume that we have created a view called `view_name`. The view does a lot of interesting things but what we need for a global hint in our query that selects data from our view is a table `tab_name` inside a subquery (e.g. inline view or factored subquery) with the alias `subquery_alias`. We would then write `SELECT /*+ SOME_HINT( view_name.subquery_alias.tab_name ) */ * FROM view_name`, where `SOME_HINT` is supposed to be any valid optimizer hint.

Similarly we could use a named query block to do the same: `/*+ SOME_HINT( @my_qb_name tab_name )`, where `my_qb_name` is the name we have given to the query block in which `tab_name` appears. You can also use the automatically generated query block names but that is begging for trouble. Named query blocks are really useful in conjunction with global hints.

2.6.5 Types of Hints

Oracle has kindly provided an alphabetical list of all *documented* hints. There are also a bunch of undocumented ones, and examples of their use can be found scattered all over the internet and in the multitude of books on Oracle performance tweaking. Undocumented hints are not more dangerous than their documented equivalents; Oracle simply has not gotten round to documenting them yet.

Oracle classifies hints based on their function:

- Optimization goals and approaches;
- Access path hints;
- In-memory column store hints;
- Join order hints;
- Join operation hints;
- Parallel execution hints;
- Online application upgrade hints;
- Query tranformation hints;
- XML hints;
- Other hints.

In Oracle Database 12c Performance Tuning Recipes, the authors provide two additional types of hints:

- Data warehousing hints;
- Optimizer hints.

The data warehousing hints are actually included in Oracle's query transformation hints.

Access path and query transformation hints are by far the largest two categories, save for the miscellaneous group.

Although interesting in their own way we shall not discuss in-memory column store hints, online application upgrade hints, and XML hints. We shall now go through the remaining categories and discuss the most important hints for each category, so you too can supercharge your SQL statements. There are many more hints than we describe here, and you are invited to check the official documentation for more hints and details.

2.6.5.1 Optimization Goals and Approaches

Oracle only lists two hints in this category: `ALL_ROWS` and `FIRST_ROWS( number_of_rows )`. These are mutually exclusive. If you happen to be drunk while programming and inadvertently write both hints in the same statement, Oracle will go with `ALL_ROWS`.

In mathematical optimization nomenclature, these two hints affect the objective function. `ALL_ROWS` causes Oracle to optimize a statement for throughput, which is the minimum *total* resource consumption. The `FIRST_ROWS` hint does not care about the throughput and instead chooses the execution plan that yields the first `number_of_rows` specified as quickly as possible.

Note: Oracle ignores `FIRST_ROWS` in all `DELETE` and `UPDATE` statements and in `SELECT` statement blocks that include sorts and/or groupings, as it needs to fetch all relevant data anyway.

2.6.5.2 Optimizer Hints

We have already mentioned the `GATHER_PLAN_STATISTICS` hint, which can be used to obtain statistics about the execution plan during the execution of a statement. It is especially helpful when you intend to diagnose performance issues with a particular statement. It is definitely not meant to be used in production instances!

There is also a `GATHER_OPTIMIZER_STATISTICS`, which Oracle lists under 'Other hints'. It can be used to collect bulk-load statistics for CTAS statements and `INSERT INTO ... SELECT` statements that use a direct-path insert, which is accomplished with the `APPEND` hint, but more on that later. The opposite, `NO_GATHER_OPTIMIZER_STATISTICS` is also provided.

The `OPTIMIZER_FEATURES_ENABLE` hint can be used to *temporarily* disable certain (newer) optimizer feature after database upgrades. This hint is typically employed as a short-term solution when a small subset of queries performs badly. Valid parameter values are listed in the official documentation.

2.6.5.3 Access Path Hints

Access path hints determine how Oracle accesses the data you require. They can be divided into two groups: access path hints for tables and access path hints for indexes.

Tables

The most prominent hint in this group is the `FULL( tab_name )` hint. It instructs the optimizer to access a table by means of a full table scan. If the table you want Oracle to access with a full table scan has an alias in the SQL

statement, you have to use the alias rather than the table name (without the schema name) as the parameter to `FULL`. For named query blocks you have to provide the query block's name as discussed previously.

In this group are also the `CLUSTER` and `HASH` hints, but they apply only to tables in an indexed cluster and hash clusters respectively.

Indexes

The hints in this group all come in pairs:

- `INDEX` / `NO_INDEX`
- `INDEX_ASC` / `INDEX_DESC`
- `INDEX_FFS` / `NO_INDEX_FFS`
- `INDEX_SS` / `NO_INDEX_SS`
- `INDEX_SS_ASC` / `INDEX_SS_DESC`
- `INDEX_COMBINE` / `INDEX_JOIN`

All these hints take at least one parameter: the table name or alias in the SQL statement. A second parameter, the index name(s), is optional but often provided. If more than one index is provided, the indexes are separated by at least one space; the `INDEX_COMBINE` hint is recommended for this use case though.

Let's get cracking. The first pair instructs the optimizer to either use (or not use) an index scan on a particular table. If a particular index is specified, then Oracle uses that index to scan the table. If no index is specified and the table has more than one index, the optimizer picks the index that leads to the lowest cost when scanning the data. These hints are valid for any function-based, domain, B-tree, bitmap, and bitmap join index.

Similarly, you can tell the optimizer that it needs to scan the specified index in ascending order with `INDEX_ASC` or descending order with `INDEX_DESC` for statements that use an index range scan. Note that if your index is already in descending order, Oracle ignores the `INDEX_DESC` hint.

No, `FFS` does not stand for "for f*ck's sake". Instead it indicates that Oracle use a fast full index scan instead of a full table scan.

An index skip scan can be enabled (disabled) with `INDEX_SS` (`NO_INDEX_SS`). For index range scans, Oracle scans index entries in ascending order if the index is in ascending order and in descending order if the index is in descending order. You can override the default scan order with the `INDEX_SS_ASC` and `INDEX_SS_DESC` hints.

The pair `INDEX_COMBINE` and `INDEX_JOIN` is the odd one out, as they are not each other's opposites. `INDEX_COMBINE` causes the optimizer to use a bitmap access path for the table specified as its parameter. If no indexes are provided, the optimizer chooses whatever combination of indexes has the lowest cost for the table. When the `WHERE` clause of a query contains several predicates that are covered by different bitmap indexes, this hint may provide superior performance, as bitmap indexes can be combined very efficiently. If the indexes are not already bitmap indexes, Oracle will perform a `BITMAP CONVERSION` operation. As Jonathan Lewis puts it in the comments section of this blog post: it's a damage-control access path. You generally would not want to rely on bitmap conversions to combine indexes; it is often much better to improve upon the index structure itself.

The `INDEX_JOIN` instructs the optimizer to join indexes (with a hash join) to access the data in the table specified. You can only benefit from this hint when there is a *sufficiently* small number of indexes that contains all columns required to resolve the query. Here, 'sufficiently' is Oraclespeak for as few as possible. This hint is worth considering when your table has many indexed columns but only few of them are referenced (p. 560) in your statement. In the unfortunate event that Oracle decides to join indexes and you are certain that that is not the optimal access path, you cannot directly disable it. Instead you can use the `INDEX` hint with only one index or the `FULL` hint to perform a full table scan.

2.6.5.4 Join Order Hints

The optimizer lists all join orders to choose the best one. What it does not do is an exhaustive search.

In case you believe a different join order to be useful, you can use one of the join order hints: `ORDERED` or `LEADING`. The latter is more versatile and should thus be preferred.

`ORDERED` takes no parameters and instructs the optimizer to join the tables in the order as they appear in the `FROM` clause. Because the `ORDERED` hint is so basic and you do not want to move around tables in the `FROM` clause, Oracle has provided us with the `LEADING` hint. It takes the table names or aliases (if specified) as parameters, separated by spaces.

In the optimizer's rock-paper-scissors game, `ORDERED` beats `LEADING` when both are specified for the same statement. Moreover, if two or more conflicting `LEADING` hints are provided, Oracle ignores all of them. Similarly, any `LEADING` hints are thrown into the bin when they are incompatible with dependencies in the join graph.

2.6.5.5 Join Operation Hints

Join operation hints are also paired:

- `USE_HASH` / `NO_USE_HASH`
- `USE_MERGE` / `NO_USE_MERGE`
- `USE_NL` / `NO_USE_NL`

These hints allow you to instruct the optimizer to use a hash join, a sort-merge join, or nested loops, respectively.

Hash joins support input swapping, which we have discussed when we talked about *left-deep and right-deep join trees*. This can be accomplished with `SWAP_JOIN_INPUTS` or prohibited with `NO_SWAP_JOIN_INPUTS`.

The left-deep join tree can be enforced with the following hints:

```
/*+ LEADING( t1 t2 t3 t4 )
    USE_HASH( t2 )
    USE_HASH( t3 )
    USE_HASH( t4 )
    NO_SWAP_JOIN_INPUTS( t2 )
    NO_SWAP_JOIN_INPUTS( t3 )
    NO_SWAP_JOIN_INPUTS( t4 ) */
```

We could have also written `USE_HASH( t4 t3 t2 )` instead of three separate hints.

So, how do we go from a left-deep join ((T1 → T2) → T3) → T4 to a right-deep join T4 → (T3 → (T2 → T1))? Remember the steps we had to perform, especially the swaps? The process to go from the left-deep join tree to the right-deep join tree is to swap the order in the following sequence: T4, T3, and T2. We can thus obtain the right-deep join tree by taking the left-deep join tree as a template and providing the necessary swaps:

```
/*+ LEADING( t1 t2 t3 t4 )
    USE_HASH( t2 )
    USE_HASH( t3 )
    USE_HASH( t4 )
    SWAP_JOIN_INPUTS( t2 )
    SWAP_JOIN_INPUTS( t3 )
    SWAP_JOIN_INPUTS( t4 ) */
```

The `LEADING` hint refers to the situation *before* all the swaps. Important to know is that the left-deep join tree is *always* the starting point.

Oracle occasionally bumps into bushy trees when views cannot be merged. Bushy trees can, however, be practical in what is sometimes referred to as a snowstorm schema, but we shall not go into more details here. In instances where a bushy join is known to be advantageous you may have to rewrite your query. For example, you can force Oracle to perform the bushy join (T1 → T2) → (T3 → T4) by writing the query schematically as follows:

```
SELECT /* LEADING ( v12 v34 )
          USE_HASH( v34 )
          NO_SWAP_JOIN_INPUTS( v34 ) */
  *
FROM
  (
    SELECT /*+ LEADING( t1 t2 )
               NO_SWAP_JOIN_INPUTS( t2 )
               USE_HASH( t2 )
               NO_MERGE */
      *
    FROM
      t1 NATURAL JOIN t2
  ) v12
NATURAL JOIN
  (
    SELECT /*+ LEADING( t3 t4 )
               NO_SWAP_JOIN_INPUTS( t4 )
               USE_HASH( t4 )
               NO_MERGE */
      *
    FROM
      t3 NATURAL JOIN t4
  ) v34
;
```

You may have noticed that we have sneaked in the `NO_MERGE` hint, which we shall describe in somewhat more detail below. What is more, we have used a `NATURAL JOIN` to save space on the `ON` or `USING` clauses as they is immaterial to our discussion.

Can you force Oracle to do a bushy join without rewriting the query?

Unfortunately not. The reason is that there is no combination of swaps to go from a left-deep join tree to any bushy join tree. You can do it with a bunch of hints for a zigzag trees, because only some of the inputs are swapped, but bushy trees are a nut too tough to crack with hints alone.

When you use `USE_MERGE` or `USE_NL` it is best to provide the `LEADING` hint as well. The table first listed in `LEADING` is generally the driving row source. The (first) table specified in `USE_NL` is used as the probe row source or inner table. The syntax is the same for the sort-merge join: whichever table is specified (first) is the inner table of the join. For instance, the combination `/*+ LEADING( t1 t2 t3 ) USE_NL( t2 t3 ) */` causes the optimizer to take T1 as the driving row source and use nested loops to join T1 and T2. Oracle then uses the result set of the join of T1 and T2 as the driving row source for the join with T3.

For nested loops there is also the alternative `USE_NL_WITH_INDEX` to instruct Oracle to use the specified table as the probe row source and use the specified index as the lookup. The index key must be applicable to the join predicate.

2.6.5.6 Parallel Execution Hints

Not all SQL statements can be run in parallel. All DML statements, including subqueries, can be run in parallel, which means that multiple blocks can be selected, inserted, deleted, or updated simultaneously. For parallelized DDL statements, multiple blocks are being created/altered and written in parallel. The DDL statements that can be run in parallel are:

- CREATE INDEX
- CREATE TABLE ... AS SELECT
- ALTER INDEX ... REBUILD
- ALTER INDEX ... [REBUILD | SPLIT] PARTITION
- ALTER TABLE ... MOVE
- ALTER TABLE ... [MOVE | SPLIT | COALESCE] PARTITION

Note that for the CTAS statement it is possible to perform a parallel DML (i.e. SELECT) operation but write the data to disk serially, which means that it is not a parallel DDL operation. We do not intend to dwell on such technicalities though.

The parallel execution of DDL statements requires that it be enabled either at the level of the session or by specifying the appropriate PARALLEL clause for the statement. When set manually for a table or index with the ALTER { TABLE | INDEX } obj_name PARALLEL dop , the degree of parallelism (DOP) used to be for both subsequent DDL *and* DML statements *prior to 12c*. Beware of that trap! Nevertheless, since this is a section on optimizer hints, we have no intention of delving into the specifics on non-hinted parallel execution.

As of 11gR2 Oracle has had the PARALLEL(dop) and NO_PARALLEL (tab_name_or_alias) for individual statements rather than sessions or objects. The degree of parallelism dop is optional, and if omitted Oracle computes it for you; the minimum degree of parallelism is 2. The PARALLEL hint will cause all access paths than can use parallelism to use it; in essence, the hint authorizes the optimizer to use parallelism. The hint can be supplied to the SELECT, INSERT, UPDATE, DELETE, or MERGE bits of a statement.

Instead of supplying dop, you can also write a) DEFAULT, which means that the DOP is equal to the number of CPUs available on all instances multiplied by the value of the PARALLEL_THREADS_PER_CPU initialization parameter, b) AUTO, which causes the optimizer to decide on the degree of parallelism or whether to run the statement serially, or c) MANUAL, for which the degree of parallelism is determined by the objects in the statement.

The PARALLEL hint can also be set for specific objects in a SQL statement as follows: PARALLEL(tab_name_or_alias dop). You may also provide DEFAULT as an alternative to dop; its behaviour is identical to the statement-level's hint.

In Expert Oracle SQL (p.152) it is noted that when inserting data in parallel *before* committing causes subsequent selects to fail *until* the data is committed. The reason is that a direct path write can sometimes be used by parallel DML statements, especially inserts. The rows for a direct path write are not in the *SGA* and must be read from disk. However, before the data is committed there is no fresh data to read from disk!

