Leonid Nossov
Hanno Ernst
Victor Chupis

Formal SQL Tuning for Oracle Databases

Practical Efficiency - Efficient Practice

www.allitebooks.com

Formal SQL Tuning for Oracle Databases

www.allitebooks.com

www.allitebooks.com

Leonid Nossov • Hanno Ernst •
Victor Chupis

Formal SQL Tuning for Oracle Databases

Practical Efficiency - Efficient Practice

www.allitebooks.com

Leonid Nossov
ORACLE Deutschland B.V. & Co. KG
München, Germany

Hanno Ernst
T-SYSTEMS INTERNATIONAL GMBH
Bamberg, Germany

Victor Chupis
Vodafone GmbH
Düsseldorf, Germany

Translation from the German language edition “Formales SQL-Tuning für Oracle-Datenbanken”. © Springer-Verlag 2016.
Translated from the German by Jane Scorah and David Thackray.
Graphical Illustrations by Anna Nosova.

ISBN 978-3-662-50416-1 ISBN 978-3-662-50417-8 (eBook)
DOI 10.1007/978-3-662-50417-8

Library of Congress Control Number: 2016950905

© Springer-Verlag Berlin Heidelberg 2016
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer-Verlag GmbH Berlin Heidelberg

www.allitebooks.com

I know that I know nothing
Socrates

www.allitebooks.com

www.allitebooks.com

Foreword by Watson

The mantra I chant relentlessly is "you must understand your data; you must understand your queries." By this I mean that if a developer has a complete understanding of the information he is processing, he will be able to write code that will run efficiently. The critical decisions that Oracle's cost-based optimizer must make are join order, join method, and access method. Use of structures such as DISTINCT to remove duplicates when none exist, or OUTER JOIN when INNER JOIN will do, or needless functions around predicate columns, or inappropriate use of NULL will force the CBO to develop plans that do not make the best decisions and therefore cripple performance. There are more complex issues such as whether views are mergeable or predicate pushable, whether correlated subqueries can be factored out into common table expressions—any number of other optimizations. The CBO (cost based optimizer) will do its best, but even though it is probably the most complex software artifact with which you will ever work, all it can do is follow rules. It does not have the knowledge of what is actually possible given the business environment that a developer will (ideally) have.

If the developer has a thorough understanding of his data and queries, he will be able to write code that lets the optimizer make the best decisions. However, this knowledge may take an immense amount of time to acquire, and the necessary information may not be available. Certainly an outside consultant on a short-term contract has no chance of gaining it, and all too often the DBA (database adminstrator) cannot do so either. The methodology presented in this book will allow any competent DBA to tune SQL, even if he has minimal knowledge of the environment. That is the power of this method: You do NOT need to have a comprehensive understanding of the data or the SQL. You do not even need to know what the SQL being executed is or be able to rewrite it.

The approach taken in this book is vital to the division of responsibilities in an Oracle environment. Historically, Oracle's approach to SQL tuning was that the DBA should identify the statements that have (or are causing) problems and throw them back to the developer for tuning. However, all too often when asked to tune a statement, a developer would reply "how do I do that?" and as a result many DBAs spend 8 hours a day tuning SQL. In later releases of the database, Uncle Oracle has realized this, and now Oracle's approach appears to be that developers should concentrate on writing code that fulfills a business need and DBAs should be

www.allitebooks.com

responsible for making it run efficiently. The formal method described will help DBAs to do just that.

The step-by-step method will identify points in an execution plan that are problematic for performance and suggest how to tune them. Comprehensive instruction on how to capture, read, and interpret an execution plan is essential for this, and the book delivers this in spades. Essential reading for all DBAs.

Oracle Certified Master DBA John Watson
Director of Database Services
Skillbuilders Inc.
Wakefield
RI, USA
June 2016

www.allitebooks.com

Foreword by Gosejacob

Leonid Nossov's current work deals specifically with the central aspect of performance tuning for Oracle databases, the analysis and acceleration of SQL statements. I was, of course, delighted when Leonid asked me to write a few introductory lines for this book too.

I can well imagine some potential readers being somewhat deterred by the title "Formal SQL Tuning," which sounds very dry and theoretical. In my view, however, it best reflects the nature of the method described. The term "formal tuning" can also be seen here as a synonym for a structural approach.

The current version of the Oracle database provides a comprehensive arsenal of analysis possibilities and tools, which simplify, or even automate, the tuning of SQL statements. However, situations repeatedly occur in which these possibilities can only be exploited to a limited extent or, in some cases, not at all. One of my colleagues jokingly calls this the expert mode. We regularly encounter disbelieving faces when this method achieves amazing runtime improvements. The colleague in question has now been awarded the title Dr. SQL by some clients.

Particularly when swift action and, hopefully, positive results are expected, stress levels for DBAs can increase sharply. From my own experience, I remember situations in which the DBA was trying to solve performance problems in front of the screen at the keyboard while the rest of the room was filled with a dozen people, including the production manager. The consistent formal approach presented in this book can be a lifeline in such a situation.

Leonid and his co-authors succeed in serving up this rather dry material in an easily digestible and appetizing manner in the lively dialogues with Peter Smith, with whom I, as a reader, can easily identify. This is made possible by the use of numerous examples which present the material under consideration in a clear manner and make the relevant information directly accessible to the reader. Possible solutions for the diagnosed problem are supplied at the same time.

www.allitebooks.com

Let me congratulate you Leonid on another fine book, and I wish you the reader an enjoyable reading experience and the time to learn formal SQL tuning at your leisure, so that you can shine with this knowledge when the next critical situation comes up.

Munich, Germany
August 2015

Martin Gosejacob

Foreword by Schwinn

I must admit that this is the first time I have held one of Leonid Nossov's books in my hands. I became aware of this book through my colleague Martin, as I myself am often confronted with questions about SQL tuning or database monitoring. SQL tuning is also "trendy" and has long been a popular topic in the database community. Almost anyone who is anyone in this field has a blog on the subject of SQL tuning. Especially after the release of new database patches or even a new version, there is an increase in the amount of information on the Internet about new optimization methods with associated tips and tricks.

Consequently, I was very curious about this book by Leonid and his co-authors Victor Chupis and Hanno Ernst with the title "Formal SQL Tuning," which was so different from other books and articles I had read. I had no idea what to expect. As a mathematician, I was, of course, familiar with formal methods. Mathematical methods are often regarded as very "dry" and of little interest to nonscientists. I was, therefore, all the more surprised by the relaxed, easily readable "question and answer" style which Leonid uses in his book. This enables one to familiarize oneself with the material quickly and with a minimum of effort. If you really read every chapter—even those designated for beginners—you can even learn how execution plans are interpreted. In this way, database administrators and developers get a chance to gain positive, personal, hands-on experience of tuning tasks even though they are not SQL tuning experts and do not possess a lot of previous knowledge about data modeling. It is then also interesting to read what co-authors Victor Chupis and Hanno Ernst have to say. They put the formal SQL tuning method to the test and successfully solve real-life problems.

The three authors are very successful in focusing on the essentials of SQL tuning. Equipped with formal SQL tuning methodology, one can then calmly and confidently cope with changes to the database due to new optimizer releases or in-house application changes. I hope all readers enjoy this book, which I heartily recommend.

Munich, Germany — Ulrike Schwinn
10 August 2015

Preface

When I wrote the chapter "Formal SQL Tuning" in [1], I never thought that there would be a sequel to it. In my opinion, the method was clear and obvious. For this reason the above-mentioned chapter seemed more than adequate for the understanding and application of the proposed formal method.

As is often the case, things turned out differently in practice. Firstly, this method is not at all common—at least not among the participants of the SQL tuning workshop which I have been conducting for some time. Up to now I have never found anyone who previously knew of this method, although there were enough experienced specialists among the participants. Secondly, this method cannot be quickly understood. I had to use a lot of examples and exercises in order to teach the others formal SQL tuning. Thirdly, this method proved to be very efficient. I have to admit that I prefer manual SQL tuning and don't use Oracle's automatic method. Instead, I have been applying the above-mentioned method for years now, always with success. Some of the workshop participants have said that this method has also been of benefit to them. I have been pleased to hear that it helps them to analyze and solve performance problems quickly.

To my amazement, I found no book which described this method. Perhaps I missed something or perhaps other authors had neglected this simple method. However, I now believe that this method is an important element of practical SQL tuning.

Two excellent database specialists, Victor Chupis and Hanno Ernst, are also of this opinion, and so the idea of writing a book together was born. I wrote the chapters in which formal SQL tuning is described. In contrast to [1], this method is presented here in a much more structured and detailed way. It was also very important for us to present our practical experience. Victor Chupis and Hanno Ernst were responsible for this part of the book.