The NO_PARALLEL hint overrides the PARALLEL parameter supplied at the creation or alteration of any table.

In a similar fashion you can instruct Oracle to scan index ranges in parallel with the PARALLEL_INDEX(tab_name_or_alias index_name dop). With NO_PARALLEL_INDEX(tab_name_or_alias index_name) you can disable parallel index range scans. In both hints, index_name is optional.

With PQ_CONCURRENT_UNION you force the optimizer to process UNION [ALL] operations in parallel. NO_PQ_CONCURRENT_UNION disables concurrent processing of said set operations.

When the distribution of the values of the join keys for a parallel join is highly skewed because many rows have the same join key values, parallelizing a join can be troublesome as the load is not easily distributed among the query servers. To that end Oracle introduced the PQ_SKEW(tab_name_or_alias) hint, which informs the optimizer of data skew in the join keys. Oracle requires a histogram on the join expression as otherwise it will probe rows at random to discover the skew; it also seems that only single inner joins are supported. Similarly, there is a NO_PQ_SKEW(tab_name_or_alias) to advise the optimizer that most rows do not share the same join keys. In both hints, tab_name_or_alias is the hash join's probe row source.

CTAS and INSERT INTO ... SELECT statements' distribution of rows between producers and consumers can be controlled with the PQ_DISTRIBUTE(tab_name_or_alias distribution) hint. The value of distribution can be one of the following:

- `NONE`: no distribution, which is ideal when there is no skew, so the overhead of distributing rows can be avoided. It is important to be aware that each query server munches between 512 KB and 1.5 MB (with compression) of *PGA* memory.
- `PARTITION`: rows are distributed from producers to consumers based on `tab_name_or_alias`'s partition information, which is best used when producer and consumer operations cannot be combined, there are more partitions than query servers, and there is no skew across partitions.
- `RANDOM`: rows are distributed from the consumers to the consumers in a round-robin fashion, which is applicable when the data is skewed.
- `RANDOM_LOCAL`: rows are distributed from the consumers to the consumers on the same RAC node in a round-robin fashion, which eliminates inter-node communication.

For joins it is also possible to use the hint in a slightly modified form: `PQ_DISTRIBUTE( tab_name_or_alias outer_distribution inner_distribution )`. All possible values are summarized in the table below.

`outer_distribution`	`inner_distribution`	Explanation	Use Case
`HASH`	`HASH`	Rows of both tables are mapped with a hash function on the join keys. Each query server performs the join between pair of resulting partitions.	Tables have comparable sizes and join uses hash-join or sort-merge join.
`BROADCAST`	`NONE`	Rows of *outer* table are broadcast to each query server; rows of inner table are partitioned randomly.	*Outer* table is small compared to inner table: inner-table size multiplied by number of query servers must be greater than outer-table size.
`NONE`	`BROADCAST`	Rows of *inner* table are broadcast to each query server; rows of outer table are partitioned randomly.	*Inner* table is small compared to outer table: inner-table size multiplied by number of query servers must be less than outer-table size.
`PARTITION`	`NONE`	Rows of *outer* table are mapped using partitioning of inner table; inner table must be partitioned on join keys.	Number of partitions of *outer* table is roughly equal to number of query servers.
`NONE`	`PARTITION`	Rows of *inner* table are mapped using partitioning of outer table; outer table must be partitioned on join keys.	Number of partitions of *outer* table is roughly equal to number of query servers.
`NONE`	`NONE`	Each query server joins a pair of matching partitions.	Tables are equipartitioned on join keys.

Please note that the last entry corresponds to the full partition-wise join we talked about *earlier*.

Finally, we have `PQ_FILTER`, which tells Oracle how to process rows for correlated subqueries. The following table shows all four parameter values, how the rows on the left-hand side and right-hand side of the filter are processed, and when best to use a particular parameter.

Parameter	LHS	RHS	Use Case
`HASH`	Parallel: hash distribution	Serial	No skew in LHS data distribution
`NONE`	Parallel	Parallel	No skew in LHS data distribution *and* LHS distribution best avoided (e.g. many rows in LHS)
`RANDOM`	Parallel: random distribution	Serial	Skew in LHS data distribution
`SERIAL`	Serial	Serial	Overhead of parallelization too high (e.g. few rows in LHS)

2.6.5.7 Query Transformation Hints

Again, all hints in this category, save for the generic NO_QUERY_TRANSFORMATION hint, come in couples:

- FACT / NO_FACT
- MERGE / NO_MERGE
- NO_EXPAND / USE_CONCAT
- REWRITE / NO_REWRITE
- UNNEST / NO_UNNEST
- STAR_TRANSFORMATION / NO_STAR_TRANSFORMATION

With NO_QUERY_TRANSFORMATION you disable all query transformation that the optimizer can perform. What the hint does *not* disable, though, are transformations that the optimizer *always* applies, such as the count transformation, predicate move-around, filter push-down, distinct elimination, and select-list pruning. This is of course no hint for a production environment, and it should only be used for testing purposes.

Generic Transformations

We have already briefly seen the NO_MERGE(view_name_or_alias) hint. It prohibits the optimizer from merging views in a query. Similarly, you can force Oracle to merge (inline) views with MERGE(view_name).

Note: MERGE and NO_MERGE have nothing to do with the sort-merge join!

When the view contains a GROUP BY clause or DISTINCT operator (or UNIQUE) operator, the MERGE hint only works if complex view merging is enabled. The delayed evaluation of these operations can either improve or worsen performance, so use these hints wisely and sparingly. For instance, join conditions may reduce the data volume to be grouped or sorted, which may be beneficial to performance. Likewise, it can be advantageous to aggregate data as early as possible to deal with less data in subsequent operations. The optimizer uses the cost to determine whether it is better to merge views or not. Complex view merging also allows uncorrelated IN-subqueries to be merged into the main SQL statement.

USE_CONCAT always enables the OR-expansion, which transforms combined OR-conditions or IN-lists in the WHERE clause into a compound query with the UNION ALL operator. Whether the cost with such an OR-expansion is truly lower than without it is irrelevant: when specified Oracle does as instructed. NO_EXPAND makes the optimizer discard the OR-expansion as a possible query transformation.

Subquery unnesting can be forced without regard for the cost with the UNNEST hint. It combines subqueries in the WHERE, such as in IN-lists, into the FROM clause, which opens the door to more access paths for the optimizer to tinker with. Without subquery unnesting, Oracle treats the main query and its subqueries as separate statements: the subqueries are executed, and their results are used to run the main query. Subquery unnesting is possible if and only if the resulting join statement is guaranteed to returns the same rows as the original statement, for instance thanks to a primary key, and the subquery does not contain any aggregate functions. NO_UNNEST is, as you may have guessed, used to disable subquery unnesting. Oracle unnests subqueries automatically unless hinted, regardless of cost expected.

Materialized views that have been created with the ENABLE QUERY REWRITE clause can be used to provide data to queries that do not explicitly call these materialized view in their FROM clauses. Contrary to regular views, which are nothing but stored queries, materialized views store the result sets of the queries that define them and regularly refresh the data. Materialized views are particularly useful for queries that are run often, as a snapshot of the data is taken and stored, so the data does not have to be calculated from scratch every time a user asks for it. However, some users may not be aware of these materialized views, which is why they are executing their own queries that ask for the same data as contained in the materialized views. With REWRITE you allow people to benefit from the data of these

materialized views; the hint has an optional parameter, which is the name of the materialized view. Typically, Oracle does this automatically when it determines that such a rewrite is beneficial. If successful, it shows up in the execution plan as `MAT_VIEW REWRITE`.

`NO_REWRITE` overrides the `ENABLE QUERY REWRITE` clause, if present. This can be helpful if you know that the data in the materialized view is stale compared to the source tables, and your query needs the current state of affairs.

The Star Transformation

In many data warehouses and OLAP databases that power business intelligence solutions, the dimensional rather than the entity-relationship data model is the gold standard.[1] Fact tables contain all information pertinent to a user's queries, and they can easily be joined to so-called dimension tables with more details on the dimensions listed in the fact table. The schema for such databases resembles what we refer to as a snowflake schema. In such instances, a star transformation can be useful, to which end Oracle has introduced the `STAR_TRANSFORMATION` hint. When specified, Oracle does not guarantee that it will be used.

A requirement for the star transformation is that there be a single-column bitmap index on all foreign-key columns of the fact table that participate in the join. The star transformation progresses in two steps:

1. Transform the original query with the join into a query with the fact table in the `FROM` clause and the dimension tables as subqueries in the `WHERE` clause to filter rows from the fact table based on the dimensions' values or ranges. The bitmap indexes are then combined with the bitmap `AND`-operation to select only the rows that satisfy all dimensional constraints. The advantage is that all the dimension tables are logically joined with the fact table only once rather than once for each dimension table. Which join method is used depends on what the optimizer decides is best, although typically for large data volumes a hash join is chosen.
2. Adjoin the rows from the dimension tables to the fact table using the best access method available to the optimizer, which is typically a full table scan because dimension tables are often relatively small.

We have said that Oracle does not always perform a star transformation, even though the `STAR_TRANSFORMATION` hint is specified. This is even true when all prerequisites, such as said bitmap indexes on the fact table, are met. In fact, the optimizer calculates the best plan without the transformation and only then compares it to the best plan with the transformation. Based on the costs of both plans, it picks one, which may not always be the one with the transformation enabled. One such case is when a large fraction of the rows in the fact table need to be fetched, for instance because the constraints on the dimension tables are not selective enough. It is then often advantageous to do a full table scan with multi-block reads.

Most of the time, database developers are told that bind variables are the key to great performance. When your query has bind variables, the star transformation will never be used though.

Another instance when star transformations are never applied is when *fact* tables are accessed remotely, that is through a database link. Dimension tables may, however, be on different Oracle database instances.

Anti-joins, fact tables that are unmerged or partitioned views, and dimension tables that appear both in the `FROM` clause and as subqueries in the `WHERE` clause are a few other party poopers for the star transformation.

The `FACT( tab_name_or_alias )` hint can be used to inform the optimizer which table should be considered the fact table. `NO_FACT` is exactly the opposite.

[1] We have no intention of starting a debate on the data model paradigms of Kimball and Inmon. The interested reader will find plenty of insightful articles on the internet.

Notes

2.6.5.8 Miscellaneous Hints

This category contains both documented and undocumented hints. The ones we describe below are by no means meant to be an exhaustive list. We have grouped them by topic for your convenience.

Direct-Path Inserts

A direct-path insert is an `INSERT` that stores data from the high-water mark (HWM) onward irrespective of the space available in the blocks below the HWM. The advantage of a direct-path insert is that Oracle does not have to check whether any blocks below the HWM are available. If a table is set to `NOLOGGING`, then Oracle minimizes redo generation, which means that a direct-path insert is generally faster than a regular insert.

For tables that data is never deleted from, this is fine, as there probably is no space below the HWM anyway. When a table does have ample space below the HWM because of occasional `DELETE` statements, which do not cause the HWM to drop, the table may take up (and waste) a lot of space, even if it contains very little data, as the HWM is gradually moved up with each direct-path insert and never dropped. This in turn may significantly (negatively) affect the performance of queries against that table. With `TRUNC` the HWM is always dropped to the lowest level possible, which is best in conjunction with direct-path inserts.

Since a direct-path insert is basically the same as appending data, the hint is named accordingly: `APPEND`. This hint is used for `INSERT INTO ... SELECT` statements, whereas the `APPEND_VALUES` hint is for `INSERT INTO ... VALUES` statements. `NOAPPEND` – without an underscore! – makes sure that the data is not inserted by means of a direct-path insert. These hints do not affect anything other than `INSERT` statements. How space is managed during a direct-path insert is described in detail on the official optimizer blog.

Important to know is that during direct-path inserts certain constraints are disabled. Only `NOT NULL` and `UNIQUE` (hence also `PRIMARY KEY`) constraints remain *enabled*. Rows that violate `UNIQUE` constraints are, however, still loaded, which is different from the normal behaviour, where such rows are rejected.

Not all tables can use a direct-path insert though. In particular, clustered tables, tables with *VPD* policies, tables with `BFILE` columns. Similarly, direct-path insert is not possible on a *single partition* of a table if it has global indexes defined on it, referential (i.e. foreign-key) and/or check constraints, or *enabled* triggers defined. Furthermore, no segments of a table can have open transactions.

What about partial deletions that cannot be simply `TRUNC`'d? The best solution is to partition the table and drop entire partitions. Beware that `TRUNC` is a DDL statement, which means that it comes with an implicit `COMMIT` in contrast to `DELETE`, which is a DML statement and requires an explicit `COMMIT` (or `ROLLBACK`).

Caching vs Materialization

When Oracle performs a full table scan, it can place the blocks retrieved in the buffer cache, so that other SQL statements can benefit from the cached data. This can be accomplished with the `CACHE( tab_name_or_alias )` hint, which typically has to be supplemented with the `FULL( tab_name_or_alias )` hint to ensure a full table scan is used. Because this only works for full table scans and the buffer cache is limited in size, this is often best for small lookup tables. The data is placed at the *most* recently used end of the least recently used (LRU) list in the buffer cache, which is Oracle's way of saying that the blocks line up for a LIFO queue. `NOCACHE` – again, no underscore – puts the blocks retrieved at the *least* recently used end of the LRU, which is the default and in most cases means that the data is discarded from the cache almost immediately.

Results of a query or query fragment, including those that are not obtained by means of a full table scan, can be cached with the `RESULT_CACHE` hint. The hint can be placed at the top level, in a factored subquery, or an inline view. Subsequent executions of the same statement can be satisfied with data from the cache, which means that Oracle can

save on a few round trips to the database. Cached results are automatically invalidated when a database object upon which the result depends is modified.

It is also possible to use system or session settings and/or table annotations to enable the *result cache*. Typically the initialization parameter `RESULT_CACHE_MODE` is set to `MANUAL`, as `FORCE` causes all statements' results to be cached, which is a bad idea when set at the system level. The `RESULT_CACHE` attribute of tables is set to either the `FORCE` or `DEFAULT` mode. `DEFAULT` requires the `RESULT_CACHE` hint in all queries where the results should be cached, and because it is the default often requires no further action. In case a table is set to `FORCE` mode, the `NO_RESULT_CACHE` hint can be used to override this behaviour for individual queries. Table annotations apply to entire queries that reference these tables, not just individual query blocks.

Read consistency requires that whenever a session transaction references tables or views in query, the results from this query are not cached. Furthermore, any (user-defined) functions used in the query have to be *deterministic*, and the query may not contain temporary tables, tables owned by `SYS` or `SYSTEM`, the `CURRVAL` or `NEXTVAL` pseudocolumns, or instantaneous time functions such `SYSDATE` or `SYS_TIMESTAMP`.

There is also an undocumented `MATERIALIZE` hint that causes factored subqueries to be materialized, that is they are stored in a global temporary table that is created (and dropped) on the fly. Whenever a factored subquery is accessed more than once in the same SQL statement, the factored subquery in question is automatically materialized.

You can use the `INLINE` hint on factored subqueries to prevent the optimizer from materializing them. This inlining can be useful when the data of a factored subquery is accessed several times but based on disjoint predicates from the main query that combines these intermediate results with `UNION ALL`. When the factored subquery is materialized, which would be the default behaviour in this case, Oracle cannot push a common predicate into the view because the predicates are disjoint. This means that the factored subquery is evaluated for all possible values, materialized, and only then filtered accordingly. With the `INLINE` hint you can prevent the materialization, which in turn means that Oracle can eliminate partitions, if the underlying tables are partitioned appropriately, or access data through indexes, meaning that it does not have to compute the factored subquery for all values *before* it filters.

Manual Cardinality Estimates

As we have said before, the cardinality is simply the number of rows divided by the number of distinct values (NDV); a rough estimate of the selectivity is $1/NDV$. The cardinality is in all but heavily hinted SQL statements one of the top measures that influences the cost and thus the execution plan to be taken. Consequently, accurate statistics are essential.

The optimizer is exceptionally good at its job, especially if it has all the data it needs. That is also exactly the point: Oracle runs into problems when it either has no information or the information is not representative. A case where Oracle has no real information is when it joins data with the data from a (pipelined) table function. Oracle guesses the cardinality of a (pipelined) table function based on the block size, which is perfectly fine for simple queries. It gets tricky when you join the table function to other database objects, as now the cardinality affects the execution plan. By the way, in case you are not familiar with table functions, you have already seen one example: `SELECT * FROM TABLE( DBMS_XPLAN.DISPLAY )`.

An undocumented yet often used hint to aid Oracle when statistics are unavailable or inaccurate is `CARDINALITY( tab_name_or_alias number_of_rows )`. It instructs the optimizer to treat the integer `number_of_rows` as the cardinality estimate of the table (function) `tab_name_or_alias` without actually checking it.

Whether the `CARDINALITY` hint is safe or rather very dangerous depends on whether you subscribe to Tom Kyte's or Tony Hasler's (p. 479) views. Changing the cardinality estimate is one of the surest ways to affect the execution plan, and, when used without caution and due diligence, can lead to sub-optimal or even horrible execution plans.

Another undocumented hint that serves a similar purpose is `OPT_ESTIMATE( TABLE tab_name_or_alias SCALE_ROWS = scaling_factor )`. You have to supply `tab_name_or_alias` and the `scaling_factor`, which is a correction (or fudge) factor to scale the optimizer's estimates up or down. The cardinality estimate used by the optimizer is thus the original estimate times the scaling factor.

There is also a variation on `OPT_ESTIMATE` that works exactly like `CARDINALITY`: `OPT_ESTIMATE( TABLE tab_name_or_alias ROWS = number_of_rows )`. The main advantage of the `OPT_ESTIMATE` hint is its versatility. We can also use it to specify an estimate of the number of rows returned from a join: `OPT_ESTIMATE( JOIN (tab_name_or_alias, another_tab_name_or_alias) ROWS = number_of_rows )`.

In addition, there is the `DYNAMIC_SAMPLING( tab_name_or_alias sampling_level )` hint for (pipelined) table functions. When you set `sampling_level` to 2 or higher for pipelined table functions, a full sample of the row source is *always* taken.

Alternatively, you can use the extensible optimizer or rely on cardinality feedback, which is also known as statistics feedback. For cardinality feedback it is important to note that on 11gR2, the feedback was lost once the statement departed from the shared pool. From 12c onwards, the cardinality feedback is still available in the `SYSAUX` tablespace.

Distributed Queries

Distributed queries access data from at least one remote data source. To decrease overall I/O and thus improve the performance of the execution of a SQL statement, you want to minimize the amount of data to be moved around, especially across the database link. With `DRIVING_SITE( tab_name_or_alias )` you tell the optimizer to use the database in which `tab_name_or_alias` resides as the location to do all operations in; all data that is required to execute the statement is moved to that database through database links emanating from the initiating (local) database to the remote data sources. This hint may be required because the local database may not have access to statistics on the remote site(s). Oracle only chooses a remote database without the `DRIVING_SITE` hint when *all* the row sources are at that site.

You typically use this hint to instruct Oracle to choose the database with the largest amount of data as the driving site. What you have to be aware of are user-defined PL/SQL functions that are on a different site than the driving site, as they cause a sizeable performance hit because of data ping-pong. Similarly, beware of sort operations on the final result set as they are taken care of by the local database. Ian Hellström has described some of the issues with distributed queries in more detail.

Join Transformations

Sometimes Oracle can eliminate a join when querying from views. This can happen when you query a view that joins two or more tables but you only ask for data from one of the tables involved in the view's join. Oracle can automatically do a join elimination in these cases but it is also able to do so when referential integrity (i.e. a foreign-key constraint) guarantees that it is OK to do so.

For instance, the child table is the one we're mainly interested in but we would also like to have data from the parent table that is linked to the child table's data by means of a foreign key. Oracle now *knows* that it can simply obtain the information from the child table because referential integrity guarantees that any reference to the parent's column(s) can be replaced by a corresponding reference to the child's column(s). What often cause Oracle to miss referential integrity constraints and thus the join elimination are aggregate functions, as it may not be clear to the optimizer that each row in the child table has exactly one matching row in the parent table. If that is the case, it may often help to rewrite the join such that Oracle can be sure that the integrity is preserved: a simple left-join of child table to its parent will do the trick.

When you have ensured referential integrity with a foreign-key constraint, a join elimination (default) can be forced with the `ELIMINATE_JOIN( tab_name_or_alias )` or disabled with `NO_ELIMINATE_JOIN ( tab_name_or_alias )`. The parameter `tab_name_or_alias` can be either a (parent) table or alias thereof, or a space-separated list of tables or aliases thereof.

There are also instances when an outer join can be transformed to an inner join without affecting the result set because of `IS NOT NULL`-predicates on the columns in the independent (parent) table, which are referred to by the dependent (child) tables in foreign-key constraints. Oracle does this automatically but it can be enabled (disabled) with `OUTER_JOIN_TO_INNER( tab_name_or_alias )` (`NO_OUTER_JOIN_TO_INNER(`

`tab_name_or_alias`)). Again, the parameter `tab_name_or_alias` can be either a (parent) table or alias thereof, or a space-separated list of tables or aliases thereof.

There is an analogous hint for a conversion from a full outer join to an outer join: `FULL_OUTER_JOIN_TO_OUTER( tab_name_or_alias )`, where `tab_name_or_alias` is the (parent) table with a `IS NOT NULL`-predicate (or similar).

The last transformation that we wish to discuss in this group is the semi-to-inner join transformation with its hints `SEMI_TO_INNER( tab_name_or_alias )` and `NO_SEMI_TO_INNER( tab_name_or_alias )`. It applies to subqueries in `EXISTS`-predicates and it causes the nested (correlated) subqueries to be joined to the main query as separate inline views.

How is this different from subquery unnesting? Good question! After a subquery has been unnested, the previously nested subquery always becomes the probe row source. With a semi-to-inner join transformation this subquery can also be used as the driving row source.

Predicate and Subquery Push-Downs

The `PUSH_PRED( tab_name_or_alias )` hint can be used to push a join predicate into an inline view, thus making the inline view a correlated subquery. As a consequence, the subquery must be evaluated for each row of the main query, which may not sound like a good idea until you realize that it enables the optimizer to access the base tables and views in the inline view through indexes in nested loops.

When the main query returns many rows this transformation rarely leads to an optimal plan. The optimizer typically considers this transformation based on cost but if you believe the optimizer to be mistaken in its decision to discard this transformation, you can provide the hint.

A join predicate push-down (JPPD) transformation can be applied when the inline view is one of the following:

- A `UNION [ ALL ]` view.
- An outer-joined, anti-joined, or semi-joined view.
- A `DISTINCT` view.
- A `GROUP BY` view.

Compare this to the list of when view merging is not possible:

- When the view contains any of the following constructs:
  - an outer join;
  - set operators (e.g. `UNION ALL`);
  - `CONNECT BY`;
  - `DISTINCT`;
  - `GROUP BY`.
- When the view appears on the right-hand side of an anti- or semi-join.
- When the view contains scalar subqueries in the `SELECT`-list.
- When the outer query block contains PL/SQL functions.

When using the `PUSH_PRED` hint you also have to supply `NO_MERGE` to prevent the inline view from being merged into the main query, although – as you can see from the aforementioned criteria – view merging and JPPD are generally mutually exclusive. Notably absent from the list of inline views that allow a JPPD is the inner join, which means that *if* you believe a JPPD to be favourable *and* the optimizer does not already consider it to yield the optimal execution plan, you may have to convert an inner to an outer join, just to allow the JPPD transformation.

The execution plan contains an operation PUSHED PREDICATE when the JPPD is successfully applied. NO_PUSH_PRED does exactly the opposite of PUSH_PRED.

The optimizer can also evaluate non-merged or non-unnested (i.e. nested) subqueries as soon as possible. Usually such subqueries are evaluated as the last step, but it may be useful to favour the subquery earlier in the process, for instance because its evaluation is relatively inexpensive and reduces the overall number of rows considerably. The PUSH_SUBQ hint can be used to that end. It is recommended that you specify the query block as a parameter, because as of 10g this hint can be applied to individual subqueries rather than all subqueries. When you apply the hint to a remote table or a table that is joined with a sort-merge join, it has no effect. There is of course also a NO_PUSH_SUBQ to disable subquery push-downs.

The PRECOMPUTE_SUBQUERY hint is related but not the same; it applies to IN-list subqueries. In fact, it instructs the optimizer to isolate the subquery specified with a global temporary table. The results from this separate execution are then hard-coded into the main query as filter values.

Set-Operation Transformations

Set transformation hints that have been deprecated, such as HASH_XJ, MERGE_XJ, and NL_XJ, where X can be either S for semi-joins or A for anti-joins, are not listed here. One set-operation transformation that appears to have slipped through the cracks of deprecation is the SET_TO_JOIN(@SET$N) with N the identifier of the set. It can be used to transform queries with MINUS and INTERSECT to their equivalents with joins. Without the hint the optimizer *never* considers the set-to-join transformation.

In case the initialization parameter _CONVERT_SET_TO_JOIN has been set, you can use NO_SET_TO_JOIN(@SET$N) to disable the transformation.

2.6.6 SQL Optimization Techniques

Before you start fidgeting with individual SQL statements, it is important to note that hints are probably the last thing you should consider adding when attempting to optimize your code. There are several levels of optimization and it is recommended that you start with the server, then the database instance, and finally go down through the database objects to individual statements. After all, the effect of any changes made to individual statements, particularly hints, may be lost when the database is fine-tuned later on.

As a database developer/architect you may not want to tread the path that leads to the desk of the DBA. Fortunately, there is a bunch of things you can do to improve the runtime performance of your statements:

- Optimize access structures:
  - Database design and normalization.
  - Tables: heap or index-organized tables, and table or indexed clusters.
  - Indexes.
  - Constraints.
  - Materialized views.
  - Partitioning schemes.
  - Statistics, including a comprehensive refresh strategy.
- Rewrite SQL statements:
  - Exclude projections that are not required.
  - Minimize the amount of work done more than once.
  - Factor subqueries that are used multiple times in the same statement.

- Use `EXISTS` instead of `IN` because the former stops processing once it has found a match.
- Use `CASE` and/or `DECODE` to avoid having to scan the same rows over and over again, especially for aggregation functions that act on different subsets of the same data.
- Use analytic functions to do multiple or moving/rolling aggregations with a single pass through the data.
- Avoid scalar subqueries in the `SELECT`-list.
- Use joins instead of subqueries, as it gives the optimizer more room to play around in.
- Say what you mean and pick the right join: if you only need an inner join don't write an outer join.
- Add logically superfluous predicates that may still aid in the search for an optimal execution plan, particularly for outer joins.
- Avoid implicit conversions of data types, especially in the `WHERE` clause.
- Write `WHERE` clause predicates with a close eye on the indexes available, including the leading edge of a composite index.
- Avoid, whenever possible, comparison operators such as `<>`, `NOT IN`, `NOT EXISTS`, and `LIKE` without a leading `'%'` for indexed columns in predicates.
- Do not apply functions on indexed columns in the `WHERE` clause when there is no corresponding function-based index.
- Don't abuse `HAVING` to filter rows *before* aggregating.
- Avoid unnecessary sorts, including when `UNION ALL` rather than `UNION` is applicable.
- Avoid `DISTINCT` unless you have to use it.
- Use PL/SQL, especially packages with stored procedures (and bind variables) and shared cursors to provide a clean interface through which all data requests are handled.
- Add hints once you have determined that it is right and necessary to do so.

The advantage of PL/SQL packages to provide all data to users is that there is, when set up properly, exactly one place where a query is written, and that's the only place where you have to go to to change anything, should you ever wish or need to modify the code. PL/SQL will be in our sights in the next part but suffice to say it is the key to maintainable code on Oracle. Obviously, ad-hoc queries cannot benefit from packages, but at least they profit from having solid access structures, which are of course important to PL/SQL too.

One important thing to keep in mind is that you should always strive to write efficient, legible code, but that premature optimization is not the way to go. Premature optimization involves tinkering with access structures and execution plans; it does not include simplifying, refactoring and rewriting queries in ways that enable Oracle to optimally use the database objects involved.

Rewriting queries with or without hints and studying the corresponding execution plans is tedious and best left for high-impact SQL only: queries that process many rows, have a high number of buffer gets, require many disk reads, consume a lot of memory or CPU time, perform many sorts, and/or are executed frequently. You can identify such queries from the dynamic performance views. Whatever you, the database developer, do, be consistent and document your findings, so that all developers on your team may benefit from your experiences.

CHAPTER 3

PL/SQL

PL/SQL stands for Procedural Language/Structured Query Language and it is Oracle's extension of SQL that aims to seamlessly process SQL commands. One reason why PL/SQL is so important to database and application developers is that SQL itself offers no robust procedural constructs to apply logical processing to DML statements. Because it has been designed to integrate with SQL it supports the same data types. Stored PL/SQL source code is compiled and saved in the Oracle database. With PL/SQL you have access to common 3GL constructs such as conditional blocks, loops, and exceptions. IBM DB2 supports PL/SQL (as of version 9.7.0) and PostgreSQL has a procedural language called PL/pgSQL that borrows heavily from PL/SQL but is not compliant.

Each PL/SQL unit at schema level is a piece of code that is compiled and at some point executed on the database. Each unit can be one of the following: anonymous block, function, library, package (specification), package body, procedure, trigger, type, or type body.