Dortmund, Germany
25 May 2015

Leonid Nossov

Contents

1 Introduction 1
1.1 Aims and Target Groups 2
1.2 An Overview of the Book 2
1.3 Acknowledgments 3

2 Some Thoughts on the Term "SQL Tuning" 5
2.1 SQL Tuning: Definitions and Objectives 5
2.2 SQL Tuners 6
2.2.1 Oracle 6
2.2.2 Developer 8
2.2.3 Database Administrator 9

3 Minimum Minimorum on the Subject of the "Execution Plan" . . . 11
3.1 Can You Read Execution Plans? 11
3.2 Some Important Details 11
3.2.1 Sections of the Execution Plan 12
3.2.2 Optimizer Estimations and Costs 20
3.2.3 Runtime Statistics 20
3.3 Summary 22

4 Approaches to Formal SQL Tuning 25
4.1 The Objective: Effective SQL Tuning 25
4.2 The Principle: Elimination of "Brakes" in the Execution Plan . . . 26
4.3 The Method: Analysis of Runtime Statistics in the Execution Plan 26
4.4 The Main Criterion: The Runtime Statistic "Cardinality" 27
4.5 The Procedure: An Iterative Process 28
4.6 The Guideline: Tuning Without Changing the SQL Statement . . . 28

5 Bottlenecks in the Execution Plan 31
5.1 "Local" Problems in the Execution Plan 31
5.1.1 A Missing Index 31
5.1.2 A Nonselective Index 35
5.1.3 An Index with a Large Clustering Factor 38

5.1.4 A Sparse Index 39
5.1.5 Nested Loop Join Instead of Hash Join and Vice Versa 43
5.2 "Global" Problems in the Execution Plan 48
5.2.1 Formal Rules for Changing the Table Order in a Join . . . 48
5.2.2 Joins with a Low Number of Hits 53
5.2.3 Joins with a Large Hit Quantity 64
5.3 Summary 70

6 Procedure of Formal SQL Tuning 73

7 Practical Experience with Formal SQL Tuning 79
7.1 Hanno's Experience 79
7.1.1 Statistics on Problem Categories 80
7.1.2 A Small Synthetic Test Case in Respect of a Nonselective Index 81
7.1.3 Practical Example 83
7.2 Victor's Experience 87
7.2.1 The First Practical Example 88
7.2.2 The Second Practical Example 90

8 Closing Remarks 93

Appendix: Application of the Formal Principle for the Analysis of Performance Problems After an Oracle Migration 97

Literature 109

1 Introduction

This chapter contains a brief description of our aims and the readership to which this book is addressed. A brief overview is provided here to help potential readers decide whether to purchase the book. This overview will also provide a useful orientation aid to those who already have a copy. A number of people assisted us in writing this manuscript. We would like to take this opportunity to thank them for their help.

Another important point to mention is that Peter Smith, a friend of Leonid's, was a character in his first book. Several chapters were written in the form of dialogues with Peter. This had the effect of making the dry material a bit more palatable for the reader. His questions, input, and suggestions were extremely helpful in presenting a difficult topic like performance tuning in an easier and more comprehensible manner for the reader. In the meantime, Hanno and Victor have also made friends with Peter. This gave us the idea of inviting Peter Smith to participate again. As he had enjoyed his first performance in front of an audience so much, he agreed to appear again. With the first book, he increased his knowledge of performance tuning. Now it is the turn of SQL tuning. For those who are not yet acquainted with Peter, he would like to introduce himself:

Peter: "As you already know, my name is Peter Smith. I am still working as an Oracle database administrator for a medium-sized enterprise. As a result of my participation in the book on performance tuning, I have developed a keen interest in this topic. I have learned a great deal, and the difficult times when I was confronted with two suboptimally performing databases and didn't know what to do are a thing of the past. Although I have improved my knowledge of SQL tuning, it is, unfortunately, far from adequate. This is what prompted me to take on the role of Dr. Watson once again."

© Springer-Verlag Berlin Heidelberg 2016

L. Nossov et al., *Formal SQL Tuning for Oracle Databases*,
DOI 10.1007/978-3-662-50417-8_1

1.1 Aims and Target Groups

The main objective of this book is to popularize formal SQL tuning. This method has a number of advantages:

- It is very simple. Even database specialists with minimal knowledge of performance tuning can quickly master and successfully apply this technique.
- For experienced specialists, this method provides a structured, targeted approach, which simplifies and accelerates SQL tuning considerably.
- Knowledge of data models is not a must for formal SQL tuning. The motto of the book "I know that I know nothing" (as far as the data model is concerned) reflects this advantage—even if somewhat exaggeratedly. This is especially advantageous for people who are often involved with unfamiliar databases or applications.
- By means of a few simple formal rules, one can quickly recognize problematical steps in the execution plan and initiate appropriate improvement measures.

We, the authors, are fascinated by this method. Consider how often you have been confronted by a SQL problem and didn't know where to begin. Formal SQL tuning offers an action plan, which normally provides a quick solution to the problem.

A lot of books on SQL tuning don't actually describe a tuning method, but more a method for efficient SQL programming. This assumes good SQL skills and data model knowledge. Such books are primarily aimed at developers because they possess this knowledge. It is different with database administrators, who usually know little or nothing about the relevant data model. Their SQL skills are often relatively modest compared to that of developers.

The formal method enables database administrators to carry out SQL tuning too, as the technique does not require knowledge of data models. Good SQL know-how is always an advantage when carrying out SQL tuning. If, however, your knowledge of SQL is not perfect, this is no major obstacle to the successful use of formal SQL tuning because, in practice, most SQL problems can be solved by relatively simple means. For this reason, this book is mainly directed at database administrators.

Each of the authors can name several examples of the successful use of formal SQL tuning. Quite often the authors proved to be even faster and more efficient tuners than the developers. Developers can therefore use formal SQL tuning to supplement their own methods and, in some cases, as a better alternative.

1.2 An Overview of the Book

This is not a SQL tuning textbook. The authors try to present the idea of SQL tuning in as concise and as comprehensible manner as possible. Before we begin with SQL tuning, we have to clarify what we understand by SQL tuning, what aims we are pursuing, etc. This is the topic of Chap. 2.

It is not necessary to have any in-depth knowledge of SQL tuning to make a successful start with the formal method. Basic knowledge is sufficient. It is, however, essential to have a rough understanding of execution plans, as one has to analyze these during formal SQL tuning. At the beginning, we thought that it was obvious that database specialists would have a rough notion of execution plans. Actually, this is true for most of them, but by no means all. When conducting a workshop on SQL tuning, one of us got into difficulties: When he was explaining very simple details, the participants were unable to follow him. He tried in vain to make them understand by reformulating and giving various examples but finally realized that the participants lacked even an elementary knowledge of execution plans. In order to avoid such cases, we decided to include Chap. 3. As the name of the chapter implies, it contains the essentials for understanding the formal method without difficulty.

Experienced readers can omit this chapter or perhaps leaf through it. Maybe they will find something interesting there.

The next chapter (Chap. 4) deals with the "philosophy" of this method. Without going into great detail, this chapter describes the objectives of the formal method, what the method is based on, what criteria it uses, and how it is carried out.

Formal SQL tuning consists of several steps. The most important of these, without doubt, is the analysis of the execution plan or, in other words, the recognition of bottlenecks in the execution plan. Such an analysis can be applied not only when tuning individual SQL statements but also for other problems, such as troubleshooting. For example, it can be used successfully for performance problems after an Oracle release change. Chapter 5 deals with this analysis. There we examine the bottlenecks involved in most practical cases encountered on a daily basis. We deliberately limit these cases to their essentials, so that the description is clear to the reader. Other cases can be analyzed according to the same principle.

In Chap. 6 the formal method is described in full.

Some practical experience is summarized in Chap. 7. In this chapter, some statistical information on the categories of problems which have occurred in practice is also presented.

In the appendix, an example is shown of how the same formal principle can be used in the analysis of performance problems after an Oracle migration.

1.3 Acknowledgments

The authors would like to thank their families for the patience and understanding they have shown.

We are very grateful to Anna Nosova for her creative and humorous illustrations.

We imagine muses to be females who gather around the authors and stimulate their creative talents. Our muse is of a completely different kind. It is male and heftily built but is nevertheless an inexhaustible source of inspiration for us. Many thanks to Peter Smith!

Leonid Nossov would especially like to thank Wolfgang Müller for his constant support and motivation.

2 Some Thoughts on the Term "SQL Tuning"

Before we really get started, it makes sense to clarify what we mean by SQL tuning. We will do that in this chapter.

2.1 SQL Tuning: Definitions and Objectives

Peter: "It is obvious what SQL tuning is."

Author: "How would you define it then?"

P.: "SQL tuning is a process which optimizes execution plans in the best way possible."

A.: "What do you understand by optimization?"

P.: "Reducing the runtime of each SQL statement, of course."

A.: "It is not as obvious as that. When several processes execute a SQL statement simultaneously and compete for certain database resources, one can decide in favor of a non-optimum execution plan in some situations. In this way, the runtime of an execution deteriorates but the waiting time decreases due to less competition, so that the total runtime of the competing processes improves."

P.: "If that's your only comment..."

A.: "What I like even less is your best possible optimization."

P.: "Why?"

A.: "For several reasons. Firstly, this criterion isn't necessary for SQL tuning. It is perfectly good enough to achieve an acceptable performance with SQL tuning. With your definition, you could fall into a trap: Often one doesn't know what the best possible optimization is. As a result, one doesn't know if the tuning process has been completed or not. What is more, something which is optimal on one system may not be optimal on another. The definition must therefore refer to a certain system."

P.: "You've completely ruined my definition."

© Springer-Verlag Berlin Heidelberg 2016

L. Nossov et al., *Formal SQL Tuning for Oracle Databases*,
DOI 10.1007/978-3-662-50417-8_2

A.: "Wait a minute Peter, I'm not finished yet. We still have to make clear that what we mean by SQL tuning in this book is the tuning of individual SQL statements."

P.: "Isn't that obvious?"

A.: "No. Imagine that your system is performing suboptimally after an Oracle migration. In this case it doesn't help to tune the individual SQL statements because there are usually too many of them. You have to find a reason for this problem (often it is a new optimizer feature). That is a different process, although you can use a similar principle to formal SQL tuning for this analysis (see the example in the appendix)."

P.: "Could you please define what you understand by SQL tuning? Criticism should be constructive."

A.: "Before I do that, I'd like to mention one more aspect of SQL tuning. The improvement measures for one executive plan must not be to the detriment of another. This can be especially critical with acute performance problems where there is no time to test the improvement measures. Now I can formulate my understanding of SQL tuning. By SQL tuning we mean a process by which an acceptable performance of a SQL statement is achieved on a system. Improvement measures for one SQL statement must not adversely affect another. Let's begin directly with a suboptimally performing SQL statement. How you identify the problematical SQL statement as described in detail in [1]."