You can send a PL/SQL block of multiple statements to the database, thereby reducing traffic between the application and the database. Furthermore, all variables in `WHERE` and `VALUES` clauses are turned into bind variables by the PL/SQL compiler. Bind variables are great as they allow SQL statements to be reused, which can improve the performance of the database. We shall talk more about bind variables later (see *Bind Variables*).

Both static and dynamic SQL are supported by PL/SQL. This allows developers to be as flexible as one needs to be: statements can be built on the fly. The availability of collections, bind variables, and cached program modules is key to applications with scalable performance. PL/SQL even supports object-oriented programming by means of abstract data types.

Each unit must at least contain a `BEGIN` and matching `END`, and one executable statement, which may be the `NULL;` statement. The declaration and exception handling sections are optional. For stored PL/SQL units, such as functions, procedures, and packages, you also need a header that uniquely identifies the unit by means of a name and its signature.

Although PL/SQL and SQL are close friends, not all SQL functions are available in PL/SQL, most notably:

- aggregate and analytic functions, e.g. `SUM` and `MAX`
- model functions, e.g. `ITERATION_NUMBER` and `PREVIOUS`
- data mining functions, e.g. `CLUSTER_ID` and `FEATURE_VALUE`
- encoding and decoding functions, e.g. `DECODE` and `DUMP`
- object reference functions, e.g. `REF` and `VALUE`

- XML functions, e.g. `APPENDCHILDXML` and `EXISTSNODE`
- `CUBE_TABLE`
- `DATAOBJ_TO_PARTITION`
- `LNNVL`
- `NVL2`
- `SYS_CONNECT_BY_PATH`
- `SYS_TYPEID`
- `WIDTH_BUCKET`

Most of these functions involve multiple rows, which is a key indicator that it cannot work in pure PL/SQL, which operates on single rows. Of course, you *can* use these functions in SQL statements embedded in PL/SQL units.

This does not mean that PL/SQL can handle only one row at a time: bulk operations, such as `BULK COLLECT` and `FORALL`, are necessary to achieve solid performance when more than one row is involved, as rows are processed in batches. These bulk operations cause fewer context switches between the SQL and PL/SQL engine. Context switches add a slight overhead that can become significant the more data needs to be processed.

SQL statements are often all-or-nothing propositions: if only one row does not satisfy, say, a constraint, the entire data set may be rejected. With PL/SQL you can insert, update, delete, or merge data one row at a time, in batches whose size you can determine, or even all data at once just like in SQL. Because of this fine-grained control and the PL/SQL's exception handling capabilities you can develop applications that provide meaningful information to users in case something goes wrong rather than a cryptic `ORA-00001: unique constraint ... violated` or a equally uninformative `ORA-02290 error: check constraint ... violated`. Similarly, PL/SQL allows database developers to create applications that do exactly the work that is needed and nothing more. After all, as aptly phrased here on Stackoverflow: nothing is faster than work you don't do.

Anyway, time to sink our teeth into PL/SQL...

3.1 Compilation

When a user sends a PL/SQL block from a client to the database server, the PL/SQL compiler translates the code, assuming it is valid, into bytecode. Embedded SQL statements are modified before they are sent to the SQL compiler. For example, `INTO` clauses are removed from `SELECT` statements, local variables are replaced with bind variables, and many identifiers are written in upper case. If the embedded SQL statements contain any PL/SQL function calls, the SQL engine lets the PL/SQL engine take over to complete the calls in case the function result cache does not have the desired data. Once the SQL compiler is done, it hands the execution plan over to the SQL engine to fulfil the request. Meanwhile the *PVM* interprets the PL/SQL bytecode and, once the SQL engine has come up with the results, returns the status (success/failure) to the client session.

Whenever an identifier cannot be resolved by the SQL engine, it escapes, and the PL/SQL engine tries its best to resolve the situation. It first checks the current PL/SQL unit, then the schema, and if that does not yield success, it eventually generates a compilation error.

The *PL/SQL compilation* itself proceeds as follows:

1. Check syntax.
2. Check semantics and security, and resolve names.
3. Generate (and optimize) bytecode (a.k.a. MCode).

PL/SQL is derived from Ada, and uses a variant of DIANA: Distributed Intermediate Annotated Notation for Ada, a tree-structured intermediate language. DIANA is used internally after the syntactic and semantic check of the PL/SQL compilation process. The DIANA code is fed into the bytecode generator for the PL/SQL Virtual Machine (PVM)

or compiled into native machine code (C), depending on the setting of PLSQL_CODE_TYPE. For native compilation a third-party C compiler is required to create the DLLs (Windows) or shared object libraries (Linux). In the final step, the PL/SQL optimizer may, depending on the value of PLSQL_OPTIMIZE_LEVEL, whip your code into shape. What the optimizer can do to your code is described in a white paper.

When the MCode or native C code is ready to be executed, Oracle loads it into either the shared pool (MCode) or the PGA (native). Oracle then checks the EXECUTE privileges and resolves external references. MCode is interpreted and executed by the PL/SQL runtime engine; native code is simply executed.

Both the DIANA and bytecode are stored in the data dictionary. For anonymous blocks the DIANA code is discarded.

3.2 Bind Variables

Oracle Database developers drool when they hear the phrase "bind variables". Well, not literally, but it is often cited as the key to solid application performance.

When SQL statements are littered with literals, the shared pool is likely to be cluttered with many similar statements. When you execute the same statement over and over again but each time with different literals in the WHERE clause, you cause Oracle to parse the statement every time it is executed. In many cases, particularly when the data isn't horribly skewed, the execution plan is the same. Such a hard parse is a waste of valuable resources: the parsing itself costs CPU time and the various cursors bog up the shared pool.

"But what can we do?"

Use bind variables!

Straight from the horse's mouth: "[a] bind variable is a placeholder in a SQL statement that must be replaced with a valid value or value address for the statement to execute successfully. By using bind variables, you can write a SQL statement that accepts inputs or parameters at run time."

That's right, a bind variable is a simple placeholder; it's the equivalent of "insert your literal here". Whenever you change a literal, it does not matter: the statement is effectively the same. Similarly, the execution plan remains as is. Since there is no reason for Oracle to do a hard parse of a statement it still has in the inventory, Oracle merely looks up the plan.

3.2.1 PL/SQL Variables

Bind variables are related to host variables. Host variables are defined in the host or caller, whereas bind variables accept values from the caller to SQL. In PL/SQL the distinction between bind and host variables disappears.

Bind variables can be used in almost any place in PL/SQL with one exception: bind variables in anonymous PL/SQL blocks cannot appear in conditional compilation directives.

Variables in WHERE and VALUES clauses of *static* DML statements are automatically made bind variables in PL/SQL. Before you run off and think that Oracle takes care of everything for you, hang on a minute. That only applies to static statements; the emphasis in the previous sentence was intentional. Dynamic SQL is slightly different.

Let's take a look at five common examples:

```
1  --
2  -- Example 1: typically found in anonymous PL/SQL blocks
3  --
4  SELECT
5    *
6  BULK COLLECT INTO
7    ...
8  FROM
```

```
  fridge
WHERE
  product = 'Beer';                -- hard-coded literal

--
-- Example 2: typically found in anonymous PL/SQL blocks
--
SELECT
  *
BULK COLLECT INTO
  ...
FROM
  fridge
WHERE
  product = :prod;                 -- bind variable

--
-- Example 3: typically found in PL/SQL modules
--
SELECT
  *
BULK COLLECT INTO
  ...
FROM
  fridge
WHERE
  product = product_in;            -- bind variable

--
-- Example 4: typically found in PL/SQL modules
--
EXECUTE IMMEDIATE
  'SELECT * FROM fridge ' ||
  'WHERE product = ''' ||
  product_in || ''''               -- literal: value is embedded in statement
  BULK COLLECT INTO ...
;

--
-- Example 5: typically found in PL/SQL modules
--
EXECUTE IMMEDIATE
  'SELECT * FROM fridge ' ||
  'WHERE product = :prod'          -- bind variable, because...
  BULK COLLECT INTO ...
  USING product_in                 -- ... we substitute rather than embed the value
;
```

Hopefully, no one would build dynamic SQL like that as it is open to SQL injections; the package `DBMS_ASSERT` offers some basic sanity checks on raw (user) input. The code is only shown for the purposes of our demonstration of the various options with regard to literals and bind variables. You can use `DBMS_SQL` as an alternative to dynamically build a SQL statement, but we have decided not to show the code for reasons of brevity.

There is sometimes a good reason to utilize `DBMS_SQL` instead of native dynamic SQL (NDS). NDS has to be parsed every time the statement is executed; for complex statements the overhead can be significant. Parsing can be bypassed with `DBMS_SQL`. For instance, when a statement is executed for different values inside a loop, you just have to place the call to `DBMS_SQL.PARSE` outside of the loop; the calls to `DBMS_SQL.BIND_VARIABLE` need to be placed inside the loop.

3.2.2 Bind Peeking

If bind variables are so grand, why not enable them by default, everywhere?

The problem lies in what is referred to as bind peeking. When Oracle encounters a statement with bind variables for the very first time, it looks at the literals supplied, checks the histogram (if available), and then fixes the execution plan.

In itself this seems reasonable, right? When data is highly skewed that may become an issue. Let's go back to our `fridge` table and fill it in accordance with the appetite of many developers: beer, beer, and some more beer. Because we have a few guests over tonight we also buy some white wine, salad, and avocados; please, don't ask why. We have created a histogram too: 95% of the rows are related to beer, whereas the remaining 5% are split among the three newly acquired products.

When we send our friendly household robot to run to the kitchen to get beer, and the query contains a bind variable for product, a full table scan will be used. The next time we send it to get white wine and it still uses a full table scan even though an index lookup would be much faster. Why does it do a full table scan? For the first execution our database robot peeked at the bind variable's value, and because its value was `'Beer'`, the execution plan was fixed with a full table scan. The second execution did not care about the fact that we wanted `'White Wine'` since the robot had already decided that the execution plan involved a full table scan.

The reverse situation is also possible. An index scan is used based on the initial execution of the request for something other than beer, even though a full table scan is much more advantageous when we desire beer. Thus, as stated by Arup Nanda, "smart SQL coders will choose when to break the cardinal rule of using bind variables, employing literals instead."

3.2.3 Adaptive Cursor Sharing and Adaptive Execution Plans

That's not the end of the story though. As of 11g, Oracle has introduced the concept of adaptive cursor sharing. Based on the performance of a SQL statement, the execution plan may be marked for revision the next time the statement is executed, even when the underlying statistics have not changed at all.

In `v$sql` this is indicated by the columns `is_bind_sensitive` and `is_bind_aware`. The former indicates that a particular `sql_id` is a candidate for adaptive cursor sharing, whereas the latter means that Oracle acts on the information it has gathered about the cursor and alters the execution plan.

Problematic is that adaptive cursor sharing can only lead to an improved plan *after* the SQL statement has tanked at least once. You can bypass the initial monitoring by supplying the `BIND_AWARE` hint: it instructs the database that the query is bind sensitive and adaptive cursor sharing should be used from the very first execution onwards. A prerequisite for the hint to be used is that the bind variables only appear in the `WHERE` clause and an applicable histogram is available. The hint may improve the performance but you should be aware that it's rarely the answer in the case of *generic static statements*, which we describe below. The `NO_BIND_AWARE` hint does exactly the opposite: it disables bind-aware cursor sharing.

Frequency histograms are important to adaptive cursor sharing. The problem is that they are expensive to compute, unstable when sampled, and the statistics have to be collected at the right time. In Oracle Database 12c, the speed with which histograms are collected has been greatly improved.

Adaptive cursor sharing has a slight overhead though, as explained by the Oracle Optimizer team: additional cursor memory, more soft and hard parses, and more child cursors. The last one may cause cursor cache contention.

The default setting for the `CURSOR_SHARING` parameter is `'EXACT'`. You can also set it to `'FORCE'` (11g and 12c) or `'SIMILAR'` (11g). These settings are, however, generally recommended only as a temporary measure. Oracle's own recommendation boils down to the following: `'FORCE'` is only used by lazy developers who cannot be bothered to use bind variables.

Note: Oracle *never* replaces literals in the `ORDER BY` clause because the ordinal notation affects the execution plan: cursors with different column numbers in the `ORDER BY` cannot be shared.

Adaptive execution plans were introduced in 12c. When we talked about execution plans we already mentioned the *mechanics*: they allow execution plans to be modified on the fly. On a development or test system adaptive cursor sharing and adaptive execution plans may mask underlying problems that need to be investigated and solved before the code hits production, which is why they should be switched off. There are even some people who believe that these features have no place in a production system either because once you have determined the optimal execution plan, it should not be changed, lest you run into unexpected surprises. As such, untested execution plans should never be released into the wild, according to Connor McDonald and Tony Hasler.

3.2.4 Generic Static Statements

Developers sometimes write static SQL statements that are very generic in the sense that they accept different sets of parameters that are bound at runtime; some may be supplied, others may not. The reason people go for such static code rather than a dynamic solution may be based on the misguided idea that dynamic SQL is slow. This is what is referred to by Markus Winand as smart logic, probably because developers think they have created a nice, generic template.

An example might be as follows:

```
SELECT
  manufacturer
, product
, temperature
, MIN(expiry_date) AS min_expiry_date
, MAX(expiry_date) AS max_expiry_date
, COUNT(*)              AS num_items
FROM
  fridge
WHERE
    ( product      = :prod OR :prod IS NULL )
AND ( temperature  = :temp OR :temp IS NULL )
AND ( manufacturer = :manu OR :manu IS NULL )
GROUP BY
  manufacturer
, product
, temperature
;
```

In itself the query looks nice: it elegantly deals with many different cases thanks to the use of bind variables. Swell!

The genericness of such a SQL statement is a fabulous from a coder's point of view. From a performance perspective it is a tremendous weakness. The reason is that the execution plan is the same in all instances.

Suppose for example that there is an index on `manufacturer` and `product`.

Question 1: Will the index be used when the manufacturer of items in the fridge is given?

Answer 1: Unfortunately, no.

An index scan may be much more beneficial but Oracle does not know that. There are 8 possible combinations of bind parameter values and there is no way that Oracle can capture the best plan in each of these cases with a single execution plan, especially since some predicates can benefit from the index whereas other cannot. Hence, Oracle decides to do a full table scan and filter rows.

Question 2: Will the index be used when both manufacturer (`:manu`) and product (`:prod`) are provided?

Answer 2: Nope.

An `OR` confounds index use. Apart from that, we have already established that Oracle does not have a customized execution plan for this case, so that it reverts to a full table scan anyway.

Question 3: Will an index skip scan be used when only the product is supplied at runtime?

Answer 3: Again, no.

Conclusion: Oracle will *never* use an index for our query. The issue with such generic pieces of static code is that the one-size-fits-all code leads to a one-plan-suits-(almost)-none situation. The only cases where full table scans are *always* appropriate are when either only the temperature is known at runtime or all items from the fridge need to be returned. Since `temperature` is not in our index it cannot benefit from an the index anyway. For some special values of `manufacturer` and `product` a full table scan may also be the best solution, but it is doubtful that this applies to all possible cases.

The solution is to use dynamic SQL with bind variables and separate each case. Alternatively, you can write a function that executes different SQL statements with bind variables based on the input of the function. This entails that you will have a few more execution plans in the shared pool, but they are at least tuned to each instance rather than bad for all.

3.3 Loops, Cursors, and Bulk Operations

Oracle generously provides a list of things developers can do to tune their PL/SQL code. One item from that list is probably the single best tool developers have at their disposal to supercharge their code: bulk processing.

Whenever you need to retrieve and modify more than one row of a database table you have a few options:

- A single SQL statement.
- A cursor `FOR` loop to iterate through the results one row at a time to insert, update and/or delete records.
- Retrieve and temporarily store all records in a collection (`BULK COLLECT`), and process the rows
  - one at a time with a `FOR` loop;
  - in bulk with `FORALL`.
- Retrieve all records in batches with an explicit cursor and store the results in a collection (`BULK COLLECT ... LIMIT`), after which the rows are processed
  - one at a time with a `FOR` loop;
  - in bulk with `FORALL`.

Even though it is possible to use a single SQL statement to modify data in a table, PL/SQL developers rarely take this approach. One of the primary reasons against such a single statement is that it is an all-or-nothing proposition: if anything goes wrong all modifications are rolled back. With PL/SQL you have more control over what happens when exceptions occur.

The advantage of `FORALL` is that the statements are sent (in batches) from PL/SQL to the SQL engine in one go, thereby minimizing the number of *context switches*. A disadvantage is that with `FORALL` you can only throw one DML statement over the fence to the SQL engine, and the only differences allowed are in the `VALUES` and `WHERE` clauses of the modification statement. Beware that when multiple `INSERT` statements are sent to the SQL statement executor with `FORALL`, any statement-level triggers will fire only once: either before all statements have been executed or after all statements have completed.

`BULK COLLECT` does for queries what `FORALL` does for *IUD statements*; `MERGE` is supported from Oracle Database 11g onwards. Since collections are required in `BULK COLLECT` and `FORALL` statements, and (pipelined) table functions, where they can greatly improve the runtime performance, we take a moment to go through the fundamentals.

3.3.1 Collections

PL/SQL has three *homogeneous* one-dimensional collection types: associative arrays (PL/SQL or index-by tables), nested tables, and variable-size or varying arrays (varrays). Homogeneous refers to the fact that the data elements in a collection all have the same data type. Collections may be nested to simulate multi-dimensional data structures, but these are currently not supported with the traditional syntax you may be familiar with from other programming languages, such as C# or Java.

Summarized in the table below are the distinguishing features of each of these three collection types, where we have omitted `[ CREATE [ OR REPLACE ] ] TYPE type_identifier IS ...` from the declarations:

Collection	Type declaration	Index	Sparse	Dense	Persistent	Initialization	EXTEND	TRIM	MULTISET
Nested table	`TABLE OF data_type`	positive integer	Yes*	Yes	Yes	Yes	Yes	Yes	Yes
Associative array	`TABLE OF data_type INDEX BY ix_data_type`	`ix_data_type`	Yes	Yes	No	No	No	No	No
Variable-size array	`VARRAY( max_num_elems ) OF data_type`	positive integer	No	Yes	Yes	Yes	Yes	Yes	No

Here, `ix_data_type` can be either a `BINARY_INTEGER`, any of its subtypes, or a `VARCHAR2`. Associative arrays are thus the only collection type that can handle *negative* and *non-integer* index values.

Note: When it comes to performance, the difference between integers and small strings (i.e. fewer than 100 characters) as indexes is minimal. For large strings the overhead of hashing can be quite significant, as demonstrated by Steven Feuerstein and Bill Pribyl.

We have added an asterisk to the 'Yes' in the column 'Sparse' for nested tables because technically they can be sparse, although in practice they are often dense; they only become sparse when elements in the middle are deleted after they have been inserted. The only collection type that can be used in PL/SQL blocks but neither in SQL statements nor as the data type of database columns is an associative array. Although both nested tables and varrays can be stored in database columns, they are stored differently. Nested table columns are stored in a separate table and are intended for 'large' collections, whereas varray columns are stored in the same table and thought to be best at handling 'small' arrays.

Nested tables and variable-size arrays require initialization with the default constructor function (i.e. with the same identifier as the type), with or without arguments. All collections support the `DELETE` method to remove all or specific elements; the latter only applies to nested tables and associative arrays though. You cannot `DELETE` non-leading or non-trailing elements from a varray, as that would make it sparse.

As you can see, the `TRIM` method is only available to nested tables and varrays; it can only be used to remove elements from the back of the collection. Notwithstanding this restriction, associative arrays are by far the most common PL/SQL collection type, followed by nested tables. Variable-size arrays are fairly rare because they require the developer to know in advance the maximum number of elements.

Note: Oracle does not recommend using `DELETE` and `TRIM` on the same collection, as the results may be counter-intuitive: `DELETE` removes an element but retains its placeholder, whereas `TRIM` removes the placeholder too. Running `TRIM` on a previously deleted element causes a deleted element to be deleted. Again.

The built-in DBMS_SQL package contains a couple of collection shortcuts, for instance: NUMBER_TABLE and VARCHAR2_TABLE. These are nothing but associative arrays indexed by a BINARY_INTEGER based on the respective data types.

To iterate through a collection you have two options:

- A numeric FOR loop, which is appropriate for *dense* collections when the entire collection needs to be scanned: `FOR ix IN l_coll.FIRST .. l_coll.LAST LOOP ...`.
- A WHILE loop, which is appropriate for sparse collections or when there is a termination condition based on the collection's elements: `WHILE ( l_coll IS NOT NULL ) LOOP ...`.

When using the FORALL statement on a sparse collection, the INDICES OF or VALUES OF option of the bounds clause may prove equally useful.

Now that we have covered the basics of collections, let's go back to the performance benefits of bulk processing.

3.3.2 Performance Comparisons

There are quite a few performance comparisons documented on the internet and in books. We shall not provide our own as that would be mainly a repetition and thus a waste of time. Instead, we try and bring together all the information, so that you, the database developer, can benefit from what others have already figured out. Our service is in gathering the information, so that you don't have to wade through all the sources yourself.

3.3.2.1 Explicit vs Implicit Cursors

The discussion on whether explicit cursors (i.e. with OPEN-FETCH-CLOSE) are always to be preferred to implicit ones stems from an era that has been rendered obsolete by Oracle. The performance of explicit cursors in all but prehistoric Oracle versions is more or less on par with that of implicit cursors. In fact, sometimes an implicit cursor can be faster than an explicit one, even though it does more work behind the scenes.

Apart from that, a developer should *always* be wary of experts claiming that A is always to be preferred to B, especially when that advice is based on comparisons done on previous versions — yes, we are fully aware of the fact that our advice is reminiscent of the liar's paradox. Companies like Oracle continue to invest in their products, and features that were once considered slower but more convenient are often improved upon to make them at least as fast as the traditional approaches.

We pause to remark that the LIMIT clause is part of the FETCH statement and thus only available to explicit cursors.

What happens in the lifetime of a cursor?

1. Open: Private memory is allocated.
2. Parse: The SQL statement is parsed.
3. Bind: Placeholders are replaced by actual values (i.e. literals).
4. Execute: The SQL engine executes the statement and sets the record pointer to the very first record of the result set.
5. Fetch: A record from the current position of the record pointer is obtained after which the record pointer is incremented.
6. Close: Private memory is de-allocated and released back to the SGA.

All steps are done in the background for implicit cursors. For explicit cursors, the first four stages are included in the OPEN statement.

3.3.2.2 The Impact of Context Switches

'Are context switches really such a big deal?'

We could argue that adding little bits of overhead to each DML statement inside a cursor `FOR` loop, which — as we have seen just now — can be based on either an explicit or implicit cursor, that iterates over a large data set can quickly become a huge performance problem. However, such an argument does not measure up to actual numbers.

A simple `FORALL` is often a whole order of magnitude faster than a cursor `FOR` loop. In particular, for tables with 50,000-100,000 rows, the runtime of a `FORALL` statement is typically 5-10% of that of a cursor `FOR` loop. We have consistently found at least an order of magnitude difference with a comparison script of the PL/SQL Oracle User Group for table inserts of up to a million rows. For a million rows the speed-up was closer to 25 than 10 though.

With these results it seems to make sense to break up a cursor `FOR` loop when the number of separate IUD statements for each iteration is less than 10, which for most practical purposes implies that it is a good idea to use `FORALL` in almost all cases. After all, with a 10x runtime improvement per IUD statement you need at least 10 individual statements per iteration to arrive at the same performance as with a single cursor `FOR` loop. To avoid too bold a statement we rephrase it as that it is always a good idea to at least compare the performance of your `FOR` loops to an IUD `FORALL`.

Similarly, a `BULK COLLECT` obviates many context switches, thereby reducing the execution time. Something that is important to keep in mind, though, is that filling a collection requires sufficient memory in the PGA. As the number of simultaneous connections grows, so do the requirements on the overall PGA of the *database instance*.

Ideally you'd figure out how much PGA memory you can consume and set the `LIMIT` clause accordingly. However, in many situations you do not want to or cannot compute that number, which is why it is nice to know that a value of at least 25 has been shown to improve the performance significantly. Beyond that, the number is pretty much dependent on how much of a memory hog you want your collection to be. Note that the `LIMIT` clause does not control (i.e. constrain) the PGA, it merely attempts to manage it more effectively. In addition, the initialization parameter `PGA_AGGREGATE_TARGET` does not insulate your database from issues with allocating huge collections: the `LIMIT` option really is important.

The difference among various `LIMIT` values may be negligible when it comes to the runtime, in particular when there is enough PGA available, but it is noticeable when you look at the memory consumed. When there isn't enough memory available and you do not specify the `LIMIT`, PGA swapping can cause the performance to degrade as well. Beware!

For various tables with more than a few thousand records, a value between 100 and 1,000 for `LIMIT`'s value seems to be an apt initial choice. Above 1,000 records the PGA consumption grows considerably and may cause scalability issues. Below that value the benefit may be too small to notice, especially with smaller tables.

3.3.2.3 Table Functions

When multiple rows are inserted with individual `INSERT ... VALUES` statement, Oracle generates more redo than when using a single `INSERT ... SELECT`. This can lead to major differences in the overall runtime performance. So, how can we benefit from set-based SQL rather than row-based PL/SQL when `FORALL` is not an option?

In Chapter 21 of Oracle PL/SQL Programming Feuerstein and Pribyl discuss the benefits of `BULK COLLECT-FORALL` (BCLFA) versus pipelined table functions (PTF). Their example borrows heavily from Oracle's own stock ticker example. The essence of the problem is to retrieve data from a table, transform it, and then insert the records into another table. The table function is merely used to take advantage of set-based SQL: `INSERT INTO ... SELECT ... FROM TABLE( tab_func(...) )`.

The situation described can admittedly be handled with a classical `BULK COLLECT ... LIMIT` (BCL) and `FORALL` combination, but it could easily be extended to insert the data into multiple tables, which makes a simple single `FORALL ... INSERT` statement impossible.

So, what happens when we run the various versions? Here's an overview of several runs on 12c, where we show the improvement factors in the execution time and redo generation compared to the baseline of the FOR loop to fetch the data and a pipelined table function to insert it:

Method	Execution	Redo
FOR loop FETCH and INSERT	+3.5	+7.3
FOR loop FETCH → PTF INSERT	0	0
BCLFA	-3.2	(+7.3)
BCL FETCH → PTF INSERT	-2.7	(0)
BCL FETCH → parallel PTF INSERT	-5.3	-1500

Notice the parentheses around the redo information for BCLFA and the BCL-PTF combination. These numbers are typically close to the ones for the FOR loop (+7.3, i.e. a more than sevenfold increase in the amount of redo generated) and FOR-PTF combination (0, i.e. no improvement at all), respectively. The reason is that redo is obviously generated for IUD statements and in these cases the INSERT statements are identical to the ones mentioned. Any differences are due to what comes before: a SELECT can generate redo too due to block clean-outs. The effect of block clean-outs is most obvious directly *after* IUD statements that affect many blocks in the database; the effect is usually relatively small. So, depending on how you sequence your comparisons and what you do in-between, the numbers may be slightly different or even exactly the same.

These values obviously depend on the values used for the LIMIT clause in relation to the number of rows to be inserted, and the degree of parallelism, at least for the last entry. It is clear that a parallel INSERT with a pipelined table function is the most efficient alternative. The reason the redo generation is so low for that combination is that parallel inserts are *direct-path inserts*. For direct-path inserts, redo logging can be disabled.

Even without the parallel PTF INSERT, the BCL is responsible for a threefold decrease of the execution time. What is also obvious is that the cursor FOR loop is by far the worst option.

In the valuable yet footnote-heavy Doing SQL from PL/SQL, Bryn Llewellyn notes a factor of 3 difference in the runtime performance between all DBMS_SQL calls inside a FOR loop and the same construct but with OPEN_CURSOR, PARSE, DEFINE_COLUMN, and CLOSE_CURSOR outside of the loop on a test data set of 11,000 rows. The difference is obviously in favour of the latter alternative. A simple cursor FOR loop is about twice as fast as the best DBMS_SQL option, with no noticeable difference when the single DML statement inside the loop is replaced with an EXECUTE IMMEDIATE (NDS) solution.

An upsert (i.e. INSERT if record does not exist, otherwise UPDATE) is best done with a MERGE rather than a PL/SQL-based solution with UPDATE ... SET ROW ... that uses a DUP_VAL_ON_INDEX exception to handle the update in case the entry for the primary key already exists, as the MERGE runs noticeably faster.

What should you take from all this?

Well, a cursor FOR loop is pretty much the worst choice, and it should only be a method of last resort. Even though the same iteration can be used to extract, transform, and load the data one row at a time, it is slower than fetching it in bulk with BULK COLLECT ... LIMIT , then modifying it, and finally bulk-unloading it with FORALL .

When you are dealing with queries that return multiple records or rows, always use BULK COLLECT ... LIMIT (BCL). In case you are faced with IUD statements and whenever a simple BCLFA is possible it is probably your best shot at getting a considerable performance improvement. If, however, you require complex transformations and have multiple IUD statements, then a parallelized PTF may further drive down the cost of running your application logic. Pipelined table functions are also a good choice when you are concerned about redo generation.

3.3.3 Caveats

We have already mentioned that FORALL can only be used when one DML statement needs to be executed. Unfortunately, that's not where the bad news ends.