2.2 SQL Tuners

There are three categories of SQL tuners: Oracle, developers, and database administrators. In this section we will briefly discuss how each of these tuners operates.

2.2.1 Oracle

Peter: "Does Oracle do SQL tuning? That's new to me."

Author: "Oracle's optimizer creates high-performance execution plans and, in this way, makes life much easier for the other two SQL tuner groups."

P.: "In my view that isn't SQL tuning. We have agreed that SQL tuning is responsible for improving suboptimal execution plans."

A.: "The optimizer is being constantly upgraded with automatic tuning elements."

P.: "What exactly do you mean by that?"

A.: "For example, the statistics feedback feature (cardinality feedback in Oracle 10 and 11). That is part of automatic reoptimization in 12c. When

Oracle notices that the real cardinality in the first execution differs greatly from the estimation, the optimizer calculates a new execution plan in which the cardinality gained in the first execution is taken into consideration. It then uses this plan in the next executions. Let's consider another feature: adaptive plans. At the time of execution, Oracle decides which join method it is best to use (nested loop or hash join)."

P.: "Those are only a few elements of SQL tuning."

A.: "Oracle also has SQL tuning advisor. This advisor can function in two modes: automatic and manual. During tuning, Oracle makes some improvement proposals (e.g., regarding access paths), calculates statistics which are specific to the SQL statements in question (that is to say for the relevant predicates), and saves these statistics in the form of optimizer hints in a SQL profile. These statistics represent adjustments or corrections to the optimizer estimations in the relevant execution plan. We've already discussed SQL profiles in some detail in [1]. I assume you've forgotten."

P.: "Actually, I have forgotten. I take back my objection: Oracle does do SQL tuning. Can you say something about the quality of this tuning?"

A.: "The quality of automatic tuning in Oracle 11 and 12 has improved noticeably. The basic principle of calculating specific statistics is effective and absolutely correct. It is no wonder the same principle is used in some other methods of SQL tuning (e.g., in [4])."

P.: "What use are such methods if Oracle has already implemented that?"

A.: "These methods are similar but not identical to Oracle's SQL tuning. One shouldn't forget that Oracle's automatic SQL tuning requires a license. The other methods can be used if one has no tuning pack license, for example."

P.: "If such methods are so good, why do you still need formal SQL tuning?"

A.: "I can give you a couple of reasons. Firstly, every method has its disadvantages and doesn't always help. Secondly, calculating specific optimizer statistics can take an unacceptably long time (e.g., in the case of a join of many large tables). In our opinion, such gathering of statistics can be avoided if runtime statistics are already available in the execution plan. This saves a lot of time. There is another reason. Formal SQL tuning is very easy. The analysis of runtime statistics in the execution plan is based on some simple rules and can be carried out by any developer or database administrator. For this reason, we go our own way in SQL tuning (Fig. 2.1)."

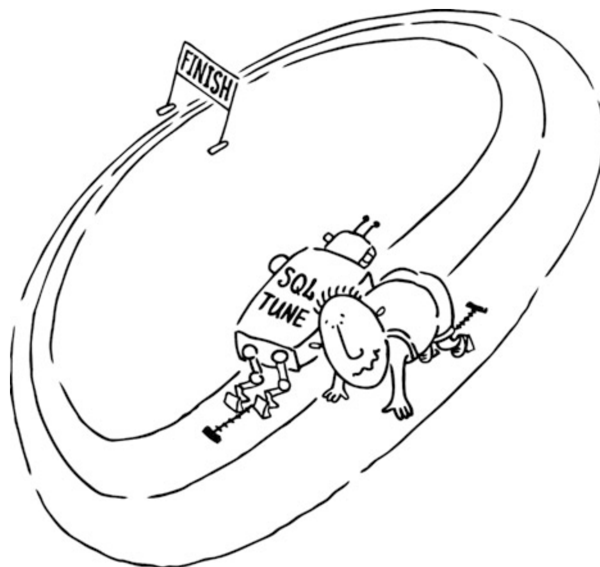


Fig. 2.1 Different routes—the same goal

2.2.2 Developer

Author: "A developer is in a very good position to achieve successful SQL tuning. He is familiar with the relevant data model and often knows immediately what an efficient execution plan should look like. If tuning requires a reformulation of the SQL statement, this is also no great problem for him because he normally has extensive SQL knowledge."

Peter: "I agree with you completely. That's why I often consult a developer when a SQL statement is running suboptimally."

A.: "Does that mean that you don't even try to solve the problem yourself?"

P.: "My knowledge of SQL tuning is mostly inadequate. Anyway, I think SQL tuning is really a job for developers."

A.: "So you are sure that a developer can apply SQL tuning much better than a database administrator?"

P.: "Yes, I think so."

A.: "Then I'll try to change your mind. In this section I'll show you some disadvantages of developers as SQL tuners. In the next section I'll explain why a database administrator can also be successful in SQL tuning."

P.: "What disadvantages do you mean? You yourself said that a developer has a good basis for carrying out SQL tuning."

A.: "If there is no developer on hand, that doesn't help you. What will you do in such a situation?"

P.: "Then I'm in trouble. But that's no argument against the developer."

A.: "It depends how you look at it. You would always have to have a developer available for SQL tuning. That's not always possible."

P.: “Obviously. But I hope you can list me a few more disadvantages.”

A.: “Imagine that software developed for relatively small companies is being used at a large company. In this case, the data volume and data distribution are different from the standard a developer is used to. Then at least some of the advantages the developer has to offer for SQL tuning are lost. I have often seen developers having problems in such situations. They may then point out alleged hardware bottlenecks or incorrect parameterization of the database. However, when a database administrator is brought in, and the situation analyzed in more detail, it often emerges that there are problems in the SQL area.”

P.: “That is something I have also experienced.”

A.: “It is not unusual for a developer to carry out unnecessary alterations to the SQL text during SQL tuning.”

P.: “Is that bad?”

A.: “When a performance problem is acute, it has to be remedied as quickly as possible. For organizational reasons, a program change usually requires several days. As the person responsible for the database, you have to survive these days somehow.”

P.: “How should SQL tuning be practiced then?”

A.: “I’ll explain that in the next section.”

2.2.3 Database Administrator

Author: “I often hear that no SQL tuning is possible without knowledge of data models. Unfortunately, this opinion is very widespread. What do you think, Peter?”

Peter: “Well, yes, you do have to know something about the data model.”

A.: “I, on the other hand, am sure that it is unnecessary. The formal method which we describe in this book does not require any knowledge of data models.”

P.: “I can’t imagine that.”

A.: “In practice I often see that database administrators are very keen to leave SQL tuning to the developers because they are convinced that they have no chance of succeeding with SQL tuning. The developers, on the other hand, don’t mind if the database administrators perform this task (Fig. 2.2).”

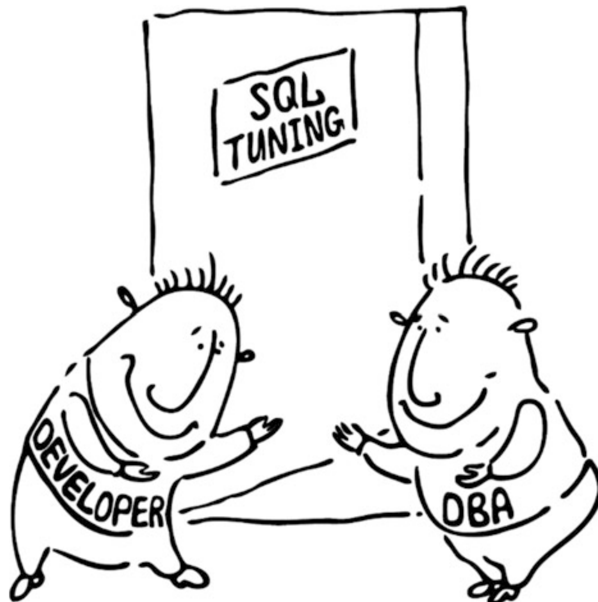
P.: “So who should carry out SQL tuning?”

A.: “The database administrator is responsible for the database. So it is primarily in his interest for the database to run efficiently.”

P.: “So you think the database administrator should perform SQL tuning?”

A.: “The database administrator should start SQL tuning. That is particularly important when there are no developers available or if the relevant performance problem is acute. Formal SQL tuning enables him to

Fig. 2.2 After you



perform this task. In most cases, he can successfully complete SQL tuning alone. It is only in very rare cases that this is impossible without a developer."

P.: "When is that?"

A.: "If one has to change the SQL statement for tuning purposes. It is also possible that the data model is so unsuitable for the SQL statement that it is hardly tunable at all. In this case, one either has to change the data model or completely rewrite the SQL statement. Then it is necessary to have a developer."

P.: "I have to get used to the idea, and that's not easy. Compared to a developer, the database administrator is at a distinct disadvantage as far as SQL tuning is concerned."

A.: "What other disadvantages can you name, apart from lack of data model knowledge, which is not necessary for SQL tuning anyway?"

P.: "An average database administrator is not very skilled in SQL. He's only able to program simple SQL statements."

A.: "That is often totally adequate, because, in most cases, one can perform tuning without any changes to the SQL statement."

P.: "How is that possible?"

A.: "We'll discuss that in the section "The Guideline: Tuning Without Changing the SQL Statement." I think we have talked enough about the term "SQL tuning." Now let's begin with formal SQL tuning. First, however, I have to say something about the execution plan."

Minimum Minimorum on the Subject of the "Execution Plan"

3

3.1 Can You Read Execution Plans?

Ask yourself this question. If you can't, or are unsure and hesitant with your answer, this chapter is for you.

It has a single objective: To impart the minimum knowledge of the execution plan necessary to enable you to read this book. In this respect, this chapter can be regarded as an introduction to the topic of the "execution plan." This minimum knowledge is also sufficient to gain a command of the method of formal SQL tuning.

3.2 Some Important Details

Imagine you have an execution plan in front of you. What do you see? What information could be useful for SQL tuning? How can this be applied practically? Let's ask our helper Peter Smith.

Author: "Peter, how familiar are you with this material?"
Peter: "I think 'to some extent' would be an appropriate answer."
A.: "Then let's try to improve your knowledge. How do you visualize execution plans?"
P.: "I normally use the DBMS_XPLAN package, but I have a question. You are talking about the execution plan. Isn't explain plan of any interest to us?"
A.: "Yes, it is, but it plays a subordinate role because these two plans often differ. So we have to start with an execution plan for SQL tuning. The function DBMS_XPLAN.DISPLAY_CURSOR displays execution plans for cursors from the SQL area. For example, the following SQL statements can be used for this purpose:

© Springer-Verlag Berlin Heidelberg 2016

L. Nossov et al., *Formal SQL Tuning for Oracle Databases*,
DOI 10.1007/978-3-662-50417-8_3

```
select plan_table_output from table
(sys.dbms_xplan.display_cursor(sql_id=><sql_id>,cursor_child_no=><child_number>,format=>'ADVANCED'))
;
```

sql_id and child_number of the respective cursor are to be found in the view V$SQL.

If you want to display an execution plan of the SQL statement just performed, you can initially execute this SQL statement followed by the command below:

```
select plan_table_output from table (sys.dbms_xplan.display_cursor('','','ADVANCED'));
```

Let's take the output of this function as a basis for our little study of the execution plan as it's relatively complete and contains practically all useful details required for SQL tuning. With the argument FORMAT, one can define the output of individual sections of the execution plan. The value 'ADVANCED' triggers the output of all sections. We already discussed this in [1]. I suggest that we start with these sections and discuss the most important ones."

3.2.1 Sections of the Execution Plan

3.2.1.1 Plan

Author: "This is the first section. What do you see there, Peter?"
Peter: "I thought the first section was the SQL text."
A.: "You're right of course, but an unformatted SQL text is not very important for us. So let's start with the plan."
P.: "Alright. In this section the execution plan itself is shown."
A.: "As you see, it is in the form of a table. The 'operation' column contains the relevant operations or steps of the execution plan, e.g., 'index range scan' or 'table access full.' These steps are in numerical order (see column 'Id')."
P.: "Some operations are indented to the right."
A.: "Yes, because the plan is hierarchical and is displayed accordingly. An operation A (e.g., 'index range scan') is subordinate to the other operation B (e.g., 'table access by index rowid') if it is located below B in the execution plan, if it is further indented to the right, and if no operation exists between A and B that is indented to the right as far as B."
P.: "To what extent is the hierarchy for SQL tuning important?"

No	Id	Operation	Name
03	0	SELECT STATEMENT	
02	1	TABLE ACCESS BY INDEX ROWID	T1
01	* 2	INDEX RANGE SCAN	I_T1

Fig. 3.1 Order of operations in the case of table access via an index range scan

A.: "It determines the order of operations in the execution plan:

- When two operations are equally indented to the right and subordinated to the same operation, the upper one is executed first. One can therefore say that the order of operations is from top to bottom.
- When one operation is subordinated to another, the subordinated one is executed first. As the subordinated operation is indented further to the right, the order of operations in this case goes from right to left.

Let's apply these rules to a simple example (see Fig. 3.1). Let's start with step 0. That is the select statement itself. 'Table access by index rowid' is subordinate to this operation. The operation 'index range scan,' in turn, is subordinate to 'table access by index rowid.' As there is no further subordinate operation, 'index range scan' is executed first."

P.: "So you start with the operation that is indented the furthest to the right."

A.: "Yes, following the 'right to left' rule. The order of operations in our example (and, incidentally, in the next one too) can be seen in the 'No.' column. In Fig. 3.2, the order is displayed in a join. Can you understand this sequence, Peter?"

P.: "If one combines two rules, 'from top to bottom' and from 'right to left,' one obtains the order. Do these rules always apply?"

A.: "You can find an exception in [2], for example. We have slightly modified this example and placed it in the 'Runtime Statistics' section. A second example with scalar subquery expressions is shown in Fig. 3.3."

P.: "I'm not really sure if I understand this example."

A.: "Right at the top of this execution plan, there are two subqueries from the table T2. According to the 'top to bottom' rule, they have to be executed first. This is impossible, however, because they correlate to the main query. So they only have to be executed after the 'hash join' (step 5 in the execution plan)."

P.: "To be honest with you, I'm a bit unsure about this rule now."

A.: "Normally it works (a few exceptions don't represent any major problems). Incidentally, I don't know of any exceptions to the rule 'from right to left.'"

P.: "When an execution plan is complex, it is not so easy to recognize how far an operation is indented to the right."

A.: "Yes, that's right. If there had been an additional column 'LEVEL' for a numerical degree of indentation in the output of DBMS_XPLAN. DISPLAY_CURSOR, it would have been much easier. But let's get back to the execution plan. The next column there is 'NAME.' This column contains object names for the respective operations. For 'table

No	Id	Operation	Name
	0	SELECT STATEMENT	
	…	…	
14	10	NESTED LOOPS OUTER	
11	11	NESTED LOOPS OUTER	
08	12	NESTED LOOPS	
05	13	NESTED LOOPS	
02	* 14	TABLE ACCESS BY INDEX ROWID	PICKLISTEN
01	* 15	INDEX RANGE SCAN	PIL_PK
04	16	TABLE ACCESS BY INDEX ROWID	PICKRUND
03	* 17	INDEX RANGE SCAN	PR_PL_FK_I
07	18	TABLE ACCESS BY INDEX ROWID	PICKAUF
06	* 19	INDEX RANGE SCAN	PI_PR_FK_I
10	* 20	TABLE ACCESS BY INDEX ROWID	QUANTEN
09	* 21	INDEX RANGE SCAN	QT_LE1_FK_I
13	* 22	TABLE ACCESS BY INDEX ROWID	PRUEFGRUENDE
12	* 23	INDEX RANGE SCAN	PG_NR_LE_I
	…	…	

Fig. 3.2 Order of operations in the join

```
SQL_ID  bvasuua44pqnu, child number 0
-------------------------------------
select (select max(b) from t2 where t2.a=t1.a) b, (select max(a) from
t2 where t2.b=t1.b) a from t1, t2 where t1.a=t2.a

Plan hash value: 970360602
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				5 (100)	
1	SORT AGGREGATE		1	26		
* 2	TABLE ACCESS FULL	T2	1	26	2 (0)	00:00:01
3	SORT AGGREGATE		1	26		
* 4	TABLE ACCESS FULL	T2	1	26	2 (0)	00:00:01
* 5	HASH JOIN		3	117	5 (20)	00:00:01
6	TABLE ACCESS FULL	T1	3	78	2 (0)	00:00:01
7	TABLE ACCESS FULL	T2	3	39	2 (0)	00:00:01

Fig. 3.3 Order of operations in the execution plan. An exception to the rule

access by index rowid,' for example, one finds the appropriate table name and an index name for 'index range scan.' In addition, there are columns which are relevant for operations on partitioned objects, for parallel operations, and for operations on remote databases. We refer to some of these columns in the examples. The remaining columns contain optimizer costs, optimizer estimates, and runtime statistics which are described later. We've almost forgotten to mention the plan hash value. Can you still remember how this value can be used in SQL tuning, Peter?"

P.: "This value can be used to compare execution plans. When execution plans have different hash values, they are in fact different. When the hash values are the same, it is highly likely that the plans are identical."

3.2.1.2 Query Block Name/Object Alias

Author: "Query block names and object aliases are listed in the next section of the execution plan. One can name query blocks with the hint QB_NAME (<Block-Name>). If one doesn't do this, Oracle generates these names automatically. The information from this section can be used for

www.allitebooks.com

optimizer hints. There is a detailed explanation of how to handle optimizer hints in [1]. Here is a brief example to illustrate this. For this purpose, let's take the appropriate section of the execution plan from Fig. 3.3."

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SEL$2
   2 - SEL$2 / T2@SEL$2
   3 - SEL$3
   4 - SEL$3 / T2@SEL$3
   5 - SEL$1
   6 - SEL$1 / T1@SEL$1
   7 - SEL$1 / T2@SEL$1
```

Peter: "Can we go through this list together please?"

A.: "The first query block at the top is SEL$2. It represents execution plan step 1. SEL$2 is therefore the first subquery and T2@SEL$2 is the alias of the table T2 in this subquery. In the same way, the query block SEL$3 is the second subquery and T2@SEL$3 is the alias of the table T2 in this subquery. The query block SEL$1 is the hash join (i.e., the main query). Accordingly, T1@SEL$1 and T2@SEL$1 are the aliases of the tables T1 and T2. Peter, see if you can use this information to change the order of operations in this join."

P.: "To do that I would use the hint LEADING. If I am not mistaken, the hint should look like this: LEADING(@SEL$1 T2@SEL$1 T1@SEL$1)."

A.: "A quick check shows that you are right (Fig. 3.4)."

P.: "What do we actually need the query block name in this hint for?"

A.: "The alternative would be to put the hint in the appropriate query block without a query block name. In this case, it is easy because the SQL statement is short and transparent. In a complex SQL statement, it is much more difficult to find the right query block. Use of the query block name renders this unnecessary in such cases."

P.: "Does one only need the information about the aliases from this section of the execution plan for use in optimizer hints?"