Any *unhandled* exception causes *all* IUD changes to be rolled back. The exception to this is when exceptions are managed with the `SAVE EXCEPTIONS` clause. If there are failures that are saved, Oracle will raise a single exception (`ORA-24381`) for the entire statement after completing the `FORALL`. You can query the pseudo-collection `SQL%BULK_EXCEPTIONS` to get the information about these exceptions.

Note: Any collection index referenced in the IUD statement of a `FORALL` cannot be an expression: it must be the plain index itself. The collection index is implicitly defined by Oracle as a `PLS_INTEGER`.

Another gotcha is that bulk SQL can only be performed on local tables: you cannot do a `BULK COLLECT` over a database connection. Furthermore, parallel DML is disabled for bulk SQL.

As of Oracle Database 10gR2, the PL/SQL compiler automatically optimizes most cursor `FOR` loops into constructs that run with performance comparable to a `BULK COLLECT`. Yay! Unfortunately, this does *not* happen automatically when there are IUD statements in your `FORALL` statements: these statements require manual intervention. Boo!

For *pipelined* table functions we can define the collection types they return inside a package specification. Oracle automatically creates database (i.e. schema-level) types based on the record and collection types defined in the package specification. The reason Oracle does this is that table functions are invoked by the SQL engine, which does not know about PL/SQL constructs inside the package. As such, Oracle must ensure that the SQL engine can do its work, so it creates the shadow types implicitly. These types typically have names that start with `SYS_PLSQL_`. So, while you have all your types in the same file (i.e. package specification), which makes maintaining your code base easier, you end up with system-generated identifiers for types, which in itself can be a maintenance issue.

3.4 Caching

A cache — pronounced like cash (i.e. money) — is the computer's equivalent of scrap paper to jot down information you may need to look up later. The piece of paper is usually a fairly limited portion of memory in which the computer stores the information, which can be either computed values or even complete copies of data blocks, so it can find what it's after faster without having to redo the computation or look up the data from disk (again).

Oracle Database has three main methods for caching: one where the developer must do all the work and two that can be switched on and off with relative ease.

Of course, there is no such thing as a free lunch and that is true of caching as well. You cannot cache everything 'just in case', because your memory is limited and the cache needs to be managed too. In other words, there is an overhead associated with function caching in Oracle. Nevertheless, enabling caching can lead to significant performance benefits that are well worth the memory sacrificed.

Interested to learn more about function caching in PL/SQL? Then read on!

3.4.1 Side Effects

Before we talk about caching it is important to mention one important related topic: side effects.

When a subprogram (i.e. named PL/SQL block) changes only its own local variables, everything is just peachy. However, when a subprogram modifies variables outside its scope we speak of side effects. What this means for Oracle databases is that subprograms that are free of side effects cannot change global variables, public variables in packages, database tables, the database itself, external states, or their own `OUT` or `IN OUT` parameters.

What the heck is an external state? Well, you can call `DBMS_OUTPUT` to send messages to a user's client or send an email with `UTL_MAIL`. That's the type of thing that is meant by an external state.

In some programming circles, side effects are seen as evil as they do not confirm to the model of *pure functions*[1]. In essence, they consider side effects in medicine and computer science to be cut from the same cloth: they do something that is not intended and are generally seen to cause harm.

From a database developer's perspective that view is a bit too restrictive. Changing table data is pretty much what a database is for, and you'd rather modify data through a PL/SQL API of your own making than type in the necessary IUD statements all day long.

Why care about side effects? Side effects can prevent parallelization of queries, yield order-dependent results, or require package states to be maintained across user sessions. When it comes to caching, they can throw a spanner in the works. It is thus good to know about side effects: the fewer side effects, the better your chances at using built-in caching mechanisms to supercharge your PL/SQL code!

Notes

3.4.2 Alternatives

Oracle offers a few alternatives for caching: deterministic functions, package-level variables, and the result cache (for functions). Each of these techniques has its pros and cons.

Package-level variables are conceptually the simplest but require the developer to think about and manage the cache. A package-level variable is really what it says on the box: a variable in a package that contains static (reference) data that needs to be accessed many times.

Suppose you have an expensive deterministic function that you need to execute many times. You can run the function for different parameter values and store the combination of input-output as a key-value pair in a collection at the package level. You can thus access the data without having to recompute it every time.

Similarly, you can store (pre-aggregated) data from a table that does not change while you work with it, so you don't have to go back and forth between the disk and the PGA. Alternatively, you can create a just-in-time cache, where each time a new value or row is requested, you check the cache and return the data (if already present) or add it to the cache (if not yet available in the cache) for future use. That way, you do not create a huge collection up front but improve the lookup performance of subsequent data access requests or computations.

Another common use case for package-level variables that has little to do with caching though is when people try to avoid the dreaded mutating table error with triggers.

A problem with package-level variables is that they are stored in the *UGA*. This means that the memory requirements go up as there are more sessions. Of course, you can add the `SERIALLY_REUSABLE` directive to the package to reduce storage requirements, but it also means that the data needs to be exactly the same across sessions, which is rare, especially in-between transactions.

You do *not* want to use package-level variables as caches when the data contained in them changes frequently during a session or requires too much memory. A measly 10 MB per user quickly becomes 1 GB when there are 100 sessions, making a what seemed to be a smart idea a veritable memory hog.

In terms of speed, packaged-based caching beats the competition, although the function result cache is a close second. But, as we have mentioned, package-based caching is also fairly limited in its uses and requires a developer to think carefully about the caching mechanism itself and the overall memory requirements. Moreover, on Oracle Database 11g and above, the built-in alternatives are almost always a better option.

[1] A pure function is both idempotent and free of side effects. Note that idempotence enjoys different definitions in mathematics and functional programming versus and the rest of computer science. In mathematics, it means that for a function $f\colon X \to Y$ and $x \in X$, $f\,(f(x)) = f(x)$. In other words, we can *apply* the function as many times as we like but the result will be the same every time. The identity map $x \mapsto 1$ is a perfect example of an idempotent function in mathematics. In computer science, an idempotent function is one that can be *executed* many times without affecting the result.

3.4.2.1 DETERMINISTIC Functions

The DETERMINISTIC clause for functions is ideal for functions that do not have any non-deterministic components. This means that each time you provide the function with the same parameter *values*, the result is the same. When you define a function you can simply add the DETERMINISTIC option to the declaration section, making sure that the function (or any functions or procedures it calls) does not depend on the state of session variables or schema objects as the results may vary across invocations. This option instructs the optimizer that it may use a cached result whenever it encounters a previously calculated result.

Any speed-up from specifying a function as DETERMINISTIC becomes apparent when you execute the same function many times with the same parameter values *in the same SQL statement*. A typical example is a user-defined conversion function that is called for each row of the result set of a query

Function-based indexes can only use functions marked DETERMINISTIC. The same goes for materialized views with REFRESH FAST or ENABLE QUERY REWRITE.

One restriction is that you cannot define a nested function as deterministic.

Whenever a function is free of side effects and deterministic, it is a good practice to add the DETERMINISTIC option. Whether the optimizer makes use of that information is entirely up to Oracle, but fellow coders will know that you intended to create a deterministic, side-effect free function, even when Oracle does not see a use for that information.

'What happens when I label a function as DETERMINISTIC but it secretly isn't?'

First off, why on earth would you do that?! Second, Oracle cannot fix stupidity. Its capability to discover non-deterministic DETERMINISTIC functions is not just limited, it is non-existent. Yes, that's right: Oracle does not check whether you are telling the truth. It trusts you implicitly.

What can happen, though, is that a user sees dirty (i.e. uncommitted) data because the incorrectly-marked-as-deterministic function queries data from a table that has been cached inappropriately. Neil Chandler has written about the multi-version concurrency control model (MVCC) used by Oracle Database, which explains the roles of isolation levels, REDO, and UNDO with regard to dirty reads, in case you are interested. If you tell Oracle that a function is deterministic even though it is not, then the results can be unpredictable.

3.4.2.2 The RESULT_CACHE Option

As of Oracle Database 11g, the function result cache has entered the caching fray. It offers the benefits of just-in-time package-level caching (and more!) but without the hassle. All you have to do is add the RESULT_CACHE option to the function declaration section and that's it. It couldn't be much easier!

The function result cache is ideal for data from tables that are queried from more frequently than they are modified, for instance lookup tables and materialized views (between refreshes). When a table's data changes every few seconds or so, the function result cache may hurt performance as Oracle needs to fill and clear the cache before the data can be used many times. On the other hand, when a table (or materialized view) changes, say, every 10 minutes or more, but it is queried from hundreds or even thousands of times in the meantime, it can be beneficial to cache the data with the RESULT_CACHE option. Recursive functions, small lookup functions, and user-defined functions that are called repeatedly but do not fetch data from the database are also ideal candidates for the function result cache.

With Oracle Database 11gR2, the RELIES ON clause is deprecated, which means that you don't even have to list the dependencies: Oracle will figure everything out for you!

The database does *not* cache SQL statements contained in your function. It 'merely' caches input values and the corresponding data from the RETURN clause.

Oracle manages the function result cache in the *SGA*. In the background. Whenever changes are committed to tables that the cache relies on, Oracle automatically invalidates the cache. Subsequent calls to the function cause the cache to be repopulated. Analogously, Oracle ages out cached results whenever the system needs more memory, so you, the database developer, are completely relieved of the burden of designing, developing, and managing the cache.

Since the function result cache is in the SGA rather than the *PGA*, it is somewhat slower than PGA-based caching. However, if you have hidden `SELECT` statements in functions, the SGA lookups thanks to the function result cache beat any non-cached solutions with context switches hands down.

Sounds too good to be true?

It is.

First, the function result cache only applies to stored functions not functions defined in the declaration section of anonymous blocks. Second, the function cannot be a pipelined table function. Third, the function cannot query from data dictionary views, temporary tables, `SYS`-owned tables, sequences, or call any non-deterministic PL/SQL function. Moreover, pseudo-columns (e.g. `LEVEL` and `ROWNUM`) are prohibited as are `SYSDATE` and similar time, context, language (NLS), or GUID functions. The function has to be free of side effects, that is, it can only have `IN` parameters; `IN OUT` and `OUT` parameters are not allowed. Finally, `IN` parameters cannot be a `LOB`, `REF CURSOR`, collection, object type, or record. The `RETURN` type can likewise be none of the following: `LOB`, `REF CURSOR`, an object type, or a record or collection that contains a `LOB`, `REF CURSOR`, and/or an object type.

The time to look up data from the function result cache is on par with a context switch or a function call. So, if a PL/SQL function is almost trivial *and* called from SQL, for instance a simple concatenation of `first_name` and `last_name`, then the function result cache solution may be slower than the same *uncached* function.

Inlining, or rather hard coding, of simple business rules seems to be even faster as demonstrated by Adrian Billington, although we hopefully all agree that hard coding is a bad practice, so we shall not dwell on these results and pretend they never existed.

Beware that the execution plan of a SQL statement does not inform you that a function result cache can or even will be used in clear contrast to the query result cache. The reason is both simple and obvious: `RESULT_CACHE` is a PL/SQL directive and thus not known to the SQL engine.

Latches

The result cache is protected by a single *latch*, the so-called result cache (RC) latch. Since latches are serialization devices, they typically stand in the way of scalability, especially in environments with a high degree of concurrency, such as OLTP applications.

Because there is only one latch on the result cache, only one session can effectively create fresh result cache entries at any given moment. A high rate of simultaneous changes to the result cache are therefore detrimental to the performance of a database. Similarly, setting the parameter `RESULT_CACHE_MODE` to `FORCE` is a guarantee to bring a database to its knees, as every single SQL statement will be blocking the RC latch.

Scalability issues have dramatically improved from 11gR1 to 11gR2, but latch contention still remains an issue when rapidly creating result sets in the cache.

It should be clear that the function result cache only makes sense for relatively small result sets, expensive SQL statements that do not experience high rates of concurrent execution, and SQL code that is against relatively static tables.

IR vs DR Units

The default mode of PL/SQL units is to run with the definer's rights (DR). Such units can benefit from the function result cache without further ado. Invoker's rights (IR) subprograms, created with the `AUTHID CURRENT_USER` rather than `AUTHID DEFINER`, cannot use the function result cache, and an attempt at compilation leads to a `PLS-00999` error, at least prior to DB12c. The reason is that a user would have been able to retrieve data cached by another user, to which the person who originally requested the data should not have access because its privileges are not sufficient.

This restriction has been lifted with 12c, and the security implications have been resolved. The solution to the security conundrum is that the function result cache is per user for IR units. This means of course that the RESULT_CACHE option is only useful for functions that the same user calls many times with the same input values.

Memory Consumption

That's all very nice, but how much memory does the function result cache gobble up?

A DBA can run EXEC DBMS_RESULT_CACHE.MEMORY_REPORT(detailed => true) to see detailed information about the memory consumption. However, the purpose of these pages is to help fellow developers to learn about optimization techniques, which means that DBMS_RESULT_CACHE is out of the question.

You can check the UGA and PGA memory consumption by looking at the data for your session from the following query:

```
SELECT
  *
FROM
  v$sesstat
NATURAL JOIN
  v$statname
;
```

You can provide the name of the statistic you're interested in. A full list of statistics can be found in the official documentation. For example, 'session uga memory' or 'session pga memory'. These are current values, so you'd check the metrics *before* and *after* you run your function a couple of times to see the PGA and UGA memory consumption of your function. Obviously, there will be no (or very little) PGA consumption in the case of the function result cache.

There are also several solutions available that calculate the various statistics for you. They typically work by checking the metrics before running a function several times, then run the function, after which they check the metrics again.

In case you need help configuring the function result cache, here's a helping hand.

3.4.2.3 DETERMINISTIC vs RESULT_CACHE

A common question with caching is whether the DETERMINISTIC option or the RESULT_CACHE is best. As always, the answer is: 'It depends.'

When you call a deterministic function many times from within the *same* SQL statement, the RESULT_CACHE does not add much to what the DETERMINISTIC option already covers. Since a single SQL statement is executed from only one session, the function result cache cannot help with multi-session caching as there is nothing to share across sessions. As we have said, marking a deterministic function as DETERMINISTIC is a good idea in any case.

When you call a deterministic function many times from *different* SQL statements — in potentially different sessions or even instances of a RAC — and even PL/SQL blocks, the RESULT_CACHE does have benefits. Now, Oracle can access a single source of cached data across statements, subprograms, sessions, or even application cluster instances.

The 'single source of cached data' is of course only true for DR units. For IR units, the function result cache is user-specific, which probably dampens your euphoria regarding the function result cache somewhat. Nevertheless, both caching mechanisms are completely handled by Oracle Database. All you have to do is add a simple DETERMINISTIC and/or RESULT_CACHE to a function's definition.

3.4.3 The UDF Pragma

The PRAGMA UDF compiler directive is not a caching mechanism. 'So, what's it doing in the chapter on caching?' we hear you ask.

It's an optimization technique for subprograms, so it fits in nicely into our current discussion. It tells the compiler that the function ought to be prepared for execution from SQL statements. Because of that information, Oracle can sometimes reduce the cost of context switches.

As of Oracle Database 12c, there is also the possibility of adding a PL/SQL function to your SQL statement with the WITH clause. A non-trivial example is described on Databaseline, from which it follows that the WITH clause is marginally faster than the UDF pragma, but that the latter has the advantage that it is modular, whereas the former is the equivalent of hard coding your functions.

We can therefore recommend that you first try to add PRAGMA UDF to your PL/SQL functions *if and only if* they are called from SQL statements but not PL/SQL code. If that does not provide a significant benefit, then try the WITH function block.

3.4.4 The NOCOPY Directive: To Pass By Value or Reference?

In many programming languages you can pass parameters by value or reference, which can be as different as night and day in terms of performance. The reason is simple: a large object that is passed by value needs to be copied, which may take a considerable amount of time, whereas a reference (i.e. pointer) to that object is about as cheap as chips.

When you specify formal parameters to subprograms you cannot only define IN, OUT, and IN OUT, but also NOCOPY. What this does is *ask* the compiler to pass the actual parameters by reference rather than value. Please note the emphasis on 'ask': it is not a directive, it is merely a request.

IN parameters are *always* passed by reference. When a parameter is passed by reference, the formal and actual parameters refer to the same memory location. Any changes to the value of the formal parameter are immediately reflected in the actual parameter as well. Aliasing is not an issue for IN parameters, because subprograms cannot assign values to them.

OUT and IN OUT parameters can be passed by reference or value. Hence, aliasing can become an issue, which means that assignments may or may not show up immediately in all parameters, especially when the same parameter is passed by reference as well as value in different places: when a pass-by-value parameter is copied back to the original data structure that is also passed by reference elsewhere, the modifications will overwrite any changes already done by the passed-by-reference subprograms. Moreover, if the subprogram raises an exception, there is no way to undo or rollback the changes made to the data structure, because there is no copy of it from before all the modifications.

The NOCOPY hint can thus help you reduce unnecessary CPU cycles and memory consumption: by adding NOCOPY you request that the compiler only pass the memory address rather than the entire data structure.

There are several cases where you can be sure the compiler ignores NOCOPY:

- The actual parameter must be *implicitly* converted to the data type of the formal parameter.
- The actual parameter is the element of a collection.
- The actual parameter is a scalar variable with a NOT NULL constraint.
- The actual parameter is a scalar numeric variable with a range, size, or precision constraint.
- The actual *and* formal parameters are records.
- The actual and/or formal parameters are declared with %ROWTYPE or %TYPE, and the constraints differ.
- The code is called through a database link or external program.

In summary, you may see a performance boost from using NOCOPY but be careful: when you run into issues, the original data structure may be corrupted!

CHAPTER 4

Data Modelling

Data modelling is typically the responsibility of database architects and to some degree DBAs. The architects design the overall, high-level logical model and the administrators may give their opinion on the physical model, as they are probably the most knowledgeable when it comes to the internals of the technology to be deployed.

You may thus be wondering why data modelling pops up as a separate topic, as these pages clearly focus on optimization techniques for database *developers*, not architects or administrators.

The thing is, althrough some organizations are large enough to separate responsibilities as outlined, many software teams do not have the sheer size to split roles among members. In some teams, especially DevOps, the various members may even need to assume various roles as required. Moreover, there are a few topics regarding data modelling that are also of interest to developers, especially when it comes to performance.

One such topic is *partitioning*. It is intimately tied to indexing, which is definitely a task for developers, as we have *already talked about*.

Another topic is data compression. Anyone who has ever needed to send a large file via email knows that zipping or compressing it can make a huge difference. The same is of course true of the data stored in an Oracle Database, or any database for that matter. Fortunately, Oracle offers several options for data compression for both data and indexes. We shall look at compression techniques because it is important you choose the right one for the task. After all, there is a performance trade-off: because the data gobbles up less space on the hard drive, the CPU has to do more work when adding or modifying data because it needs to compress and decompress the data on the fly.

Anyway, let's quit yappin' and get right to business!

4.1 Partitioning

Partitioning is often the key to improved application performance for very large databases (VLDBs). The reason is simple: the data is split into chunks and each partition behaves a little like a table itself: it has its own name and storage characteristics. Each partition is managed and operated independently from all other partitions of the same table. Physically, the data for different partitions is stored in separate segments; logically, the various partitions still make up the entire table and SQL statements still look the same as before. Moreover, no segments are allocated for the table itself, which is nothing but a collection of partitions.

The main advantages of partitioning that Oracle lists are:

- increased availability
- easier administration of schema objects
- reduced contention for shared resources in OLTP systems
- enhanced query performance in data warehouses

A common scenario in a data warehouse is to run queries against a table for a particular period of (historical) time. Creating a (range) partition on the time column of such a table may thus enable you to query the table in a more efficient way, as Oracle only has to access one particular partition rather than the whole table.

There are three partitioning strategies:

1. Range partitioning: rows are partitioned based on ranges of the partition key, which can be one or more columns of a table. A prime example of such a strategy is for time ranges.
2. List partitioning: rows are partitioned based on a (discrete) list of partition key values. A use case of list partitioning is for identifier values, for instance countries or regions.
3. Hash partitioning: rows are partitioned with a hashing algorithm; the hash function determines in which partition a particular row ends up. High update contention in OLTP applications can be resolved with hash partitioning as the *IUD* operations can be for different segments; I/O bottlenecks can be reduced or avoided because the data is randomly distributed across partitions.

The first two partitioning strategies require you to define names for each partition and perhaps use the `ALTER TABLE ... ADD PARTITION ...` statement to add more partitions as time progresses. The third partitioning strategy requires you to define the partition key and the number of partitions; Oracle takes care of the rest. Partition names for hash-partitioned tables are automatically generated.

A table can be either partitioned by one of the three partitioning strategies or by a combination of these. In the former case we speak about single-level partitioning. Composite partitioning is what the latter case is called because each partition is subdivided into a subpartition. For example, a table can be partitioned on the `SALES_DATE` and then subpartitioned on `SALES_REGION`. When you typically only run queries for, say, a time range of a month and a particular sales region, these queries may return the result set a lot faster as the data is contained in one subpartition.

A caveat of partitioning is that a non-partitioned table cannot be made into a partitioned table with a simple `ALTER TABLE` statement. You have to recreate the entire table.

Heap-organized (i.e. 'normal') tables can be stored in compressed format, which may be beneficial to query execution speed, not to mention storage requirements. Partition compression is typically only useful in OLAP systems, such as data warehouses, where IUD operations are much rarer than queries.

Note that virtual columns can be used to partition a table on. What is not allowed, though, are PL/SQL function calls in the definition of the virtual column.

4.1.1 Partitioned Indexes

Indexes of partitioned tables can be partitioned too. A **local partitioned index** is an index of a partitioned table that is partitioned on the same columns. Each index partition is associated with one table partition, hence the name 'local'. OLAP environments are prime candidates for local partitioned indexes. When data in a certain partition is updated or deleted, only the data in a single index partition is affected; only index partitions have to be rebuilt or maintained. Local partitioned index are intimately tied to the table partitions to which they belong: you cannot add or remove an index partition. Instead, you have to modify the table itself; the index will follow.

Note that bitmap indexes can *only* be local partitioned indexes of a partitioned table. Global bitmap indexes are only available to non-partitioned tables.

Local partitioned indexes can be either **local prefixed indexes** or **local non-prefixed indexes**. When the partition key is on the leading edge of the indexed column list we have a local prefixed index. Queries with predicates on the index

can always use partition elimination or *partition pruning*, which reduces I/O. Local non-prefixed index cannot in all instances eliminate partitions as the partition key is not on the leading edge of the index.

Non-prefixed indexes are often a good idea for historical information. Let's go back to our classical fridge example and the household robot. The fridge contains all products that your robot can retrieve at any given time. You now want to program the robot to to throw away all products that have been in the fridge for more than a year. However, you also want fast access to your produce based on the product. What you can do is range-partition your fridge based on the purchase date, so that you can drop the partition with the oldest stuff independently of the others. With a non-prefixed index on the product identifier you can benefit from a quick lookup. To support the fast dropping of the oldest partition and its associated index, you can partition the non-prefixed index on the purchase date too.

If that example sounds too weird for you, replace fridge with sales, purchase date with sales date, and product with account number. That ought to do it!

A **global partitioned index** is a B-tree index that is partitioned independently of its table. Global partitioned indexes are ideal for OLTP systems, where *IUD* performance is critical. It should be obvious that a global partitioned index is harder to maintain as it may span many partitions. For example, when a single table partition is dropped, the entire global index is affected. What is more, a global index must be recreated in its entirety even when only a single partition is recovered to a point in time; a local index does not have this issue. Global partitioned indexes can only be prefixed.

Oracle will automatically use multiple parallel processes for fast full index scans and multi-partition range scans, so that each process reads a local index partition. This does not apply to SQL statements that probe individual rows.

Please note that an index on a partitioned table may be global and non-partitioned. Such indexes can become an issue when data a partition is truncated and the index becomes unusable. Rebuilding such a global non-partitioned index can take a long time when the underlying table is huge. Global non-partitioned indexes are most common in data warehousing environments to enforce primary key constraints.

4.1.2 Caveats

There are a couple of impossible situations with partitioned indexes. First, you cannot have a local non-prefixed unique index. Why? Well, to check uniqueness Oracle would have to check every index partition, which may be prohibitively slow. Unique indexes can only be global or local prefixed, unless the index contains a subset of the partition key.

Second, we have already hinted at it: you cannot have a global non-prefixed index. Why? Well, you can simply create a concatenated global index on the columns that you want to index on and partition by. Suppose you want to partition the fridge's index on product by expiry date; this would be useful for queries that look for a particular products and a range of expiry dates. Queries with only the expiry date would have to search each partition of such a global index, which amounts to an index skip scan of the non-partitioned index, where the leading edge is the product. Therefore, a global non-prefixed index would not give you anything that a concatenated index on product and expiry date does not already cover. Oracle has thus done us all a favour by supplying fewer moving parts.

Third, there are no global partitioned bitmap indexes. Concurrent DML causes entire blocks of a global bitmap index to be locked, which in turn means locks on potentially many, many rows. Sure, bitmap indexes are typically recommended for data warehouses, which sort of goes against the idea of global partitioned indexes, which are the go-to indexes for partitioned tables in OLTP systems. Global partitioned bitmap indexes would offer no advantages while only adding complexity and performance hits.

4.1.3 Recommendations

The following are rough guidelines that are intended to aid the developer in identifying the best partitioning strategy.

4.1.3.1 Single-Level Partitioning

Historical tables that are frequently scanned by range predicates and/or have a rolling window of data are ideal for range/interval partitioning.

Hash partitions are best for tables with heavy I/O, where the partition key is (almost) unique, so that the load can be distributed over several partitions. The number of partitions is typically a power of two.

List partitioning is best used when there is a *single column* that is always searched for in `WHERE` filters and when the data must be mapped to different partitions based on a set of discrete values of that column.

4.1.3.2 Composite Partitioning

Composite **range-hash** partitioning is common for huge tables that have historical data that are frequently joined with other large tables. The main partition of range-hash partitioning enables historical analyses, such as long-running queries based on range predicates that can benefit from partition pruning, whereas the hash subpartitioning strategy allows *full or partial partition-wise joins* to be performed. Similarly, tables that use hash partitioning but also need to maintain a rolling window are prime candidates for range-hash partitioning.

Composite **range-list** partitioning is best for historical data that is accessed by multiple dimensions. One common access path is by, say, date, whereas another one is by country or region, or another discrete attribute. The reverse, composite **list-range** partitioning, is the go-to partitioning scheme when the primary dimension that is used to access data is not a range or interval but a list.

Another option is composite **range-range** partitioning, which is ideal for tables with multiple time dimensions. An example of multiple time dimensions is common in retail where one dimension lists when orders were placed and another one when they were shipped, invoiced, or paid. Similarly, composite **list-list** partitioning is recommended for tables that are accessed by multiple discrete dimensions.

Yet another compositing partitioning strategy is **list-hash** partitioning. The business case for this strategy is when typically the data is accessed through a the discrete (list) dimension but hashing is required to take advantage of parallel partition-wise joins that are done against another dimension that is (sub-)partitioned in the table(s) to be joined to the original one.

4.1.3.3 Prefixed vs Non-Prefixed Local Indexes

Oracle recommends that either prefixed local or global partitioned indexes be used for OLTP applications, as they minimize the number of index partition probes. Non-prefixed local indexes should be avoided in OLTP environments. DSS or OLAP systems benefit mostly from local partitioned indexes, whereas global partitioned indexes are not really a great idea in these scenarios. Non-prefixed local indexes are ideal for historical databases, especially when individual partitions need to be dropped regularly.

4.1.3.4 Partitioned vs Non-Partitioned Global Indexes

When all the rows of a particular partition are (almost) always asked for, an index won't us do any good; the partition key is enough for Oracle to take advantage of partition pruning. In fact, the overhead of maintaining the index may cause unnecessary slowdowns when doing *IUD* operations. When the partition key is absent from queries but we do filter for local index columns, Oracle needs to scan all partitions and use the local indexes to return the requested result set. In such cases a global index is to be preferred. Whether a partitioned or non-partitioned global index is best depends on the queries you run. If the index can be partitioned such that most queries only need to access a single partition, then partitioning is likely your best choice. If not, you should default to a non-partitioned global index.

4.2 Compression

Some compression options are not available to all database versions. Oracle Advanced Compression needs to be purchased as a separate option. Moreover, a few compression methods are not even available to the enterprise edition, only to Oracle Exadata.

Although there is more to Oracle's Advanced Compression add-on, we shall focus on pure data compression and not `SECUREFILE` LOB deduplication or network compression although both are important in their own right.

Oracle promises customers a storage space reduction factor of two to four with advanced row compression. So, if your database is running low on hard drive space and you desperatly need to make some space for more data, you might benefit from advanced row compression. Sure, we recommend you take these numbers with a grain of salt as they are listed in company propaganda pamphlets, but still: compression will reduce the storage required to save the data. How much obviously depends on the contents of the data.

There is no such thing as a free lunch or compression without a catch. It's not as bad as you may think though: Oracle is able to *read* compressed blocks directly, that is, without having to decompress them. This means that there is *no* performance hit when reading data. Sweet!

So, the main problem lies with modifying data, and even there Oracle has stuffed a bunch of tricks up its sleeve. Blocks are in fact decompressed in batch mode, which means that a freshly initialized block remains uncompressed until an internally controlled threshold is reached. Once the threshold is reached, Oracle compresses the block. When more data is dumped into the block, the same game is played again: once the threshold is reached, the *entire* block is recompressed to achieve the greatest savings in storage.