A.: "With this information it is easier to relate predicates to the appropriate tables. We will discuss predicates later in the section 'Predicate Information.' This information will help us to analyze an execution plan in the section 'A Non-selective Index.'"

```
SQL_ID  c1bwsfup22scs, child number 0
-------------------------------------
select /*+ LEADING(@SEL$1 T2@SEL$1 T1@SEL$1) */ (select max(b) from t2
where t2.a=t1.a) b, (select max(a) from t2 where t2.b=t1.b) a from t1,
t2 where t1.a=t2.a

Plan hash value: 4105034908
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				5 (100)	
1	SORT AGGREGATE		1	26		
* 2	TABLE ACCESS FULL	T2	1	26	2 (0)	00:00:01
3	SORT AGGREGATE		1	26		
* 4	TABLE ACCESS FULL	T2	1	26	2 (0)	00:00:01
* 5	HASH JOIN		3	117	5 (20)	00:00:01
6	TABLE ACCESS FULL	T2	3	39	2 (0)	00:00:01
7	TABLE ACCESS FULL	T1	3	78	2 (0)	00:00:01

Fig. 3.4 Query block names and table aliases can be used for hints

3.2.1.3 Outline Data

Author: "The section 'Outline Data' contains special optimizer hints (outlines), which Oracle generates for fixing the relevant execution plan. Outlines can look like this:"

```
Outline data
-------------

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.2.0.3')
      DB_VERSION('11.2.0.3')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$2")
      OUTLINE_LEAF(@"SEL$3")
      OUTLINE_LEAF(@"SEL$1")
      FULL(@"SEL$1" "T1"@"SEL$1")
      INDEX_RS_ASC(@"SEL$3" "T3"@"SEL$3" ("T3"."A"))
      INDEX_RS_ASC(@"SEL$2" "T2"@"SEL$2" ("T2"."A"))
      END_OUTLINE_DATA
  */
```

Peter: "You can enter these outlines into a SQL statement as optimizer hints to fix the relevant execution plan. Is that right?"

A.: "In principle, that's right. One has to be careful with parallel operations, however, because Oracle doesn't normally generate any outlines for them. In this case, you have to complete the outlines with the appropriate parallel hints."

P.: "In [1] we learned that one can either create outlines as stored outlines or store them in a SQL profile. Can you use outlines in any other way for SQL tuning?"

A.: "First of all, I'd like to point out that the stored outlines in 12c are no longer supported. You can use the outlines from the section 'Outline Data' as patterns for your own hints. That is especially helpful when you only need to modify the outlines slightly. There are outlines in execution plans of cursors in the SQL area and in the AWR. They are missing in the statspack, however."

3.2.1.4 Peeked Binds

Author: "In this section bind values which have been used for bind peeking are listed. We discussed the concept of bind peeking in detail in [1]."

Peter: "Yes, those are values that Oracle considered when creating the relevant execution plan. Are they also of interest for SQL tuning?"

A.: "Of course. At least as a source of bind values. We'll come to that in the section 'The Method: Analysis of Runtime Statistics in the Execution Plan.' An example of 'Peeked Binds' follows below (1 is the number or the name of the bind variable):

```
Peeked Binds (identified by position):
--------------------------------------

   1 - (NUMBER): 1300
```

Bind values can be found in execution plans from the SQL area and from the AWR. They are missing in the statspack."

3.2.1.5 Predicate Information

Author: "This section of the execution plan is enormously important for SQL tuning. Let's take this section as an example. What do you see here Peter?"

```
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(("MAIN"."COMMITTED_AT">SYSTIMESTAMP(6)-10 AND "MAIN"."NOT_EXPORTED_AT" IS NULL))
   2 - access("MAIN"."POINT_OF_SALE"=:B1)
```

Peter: "Two lines: filter and access. I assume that a table access with the predicate POINT_OF_SALE=:B1 is being carried out. An index may be used for access in the second step of the execution plan. All rows which have been selected will be filtered out in the first step of the execution plan. Predicates which belong to the filter are used for this purpose. For example, in the case of a table access by rowid."

A.: "Quite right, Peter. Please have a look below:"

Id	Operation	Name
0	SELECT STATEMENT	
* 1	TABLE ACCESS BY INDEX ROWID	ARTICLE_WRITEOFF
* 2	INDEX RANGE SCAN	I_ARTICLE_WRITEOFF_1

P.: "But how does this help me with SQL tuning? I can take the relevant predicates from the SQL statement."

A.: "Not always. For example, Oracle can generate predicates from constraints. In this case, they are missing in the SQL statement. If a SQL statement is large and not transparent, and also contains a few views, this task is then relatively complex. In this case, you can only assume which predicates Oracle is using. The execution plan section 'Predicate Information' describes this in precise detail."

P.: "I see. How can I use this information for SQL tuning?"

A.: "You can assess how selective the various predicates are. It is important that one uses selective predicates for accesses (in our example they are listed with 'access' in the execution plan). This accelerates these accesses and reduces the number of hits. Predicates with filters can also reduce the number of hits, which accelerates the subsequent execution plan steps. The performance of the execution plan step in which the relevant filtering takes place is hardly influenced by this, however. We will discuss this in detail when we describe

formal SQL tuning. With exadata, predicates can also be allocated to storage as well as filter and access. In this case, predicates are transferred to the storage layer and the relevant data access optimized by means of exadata features (e.g., with storage indexes)."

P.: "If I'm not mistaken, information about predicates is not maintained either in the AWR or in the statspack."

A.: "That's right. You have to obtain this information from an execution plan of a cursor from the SQL area. As a workaround against some Oracle bugs, the parameter setting "_cursor_plan_unparse_enabled"=false is used often. In this case, Oracle does not generate any information about predicates. They are then also missing in the SQL area. The same applies for the information from the next section of the execution plan 'Column Projection Information.' And just one final important comment. You will no doubt have noticed already that the execution plan steps with predicates are marked with an asterisk in the 'Id' column."

3.2.1.6 Column Projection Information

Author: "Here the tables or index columns selected in the relevant execution plan step are listed. Let's take the section 'Column Projection Information' from the execution plan above:"

```
Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - "MAIN"."ID"[NUMBER,22], "MAIN"."POINT_OF_SALE"[NUMBER,22],
"MAIN"."EXECUTED_BY"[NUMBER,22],
       "MAIN"."COMMITTED_AT"[TIMESTAMP,11], "MAIN"."COMMITTED_AT_LOCALDATE"[DATE,7],
"MAIN"."EXPORTED_AT"[TIMESTAMP,11],
       "MAIN"."NOT_EXPORTED_AT"[TIMESTAMP,11], "MAIN"."RESOLVED"[CHARACTER,1]
   2 - "MAIN".ROWID[ROWID,10], "MAIN"."POINT_OF_SALE"[NUMBER,22],
"MAIN"."COMMITTED_AT_LOCALDATE"[DATE,7]
```

Peter: "I find this information fairly useless."

A.: "It may not be as important as the predicates, but it definitely isn't useless. I'll try to demonstrate that using our example. In the second step of the execution plan, there occurs an index range scan of the index I_ARTICLE_WRITEOFF_1. Here Oracle uses the predicate POINT_OF_SALE=:B1. From the section 'Column Projection Information,' one can gather that this index has at least one more column COMMITED_AT_LOCALDATE, which is simply selected and not queried because it does not occur in any predicate. In the first step of the execution plan, another six table columns are selected: ID, EXECUTED_BY, COMMITED_AT, EXPORTED_AT, NOT_EXPORTED_AT, RESOLVED. If you want to avoid table access by rowid, you have to extend the index I_ARTICLE_WRITEOFF_1 by these six columns."

P.: "That makes at least eight columns in this index. Isn't that too many?"

A.: "Yes, I wouldn't do that without a very good reason. But this is just an example. If it had been possible to extend the index by a couple of columns

to avoid the table access, that would have been no problem. The information about the columns to be selected helps in making the right decision. I hope this example has changed your opinion."

3.2.1.7 Remote SQL Information

Author: "In this section of the execution plan, SQL statements which are executed on a remote database are listed. For example:"

```
Remote SQL Information (identified by operation id):
----------------------------------------------------

  17 - SELECT "OBJECT_NAME","LAST_UPDATED","START_DATE","OBJECT_TYPE" FROM
"SA_NBA"."W_PRO_NBA_OBJECT_STATUS" "W_PRO_NBA_OBJECT_STATUS" WHERE "OBJECT_NAME" LIKE
        'NBA_ACTIVITY_LIST%EXP%' AND "LAST_UPDATED" IS NOT NULL (accessing 'INKA_DW_IF_VDCW_DW' )
```

Peter: "In brackets there is a database link. Is that right?"

A.: "Yes, that's right. If remote access proves problematical, the relevant SQL statement can be found and tuned on the remote database."

P.: "How does one recognize that remote access is problematical?"

A.: "You can recognize that from the runtime statistics, which we will discuss in this chapter."

P.: "It wouldn't be bad if Oracle were to display the relevant SQL Id."

A.: "You're right. But one can calculate a signature for the displayed SQL text and search for the appropriate SQL statement on the remote database via this signature. What these signatures are, and how they are obtained, is described in [1]."

3.2.1.8 Note

Author: "The section 'Note' is very easy to explain. It contains information which can also be very useful for SQL tuning. For example, one can establish whether dynamic sampling or user bind peeking has been used."

Peter: "What information is particularly important?"

A.: "In principle, it is all important. It depends on the specific case. For example, I always check whether an inefficient execution plan is fixed with a SQL profile or with the SQL plan baselines. During SQL tuning, it makes sense to disable the relevant profile or the baselines, because they tend to be more detrimental than helpful. Sometimes the optimizer finds a better execution plan itself if no SQL profiles and SQL plan baselines are used."

3.2.2 Optimizer Estimations and Costs

Author: "The CBO (cost-based optimizer) uses a numerical value (so-called optimizer costs) to estimate the effectiveness of execution plans. This value is calculated according to certain rules and is based on optimizer statistics of objects involved in the relevant cursor. This value is cumulative, i.e., it accumulates values of the execution steps which are subordinate to the relevant step. In other words, the optimizer costs of an execution plan step accumulate optimizer costs of all steps which are performed before it. The smaller the optimizer costs of the whole execution plan are, the more effective the execution plan is (at least for the optimizer)."

Peter: "How can one use optimizer costs in SQL tuning?"

A.: "The optimizer costs help one to understand the decisions of the optimizer. For this, however, one must know how the optimizer calculates these costs. For SQL tuning, this value is normally of no interest, just like the optimizer estimation 'time.'"

P.: "Are there any optimizer estimations at all that can help with SQL tuning?"

A.: "Definitely. For example, 'rows' which are especially important for formal SQL tuning. These are the number of rows which are found in an execution plan step."

P: "So, not read but estimated on the basis of the statistics?"

A.: "Yes. That's the number of rows after the use of predicates related to the relevant execution plan step. This value is not cumulative. It is therefore calculated separately for each step, like the optimizer estimation 'bytes' (see Fig. 3.4)"

P.: "How can the 'row' estimation help us?"

A.: "We can compare this estimation with the relevant runtime statistics. We will discuss runtime statistics in the next section."

P.: "Is this estimation calculated for just one or for all executions of the relevant execution plan step?"

A.: "That's a very good question. In order to carry out a comparison with runtime statistics, we need to know the answer. This estimation refers to one execution of the execution plan step."

3.2.3 Runtime Statistics

Author: "Runtime statistics in the execution plan are extremely important for formal SQL tuning."

Peter: "You've already said that we can compare the runtime statistics with the optimizer estimations."

A.: "That's only one possibility in SQL tuning. What is much more important is that the runtime statistics help to recognize a bottleneck in the execution plan. We will discuss this in the chapter 'Bottlenecks in the Execution Plan.' In Oracle there are a number of ways of obtaining runtime statistics (we will describe these possibilities in the section 'The Method: Analysis of Runtime Statistics in the Execution Plan'). One of these possibilities is that the runtime statistics are generated directly in the execution plan. This does not occur per default because it is a costly option. In the abovementioned section, we discuss how this is done. In order to display all runtime statistics with the function DBMS_XPLAN.DISPLAY_CURSOR, the argument FORMAT has to be supplemented with the word 'ALLSTATS' (it is also possible to have these statistics displayed singly). In order to display runtime statistics for the last execution of a cursor, one has to supplement the argument FORMAT with the word 'LAST' (otherwise the statistics will be listed summarily for all executions). With the following command, you can identify an execution plan with all runtime statistics for the last execution of a cursor from the SQL area."

```
select plan_table_output from table
(sys.dbms_xplan.display_cursor(sql_id=><sql_id>,cursor_child_no=><child_number>,format=>'ADVANCED
ALLSTATS LAST'));
```

P.: "Do you really need all runtime statistics for SQL tuning?"

A.: "It is difficult to say in advance what you need in a concrete case. For this reason, I always display them all. When the optimizer estimations and the relevant runtime statistics have the same name (like 'rows' and 'time'), these names are supplemented with 'E-' (estimated) and with 'A-' (actual). Unlike the optimizer estimation 'rows,' the relevant runtime statistic is calculated for all executions of the execution plan step it belongs to."

P.: "But we want to compare these two values with each other. How is that possible?"

A.: "The runtime statistic 'starts' makes this possible. That is the number of executions for each execution plan step. If we divide the runtime statistic 'A-rows' for a step, by the number of executions, we obtain the number of rows for one execution of this step. We can compare this value with the relevant estimation 'rows.' This applies to non-partitioned objects. With partitioned objects it is somewhat more difficult because the statistic 'starts' includes 'scanned' partitions or sub-partitions. I think we can now present the example from [2] which was promised in the section 'Plan' (Fig. 3.5)."

P.: "What does this example tell us?"

A.: "Step 6 of the above execution plan has to be executed after step 2 because these steps are equally indented to the right and step 6 is below step 2. In fact, this step is executed first."

P.: "How can you see that?"

```
SQL_ID  f4yzggva3jkuy, child number 0
-------------------------------------
select * from dual d1 where exists (select /*+ no_unnest */ * from dual
where dummy='Y') and exists (select /*+ unnest */ * from dual d2 where
d1.dummy = d2.dummy)

Plan hash value: 4062679786
```

Id	Operation	Name	Starts	A-Rows
0	SELECT STATEMENT		1	0
* 1	FILTER		1	0
2	NESTED LOOPS SEMI		0	0
3	TABLE ACCESS FULL	DUAL	0	0
* 4	VIEW	VW_SQ_1	0	0
5	TABLE ACCESS FULL	DUAL	0	0
* 6	TABLE ACCESS FULL	DUAL	1	0