So, when would you consider using advanced row compression?

In data warehouses or OLAP environments compression makes a lot of sense. Most operations are read-only and they touch upon lots of data. Since reading does not cause additional I/O, it seems almost silly not to compress the data. In fact, table scans can be performed more efficiently because compressed blocks Oracle are smaller than uncompressed blocks.

In OLTP databases the situation is quite similar although some care needs to be taken. Remember that blocks are compressed and decompressed *in batches*, so that no single row-level transaction has a performance hit. The CPU usage will spike when the compression algorithm is triggered but that typically happens after many single-row transactions.

Advanced row compression is not the only kid on the block. Hybrid columnar compression (HCC) is available in Oracle Exadata, and it uses a combination of column- and row-level compression because values may be repeated many times *across* rows and *within* columns rather than rows.

Before we go into specifics and discuss performance considerations, let's take a step back and see how Oracle's compression algorithms work.

4.2.1 Compression Methods

Let's take a look at the different compression methods Oracle offers.

4.2.1.1 `BASIC` and `OLTP`

Important to note is that for `BASIC` and `OLTP` Oracle deduplicates data in rows in one block rather than compression in the classical sense. Oracle can internally rearrange the order of the columns to combine deduplication symbols and save more disk space.

Because of this Oracle does not 'decompress' the blocks, it simply reconstructs the original rows from reading the row bits (with the symbols) and combining that information with the symbol directory that contains the token or symbol values. This back-and-forth may cause significant CPU time, although it may be offset against a reduction in the number of physical reads because the same data is available in a smaller number of blocks.

When data is deleted from a table with `BASIC` compression, Oracle must maintain the symbol table. Once a symbol has no references any longer, the symbol itself is removed.

Tables defined with `BASIC` compression automatically have `PCTFREE` set to zero. In fact, whenever you `MOVE` the table, Oracle resets the value to zero. So, when you update column values and Oracle expands the symbols, there typically is not much space. This in turn means that when you run many updates on the same block, Oracle may have to migrate rows, which causes a performance degradation. An appropriate level of `PCTFREE`, which has to be reset every time you move tables, may alleviate the situation.

Another problem with `UPDATE` statements on tables with basic compression is that Oracle works on a copy of the row that has the column values modified and fully expanded, that is, decompressed. To add insult to injury, Oracle does not recompress these modified rows, irrespective of the presence of a suitable symbol in the directory.

Although `OLTP` ought to solve the `UPDATE` problem of the basic compression, it does not.

Important to note is that a standard `INSERT` does *not* compress data even when the target table has compression enabled. Only CTAS statements, direct-path or parallel inserts, and `ALTER TABLE tab_name COMPRESS ...` with a subsequent `ALTER TABLE tab_name MOVE` cause fresh data to become stored in compressed format.

4.2.1.2 Hybrid Columnar Compression

Advanced row compression looks in each row and creates smaller symbols that represent repeated values in the same row. Sure, this may be useful but let's not kid ourselves: in real databases the number of rows where, say, the first name of an employee matches the city he or she lives in is fairly limited.

That observation is the kick-off for hybrid columnar compression. Since typically values are repeated in the same column, it makes more sense to create symbols for columns rather than rows. That is exactly what HCC does. Instead of having a symbol directory for each block, Oracle can even get away with a single symbol directory for each column.

```
QUERY { LOW | HIGH }
```

These two HCC options are ideal for data warehouses. The 'low' setting is best suited for databases where the ETL is a critical factor, whereas 'high' is mainly reserved for situations where the space savings are paramount.

```
ARCHIVE { LOW | HIGH }
```

Whenever data needs to be archived but retrieval is not really a main concern, for instance for historical data that is needed for compliance but otherwise not really interesting to the business, the archival compression options are a good choice. The 'low' and 'high' options follow the same logic as before: if loading the data is a performance concern, then lowering the compression rate may be give you an edge in the ETL processes, otherwise go for glory with the highest available compression to save most on disk space.

4.2.2 Performance Considerations

Although size reduction is the main (and only) reason to switch on compression, it makes sense to look at the various option and how they affect the load times.

4.2.2.1 Size Reduction

Below is table with the size reduction compared to a simple uncompressed table. These numbers have been compiled from results published by Uwe Hesse (A), Yannick Jaquier (B), Arup Nanda (C), and Talip Hakan Öztürk (D). The values marked with an asterisk have not been included in the mean size reduction.

Option	**Size (%)**	Size A (%)	Size B (%)	Size C (%)	Size D (%)
BASIC	**48**	N/A	65.63	47.93	29.41
OLTP	**59**	N/A	78.68	56.26	41.17
QUERY LOW	**19**	14.70	22.47	19.79	N/A
QUERY HIGH	**16**	8.82	13.44	0.52	41.17
ARCHIVE LOW	**10**	6.62	12.81	0.52*	N/A
ARCHIVE HIGH	**8**	4.41	11.13	0.33*	29.41*

The numbers shown depend heavily on the data, so always be sure to try the different options with your own data! The percentages should *only* be read as an indication of the potential storage savings, not the gospel of Saint Oracle. You have been warned.

It's clear that the least compression is offered by OLTP, which is not surprising. Similarly, we see that in most cases the hybrid column compression shaves off most disk space.

4.2.2.2 CPU Overhead

What about data modification operations?

Talip Hakan Öztürk and Uwe Hesse have done extensive tests with the various options. Again, below are the summarized insert times compared to the BASIC option (not shown).

Option	**Insert time (%)**	Time A (%)	Time D (%)
OLTP	**316**	289.4	343.1
QUERY LOW	**69**	70.35	67.73
QUERY HIGH	**117**	147.9	85.40
ARCHIVE LOW	**145**	201.2	88.29
ARCHIVE HIGH	**710**	783.5	636.0

The insert for the OLTP option is a conventional insert because in an OLTP environment a direct-path insert does not make sense. For the other compression methods the inserts are direct-path inserts. Note that conventional inserts generate a lot more undo and redo, which in itself is a cause for concern, especially with regard to performance.

Although there is quite a bit of variation in the insert times compared to the basic compression option, it seems that QUERY LOW achieves both a substantial size and time reduction. The most variation is visible for the hybrid columnar compression methods.

Yannick Jaquier reports that CPU time increases by 6.5 percentage points for the insert into a table with either ARCHIVE HIGH or QUERY HIGH compression as compared to the uncompressed case. However, the number of consistent gets for a full table scan are 88% and 90%, and the physical reads are 30% and 39% of that of an uncompressed table, respectively.

Uwe Hesse has similar numbers. A full table scan takes 86.5% for BASIC, 87.0% for OLTP, 41.5% for QUERY LOW, 61.3% for QUERY HIGH, 54.6% for ARCHIVE LOW, and 141.5% for ARCHIVE HIGH of the baseline's time, which is for an uncompressed table.

So, it's true that decompression does not affect read operations. With the exception of ARCHIVE HIGH most compression options do not affect query and insert performance too badly. It is possible (and likely) that the reduced I/O makes queries faster. Nice!

Nevertheless, when you're running an OLTP database, it's probably best that you stay away from compression altogether.

CHAPTER 5

Glossary

access path An access path is a way in which a query retrieves rows from a set of rows returned by a step in an execution plan (row source). Below is a list of possible access paths:

- full table scan
- table access by ROWID
- sample table scan
- index unique scan
- index range scan
- index full scan
- index fast full scan
- index skip scan
- index join scan
- bitmap index single value
- bitmap index range scan
- bitmap merge
- bitmap index range scan
- cluster scan
- hash scan

anonymous block The basic unit of a PL/SQL source program is a block. Each block must consist of at least one executable statement. The declaration and exception-handling sections are optional. Whenever a block is unnamed, it is known as an anonymous block. Anonymous blocks are compiled each time they are loaded into memory:

1. Syntax check: check PL/SQL syntax and generate a parse tree.
2. Semantic check: check types and process the parse tree.

3. Code generation.

CGA The Call Global Area is a part of the *PGA*. It exists only for the duration of a call, that is, only as long as the process is running. Anonymous blocks and data for top-level modules (i.e. functions and procedures) are in the CGA. Package-level data is saved in the *UGA*.

context switch Whenever control is passed from either the SQL engine to the PL/SQL engine or the other way round we speak of a context switch. Procedural code is handled by the PL/SQL engine, whereas all SQL is handled by the SQL statement executor, or SQL engine. When processing control is exchanged, Oracle needs to store the process state of the executing thread before transferring control to another thread or process. The overhead of context switching may be significant, especially when looping through lots of data. Bulk binds (i.e. `BULK COLLECT INTO`) can be used to reduce the number of context switches when looping through data.

A common mistake is to use a `SELECT ... INTO ... FROM dual` in a PL/SQL assignment to obtain a value that can be obtained directly in PL/SQL too. Such an assignment requires two context switches: from PL/SQL to SQL and back from SQL to PL/SQL.

instance A database instance, or instance for short, is a combination of *SGA* and background processes. An instance is associated with exactly one database. Oracle allocates a memory area and starts background processes as soon as an instance is started.

IUD statements A subset of DML statements in SQL: IUD is an acronym of `INSERT`, `UPDATE`, and `DELETE`. The `MERGE` statement is a combination of these and thus included too, although it is not explicitly listed in the acronym.

Please note that some people (e.g. Steven Feuerstein) do *not* consider the `SELECT` to be part of DML, probably because the M refers to 'modification' and a `SELECT` does not modify data. This is in contrast to the generally accepted definition. We do not encourage such abuse of terminology as it is likely to lead to confusion. `SELECT` statements are commonly referred to as queries.

latch A latch is a mutual exclusion (mutex) device. A process that requires access to a certain data structure that is latched (i.e. locked) tries to access the data structure intermittently (like a spinlock) until it obtains the information.

Compared to *locks*, latches are lightweight serialization devices. Latches are commonly found in memory structures, such as the SGA data structures.

A common gripe with contention for shared pool latches is hard parsing. One way to reduce contention for latches in the shared pool is to eliminate literals by using bind variables as much as possible or set the `CURSOR_SHARING` parameter appropriately, although the former is recommended.

lock A lock is a mutual exclusion (mutex) device. A process enqueues until its request can be fulfilled in a first-come-first-serve (FCFS) manner. Compared to *latches*, locks are a heavyweight synchronization mechanism. Row-level locks are a primary example.

PGA Program Global Area: non-shared memory region that contains data and control information for a server process. The PGA is created when the server process is started. The PGA consists of the following areas that may or may not exist in every case:

- SQL work area(s):
  - sort area
  - hash area
  - bitmap merge area
- session memory
- private SQL area:
  - persistent area

- runtime area

A SQL work area is used for memory-intensive operations.

The private SQL area is the combination of the persistent and runtime areas. The runtime area contains information about the state of the query execution, whereas the persistent area holds bind variable values. A cursor is the name of a specific private SQL area, which is why they are sometimes used interchangeably.

The session memory is shared rather than private for shared server connections. Similarly, the persistent area is located in the *SGA* for shared server connections.

PL/SQL optimizer Since Oracle Database 10g the PL/SQL compiler can optimize PL/SQL code before it is translated to system code. The optimizer setting `PLSQL_OPTIMIZER_LEVEL` — 0, 1, 2 (default), or 3 — determines the level of optimization. The higher the value, the more time it takes to compile objects, although the difference is usually hardly noticeable and worth the extra time.

processes There are two types of processes: Oracle processes and client processes. A client process executes application or Oracle code. Oracle processes come in three flavours: server, background processes, and slave processes. A server process is one that communicates with a client processes and the database to fulfil a request:

- Parse and execute SQL statements;
- Execute PL/SQL code;
- Read data blocks from data files into the buffer cache;
- Return results.

Server processes can be either dedicated or shared. When the client connection is associated with one server process, we speak of a dedicated server connection.

In shared server connections, clients connect to a dispatcher process rather than directly to a server process. The dispatcher receives requests and places them into the request queue in the large pool (see *SGA*). Requests are handled in a first-in-first-out (FIFO) manner. Afterwards, the dispatcher places the results in the response queue.

Background processes are automatically created when an instance starts. They take care of for example maintenance and recovery (redo) tasks. Mandatory background processes include:

- PMON: process monitor process;
- LREG: listener registration process;
- SMON: system monitor process;
- DBW: database writer process;
- LGWR: log writer process;
- CKPT: checkpoint process;
- MMON/MMNL: manageability monitor process;
- RECO: recoverer process.

Slave processes are background processes that perform actions on behalf of other processes. Parallel execution (PX) server processes are a classical example of slave processes.

PVM The PL/SQL vitual machine (PVM) is a database component that executes a PL/SQL program's bytecode. Inside the VM the bytecode is translated to machine code that is executed on the database. The intermediate bytecode (or MCode for machine code) is stored in the data dictionary and interpreted at runtime.

Native compilation is a different beast altogether. When using PL/SQL native compilation, the PL/SQL code is compiled into machine-native code that bypasses the interpretation at runtime. The translation of PL/SQL code into a shared C library requires a C compiler; these shared libraries are not portable.

SARGable Search ARGumentable.

SGA The System Global Area contains data and control information. It consists of the shared pool, the database buffer cache, the redo log buffer, the Java pool, the streams pool, the in-memory column store, the fixed SGA, and the optional large pool. It is shared by all server and background processes. The so-called large pool is an optional area in the SGA that is intended for memory allocations that are larger than is appropriate for the shared pool. This is used to avoid memory fragmentation.

shared pool The shared pool is an area in the *SGA* that consists of the library cache, the data dictionary cache, the server result cache, and the reserved pool. The library cache holds the shared SQL area, and, in the case of a shared server connection, also the private SQL areas.

SQL compiler The SQL compiler consists of the parser, the optimizer, and the row source generator. It "compiles SQL statements into a shared cursor", where a cursor is simply a "handle or name for a private SQL area in the *PGA*". A private SQL area "holds a parsed statement and other information, such as bind variable values, query execution state information, and query execution work areas".

UGA The User Global Area is the memory associated per user session. Package-level data is stored in the UGA, which means that it grows linearly with each new user session. When the state of a package is serially reusable (`PRAGMA SERIALLY_REUSABLE`), the package data is stored in the *SGA* and persists for the life of the server call. Non-reusable package states remain for the life of a session.

VPD A virtual private database (VPD) allows security policies to be created that enable/disable access to columns or rows. It is a fine-grained security control system on individual objects and the data contained within rather than based on the schema (user) level.

Bibliography

[Codd69] *Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks*, E.F. Codd, IBM Research Report, 1969.

[Codd70] *A relational model of data for large shared data banks*, E.F. Codd, Communication of the ACM, Vol. 13, pp. 377—387, 1970. DOI: 10.1145/362384.362685.

Index