```
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter( IS NOT NULL)
   4 - filter("D1"."DUMMY"="ITEM_1")
   6 - filter("DUMMY"='Y')
```

Fig. 3.5 Order of operations in the execution plan. An example from J. Lewis

A.: "From the runtime statistic 'starts': Step 6 has been executed once, whereas step 2 has never been executed."

P.: "Very interesting. Without runtime statistics it would have been impossible to recognize that."

A.: "One could have assumed that because the first subquery does not correlate with the main query and can be executed first for this reason. That can easily be recognized with the runtime statistics. Let's continue with the runtime statistics. The next 3 are 'buffers,' 'reads,' and 'writes.' That is the number of data blocks which is respectively read from the buffer cache or from the hard disk or is written on the hard disk. These statistics are cumulative. In addition to the runtime statistics mentioned, some work area statistics are listed, e.g., 'used-tmp'—the size of the used temporary tablespace. These statistics refer to the relevant execution plan steps and are not cumulative. I have forgotten to mention that the runtime statistic 'time' displays runtime in hours, minutes, and seconds and is cumulative."

3.3 Summary

- Execution plans are hierarchical and are displayed accordingly. This hierarchy determines the order of operations:
  - When two operations are equally indented to the right, the upper of these two is executed first. That is the "top to bottom" rule.
  - When one operation is subordinate to another, the subordinate operation is executed first. That is the "right to left" rule, because the subordinate operation is indented further to the right. There are a few exceptions to the "top to bottom" rule, but in most cases it is correct.

- Information from the section “Query Block Name/Object Aliases” of the execution plan can be used for optimizer hints.
- Outlines are special optimizer hints, which are designed to fix the relevant execution plan. Oracle does not normally generate any hints in the outlines for parallel operations. There are no outlines in the statspack.
- The section “Peeked Bind” can be used as a source of bind values for execution of the relevant SQL statement. These bind values are missing in the statspack.
- There are two types of “predicate information”: access and filter. In access, predicates are listed which Oracle uses when accessing data in the relevant execution plan step. In contrast to access, predicates from filter are only used for filtering the rows found. The section “Predicate Information” is very important for formal SQL tuning.
- In the section “Column Projection Information,” the tables or index columns which are to be selected in the relevant execution plan step are listed.
- The section “Note” contains some additional information, for example, concerning the use of dynamic sampling or cardinality feedback.

4 Approaches to Formal SQL Tuning

This is the first of the three chapters in which the formal method is described. In this chapter, we formulate the objective, principle, method, main criterion, approach, and guideline for formal SQL tuning.

4.1 The Objective: Effective SQL Tuning

The objective of formal SQL tuning is an acceptable performance of a SQL statement. We have already formulated this objective in the section "SQL Tuning: Definition and Objectives."

The formal character of this method, i.e., tuning according to formal rules, enables tuning to be carried out without knowledge of data models. Perhaps this is not so attractive for developers (although developers can also have certain problems with large, complex SQL statements). For database administrators, this method is a key to independent SQL tuning. They are not reliant on developers when performing SQL tuning.

Formal SQL tuning deals with those cases which occur most often in practice. Consequently, we can give a clear, concise formulation of the formal method. The formal rules are so simple that even a beginner can master the method and use it successfully. This method is also "extendable." It can be developed further if necessary.

The formal method is very effective in case of acute performance problems. It very quickly helps to find an acceptable execution plan (even if the means used are sometimes fairly "rough and ready"). A more refined analysis can be carried out at leisure later.

© Springer-Verlag Berlin Heidelberg 2016

L. Nossov et al., *Formal SQL Tuning for Oracle Databases*,
DOI 10.1007/978-3-662-50417-8_4

4.2 The Principle: Elimination of "Brakes" in the Execution Plan

Oracle's optimizer always tries to generate the best execution plan. This is actually unnecessary for SQL tuning. It is perfectly adequate to upgrade a poor execution plan to an acceptable one. It is irrelevant whether the improved plan is the optimum plan. This makes the task of SQL tuning easier.

This is precisely the principle by which formal SQL tuning functions: One finds problematical steps in the execution plan and eliminates them. In this way, one very often achieves astonishingly good solutions.

4.3 The Method: Analysis of Runtime Statistics in the Execution Plan

In order to find problematical execution plan steps, one can either use optimizer estimations or runtime statistics. In the case of optimizer estimations, one has to be careful. They can be relatively imprecise for a number of reasons. If optimizer statistics of the objects involved in the SQL statement are incorrect, then the optimizer estimations are also far from realistic. It's possible for optimizer statistics to be exact but estimations not. This is because the optimizer makes assumptions in some places. One must be particularly careful with optimizer estimations in poor execution plans because it is often precisely these which cause suboptimal plans.

Optimizer estimations are, therefore, not a reliable basis for the analysis of execution plans. The runtime statistics in execution plans are much more suitable. They provide information on the actual course of individual execution plan steps.

How one can request Oracle to display runtime statistics in the execution plan has already been described in the section "Runtime Statistics." As the generation of runtime statistics is expensive, this is omitted by default. It can, however, be activated for test runs, either for one session (with the parameter setting statistics_level=all) or with the hint GATHER_PLAN_STATISTICS (also as a hidden hint, see [1]), for a SQL statement.

The other possibility of accessing the statistics in the execution plan is the feature "SQL monitoring," which is available from Oracle 11 onward. SQL monitoring is described in detail in [1]. Unfortunately, no information is listed on predicates and on column projections in the SQL monitoring reports. This has to be obtained separately in the execution plans from the SQL area. It should not be forgotten that SQL monitoring requires a license.

SQL tracing is another source of runtime statistics in the execution plan. As it is relatively complicated to generate and analyze a SQL tracing (at least more complicated than the other two features described above), this option is not used as often as the others.

4.4 The Main Criterion: The Runtime Statistic "Cardinality"

Cardinality is the main criterion in the analysis of execution plans with the formal method (Fig. 4.1). In the output of DBMS_XPLAN.DISPLAY_CURSOR, this statistic is referred to as "A-rows", in the SQL monitoring report as "rows (actual)," and in SQL tracing as "rows."

This statistic contains the number of rows which have been selected in the relevant execution plan step. A high cardinality indicates the processing of a large number of rows in an execution plan step. Accordingly, this step can be expensive. That is the first reason why cardinality is selected as the main criterion. The rows selected in an execution plan step are processed further in the subsequent steps. If there are a large number of these, this makes the next execution plan steps more expensive. That is the second reason.

It is possible for a large number of rows to be processed in one step although its cardinality is low, e.g., if a full table scan (FTS) is used to find data about very selective predicates from a large table. In such a case, one also has to consider other runtime statistics for the analysis (e.g., buffer gets or buffers in the output of DBMS_XPLAN.DISPLAY_CURSOR). But in this case too, cardinality is very important for the analysis: In the case of an FTS on a large table with a low cardinality, one can immediately assume that the relevant predicates are selective, and one can create an index for the relevant columns as an improvement measure.

Although the other runtime statistics play a subordinate role in the formal method, they can also be helpful for the analysis of the execution plan.

Fig. 4.1 If Archimedes had used SQL tuning

4.5 The Procedure: An Iterative Process

Unfortunately, it is seldom the case that the first improvement already results in an acceptable execution plan. As a result, one has to be prepared for several executions of the problematical SQL statement or the problematical process.

One can, for example, execute the problematical SQL statement in SQL*Plus. If this statement contains bind variables, these can be defined as variables in SQL*Plus and set to corresponding values. This is possible for some common data types (e.g., VARCHAR2 or NUMBER).

When tuning a data manipulation language (DML) statement, it is sometimes possible to extract a select statement, to tune it, and to use the results of the tuning on the DML statement (see [1]). As a result, it is no longer necessary to execute this DML statement for tuning purposes and to change the relevant data.

When one tunes a process which is executed repeatedly, it is often very practical to use SQL monitoring for tuning (e.g., see [1]). In this case, no separate executions of individual SQL statements of this process are necessary. This saves time and effort.

Here we have outlined a number of possibilities of how problematical SQL statements can be executed for the purpose of SQL tuning. When carrying out SQL tuning, it is important to consider these possibilities in advance and to select a suitable one.

4.6 The Guideline: Tuning Without Changing the SQL Statement

With SQL tuning, one must also consider how to implement the relevant improvements. Very often, changes to the SQL text are not possible because the application developers are not readily available (e.g., in the case of a standard application). If a performance problem is acute, one must act very quickly. In this case, it does not help if the developers are in a neighboring office, because any code change cannot be made immediately—mostly for organizational reasons.

It therefore makes sense if you are prepared for SQL tuning without changes to the SQL statement from the outset. The following improvements are welcome in SQL tuning:

1. Creation of new indexes. One has to be careful and only create an index when there is no other solution possible because each new index influences other execution plans and can, in some cases, affect them adversely.
2. Extension of existing indexes. Here there is also a risk that the extended index could adversely affect other execution plans. However, this risk is generally much lower than is the case with a completely new index.
3. Use of optimizer hints. One can use optimizer hints as hidden hints (see [1]). Alternatively, the plan optimized with the hints can be fixed (some methods to

achieve this can be found in [1]). A change to the SQL statement is unnecessary in either of these cases.

4. Change of database parameters. If one optimizes a SQL statement with database parameters, it makes sense to fix the optimized plan instead of implementing the relevant parameter settings throughout the system. Alternatively, one can try setting the relevant parameters with the hint OPT_PARAM dedicated to a problematical SQL statement (see [1]). According to the documentation, this is only possible for a handful of parameters. In reality, this hint works for many parameters.
5. Gathering of optimizer statistics. New optimizer statistics can also influence other plans. The risk of other plans deteriorating is relatively low, however. In order to gather optimizer statistics without any risk, one can generate new statistics but not publish them (set preference PUBLISH to FALSE). In a separate session, one can then allow these pending statistics with the parameter setting optimizer_use_pending_statistics=true and fix a plan optimized with these statistics.

Some of these improvements are pure workarounds. However, they are completely legitimate in the case of an acute performance problem. An ultimate solution will be worked out later when the acute problem has been eliminated.

One might think that these improvements are only suitable for a relatively limited group of cases. The reality looks different, however: In most practical cases, SQL tuning can be carried out without changing the SQL statement. Only rarely must the SQL statement in question be changed in order to achieve an acceptable plan.

Bottlenecks in the Execution Plan

5

In this chapter we will describe the core elements of the formal method: The recognition of bottlenecks in the execution plan. Formal SQL tuning provides a very simple procedure for recognizing bottlenecks (even for large, complex execution plans). One could say that this method is your Ariadne thread in the labyrinth of execution plans (Fig. 5.1).

5.1 "Local" Problems in the Execution Plan

Local problems are those which are "localized" to one or more (as a rule two) execution plan steps.

5.1.1 A Missing Index

Author: "Peter, how would you recognize that an index is missing in an execution plan in case of an FTS?"
Peter: "A low cardinality of an FTS can be an indication of a possible selective index."
A.: "What do you mean by a low cardinality? 1, 2, or 100 maybe?"
P.: "I can't give you a concrete figure."
A.: "So when can you say that the cardinality of an FTS is low?"
P.: "Wait a minute. When the table in question is relatively large (and those are the ones that interest us), Oracle has to do quite a lot of buffer gets or physical reads with an FTS. I would compare the cardinality with the number of buffer gets."
A.: "Would you check every FTS in the execution plan in this way?"
P.: "Only the steps with a long runtime."

© Springer-Verlag Berlin Heidelberg 2016

L. Nossov et al., *Formal SQL Tuning for Oracle Databases*,
DOI 10.1007/978-3-662-50417-8_5

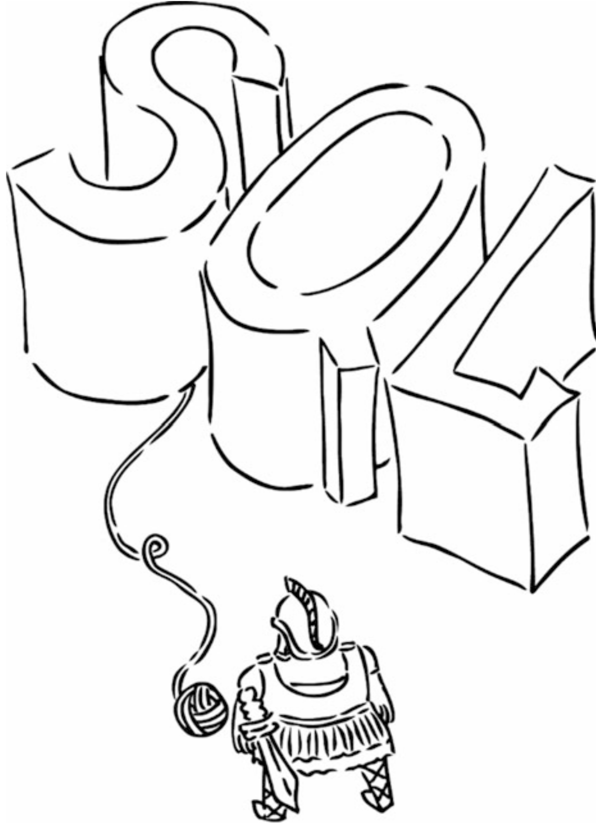


Fig. 5.1 Formal SQL tuning is your Ariadne thread in the labyrinth of execution plans

```
SQL_ID  8dwcq7z24k9xt, child number 0
-------------------------------------
select count(*) from t1 where b=4000

Plan hash value: 3724264953
```

Id	Operation	Name	Starts	A-Rows	Buffers
0	SELECT STATEMENT		1	1	185
1	SORT AGGREGATE		1	1	185
* 2	TABLE ACCESS FULL	T1	1	1	185

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("B"=4000)
```

Fig. 5.2 An indication of a missing index: FTS with a low cardinality and a large number of buffer gets

A.: “Let’s use a test case to show how one identifies a missing index. For this purpose, we will create a table T1 with 100,000 rows. After that we will execute a SQL statement and display the relevant execution plan with runtime statistics (Fig. 5.2).”

P.: “In the second execution plan step, an FTS was carried out. 185 buffer gets were done and one row found. An index for column B is intended to

```
SQL_ID  8dwcq7z24k9xt, child number 0
-------------------------------------
select count(*) from t1 where b=4000

Plan hash value: 3547404373
```

Id	Operation	Name	Starts	A-Rows	Buffers
0	SELECT STATEMENT		1	1	2
1	SORT AGGREGATE		1	1	2
* 2	INDEX RANGE SCAN	I_T1	1	1	2

Fig. 5.3 An index access substantially reduces the number of buffer gets

reduce the number of buffer gets and thereby improve the execution plan."

A.: "You took column B from the filter in the section 'Predicate Information,' didn't you?"

P.: "That wasn't necessary in our simple example. If the SQL statement had been much more complex, I would have done that."

A.: "Let's create the index:

```
SQL> create index i_t1 on t1(b);

Index created.
```

Now we can execute the SQL statement (Fig. 5.3)."

P.: "The index access has reduced the number of buffer gets to two! But weren't we a bit premature in creating the index? Theoretically, it's possible that column B isn't selective. The value of 4000 could be an exception."

A.: "The word 'theoretically' is very appropriate here. In practice, things look rather different. When a SQL statement is problematical, it is normally slow in several executions. During these executions, the FTS is carried out several times with a low cardinality. In this case, the creation of the relevant index is definitely helpful. But if you really want to be on the safe side, you can check the selectivity of the relevant predicates directly. I have a question for you too. Is a missing index the only possible reason for an expensive FTS with a low cardinality?"

P.: "When you ask in that way, then presumably there is at least one further reason, but I've no idea what it could be."

A.: "Think about it, Peter. A large table is scanned and only a few rows found. What could be the reason?"

P.: "Possibly that this table doesn't contain many rows?"

A.: "Correct! What do we call a table with a lot of data blocks and with only a few rows?"

P.: "A sparse table?"

A.: "Exactly. Now let's drop the index and delete all rows apart from one in our table:

```
SQL> drop index i_t1;

Index dropped.

SQL>
SQL> delete from t1 where b!=4000;

99999 rows deleted.

SQL>
SQL> select count(*) from t1;

  COUNT(*)
----------
         1
```

Let's execute a SQL statement with another predicate (Fig. 5.4)."

P.: "The runtime statistics are exactly the same as in Fig. 5.2."

A.: "What would you do to improve this execution plan?"

P.: "That's obvious. You have to reorganize a sparse table."

A.: "In principle you're right, but imagine that the table is very large. Then reorganizing the table will take a long time, but you need a quick solution."

P.: "Could an index be useful here too?"

A.: "Very good Peter! Let's test that (Fig. 5.5)."

P.: "Have you had such cases in practice?"

A.: "Yes, once I tuned a database with a lot of direct reads. There were some sparse tables with a lot of data blocks, so that Oracle executed an FTS with direct reads (i.e., not via the buffer cache, more information on this feature can be found in [1]). I proposed the solution with indexes as a workaround. With this workaround, all sparse tables were gradually reorganized."

```
SQL_ID  5p7s1hq62rf59, child number 0
-------------------------------------
select count(*) from t1 where a=1

Plan hash value: 3724264953
```

Id	Operation	Name	Starts	A-Rows	Buffers
0	SELECT STATEMENT		1	1	185
1	SORT AGGREGATE		1	1	185
* 2	TABLE ACCESS FULL	T1	1	1	185

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("A"=1)
```

Fig. 5.4 An FTS on a sparse table

```
SQL> create index i_t2 on t1(a);

Index created.


SQL_ID  5p7s1hq62rf59, child number 0
-------------------------------------
select count(*) from t1 where a=1

Plan hash value: 4157480490
```

Id	Operation	Name	Starts	A-Rows	Buffers
0	SELECT STATEMENT		1	1	1
1	SORT AGGREGATE		1	1	1
* 2	INDEX RANGE SCAN	I_T2	1	1	1

Fig. 5.5 An index as a quick source of help with FTS on a large sparse table

5.1.2 A Nonselective Index

Author: "In this section we will discuss a very important and interesting topic: How does one recognize a nonselective index in the execution plan? How would you do that, Peter?"

Peter: "When a nonselective index is used, the relevant index scan has to return a large number of rows. The relevant cardinality must therefore be high."

A.: "What do you regard as a high cardinality? How can you recognize that an index isn't selective and that something can be improved?"

P.: "You ask difficult questions. I don't know."

A.: "Let's start with the high cardinality. It's true that it isn't possible to say which cardinality is high when we talk about it abstractly and not in relation to any execution plan. The cardinality therefore has to be high for a concrete execution plan, for example, the highest one there. How can one decide that the index in question isn't selective? The answer to this question can also be found in the execution plan. When the SQL statement for a table access contains selective predicates, and these predicates are only partly present in an index (which makes this index nonselective), the remaining predicates have to be queried elsewhere in the execution plan and the cardinality substantially reduced. This normally occurs in the filter of the subsequent table access by rowid."

P.: "Stop, stop, stop! I can't follow you."

A.: "Basically, it's very simple. I'll demonstrate it using the following example (Fig. 5.6)."

P.: "I can't say that I understand it any better now."

A.: "First let's look for the steps with a high cardinality. Which are those, Peter?"

P.: "Those are steps 16 and 14. There an index range scan of the index IDX_PROCESS_PERSON has been executed and 10M and 12M rows found, respectively."

A.: "Is this index selective?"

P.: "In step 14, an average of approximately 24,000 rows (12M/500) are selected per execution. That is no small value. We do not know, however, how large the table TA_PROCESS is. I don't know if this index is selective."

A.: "But look what happens in the next execution plan step."

P.: "The table TA_PROCESS is read by rowid. Oracle needed 7.53 seconds for the index range scan and substantially longer 35.94 seconds for the table access in step 13."

A.: "It's good that you noticed that. It often happens that a high cardinality only really has an effect in one of the following execution plan steps.

Id	Operation	Name	Starts	A-Rows	A-Time
0	SELECT STATEMENT		1	3	00:01:01.99
* 1	COUNT STOPKEY		1	3	00:01:01.99
2	VIEW		1	3	00:01:01.99
* 3	SORT ORDER BY STOPKEY		1	3	00:01:01.99
4	VIEW		1	133	00:01:01.99
5	UNION-ALL		1	133	00:01:01.99
6	INLIST ITERATOR		1	1	00:00:00.01
* 7	TABLE ACCESS BY INDEX ROWID	TA_PROCESS	2	1	00:00:00.01
* 8	INDEX RANGE SCAN	IDX_PROCESS_PROCESSSTATE	2	266	00:00:00.01
9	NESTED LOOPS ANTI		1	132	00:01:01.98
10	NESTED LOOPS ANTI		1	602	00:00:35.96
* 11	TABLE ACCESS BY INDEX ROWID	TA_PROCESS	1	643	00:00:00.01
* 12	INDEX RANGE SCAN	IDX_PROCESS_PROCESSSTATE	1	667	00:00:00.01
* 13	TABLE ACCESS BY INDEX ROWID	TA_PROCESS	500	41	00:00:35.94
* 14	INDEX RANGE SCAN	IDX_PROCESS_PERSON	500	12M	00:00:07.53
* 15	TABLE ACCESS BY INDEX ROWID	TA_PROCESS	456	324	00:00:26.02
* 16	INDEX RANGE SCAN	IDX_PROCESS_PERSON	456	10M	00:00:06.09

```
Predicate Information (identified by operation id):
---------------------------------------------------
 …
  13 - filter((INTERNAL_FUNCTION("B"."PROCESS_STATE") AND
"B"."PROCESS_CREATION"<"A"."PROCESS_CREATION" AND
              "B"."PROCESS_CREATION">SYSDATE@!-:B2/24))
  14 - access("B"."PERSON_NUMBER"="A"."PERSON_NUMBER" AND "B"."CS_ID"="A"."CS_ID")
  15 - filter(("B"."PROCESS_STATE"<>8 AND "B"."PROCESS_STATE"<>5 AND "B"."PROCESS_STATE"<>10 AND
              "B"."PROCESS_CREATION"<"A"."PROCESS_CREATION" AND "B"."PROCESS_CREATION">SYSDATE@!-
:B3/24))
  16 - access("B"."PERSON_NUMBER"="A"."PERSON_NUMBER" AND "B"."CS_ID"="A"."CS_ID")
```

Fig. 5.6 A nonselective index. Example 1

Have you noticed that the cardinality in the 13th step falls from 12M to 41?"

P.: "Do you mean that that happens in this step due to the filter?"

A.: "Exactly."

P.: "That indicates a strong selectivity of these filter predicates. I would create an index for the relevant columns."

A.: "That might solve the problem, but I have one objection to this solution. In the section 'The Guideline: Tuning Without Changing the SQL Statement,' we have already learned that a new index is more likely to influence other execution plans than an extension of the existing index."

P.: "So you are for an index extension?"

A.: "Yes, if that is possible, as it is in our case. The index IDX_PROCESS_PERSON had two columns: PERSON_NUMBER and CS_ID. It has been extended to include the column PROCESS_CREATION because precisely this column was selective. The extended index simultaneously improved step 16 (see the filter for this step) and reduced the total runtime to a fraction of a second. Was this analysis complicated for you?"

P.: "On the contrary."

A.: "Then I suggest that you carry out the next analysis yourself. The next example is interesting for two reasons. It shows that a 'local' problem in the execution plan can have a very negative effect on the runtime. In contrast to example 1, the second execution plan with runtime statistics

www.allitebooks.com

SQL Plan Monitoring Details (Plan Hash Value=2028435987)

Id	Operation	Name	Execs	Rows (Actual)	Activity (%)
0	INSERT STATEMENT		1		
1	LOAD TABLE CONVENTIONAL		1		
2	NESTED LOOPS		1		
3	NESTED LOOPS		1		
4	NESTED LOOPS ANTI		1	0	
5	NESTED LOOPS		1	1241	
6	NESTED LOOPS		1	1241	
7	NESTED LOOPS		1	1241	
8	TABLE ACCESS FULL	FX_TEMP_RISKFONDS	1	1241	
9	TABLE ACCESS BY INDEX ROWID	KT_AN_BAV_AKTEN	1241	1241	
10	INDEX UNIQUE SCAN	KTABA_PK	1241	1241	
11	TABLE ACCESS BY INDEX ROWID	AN_GRUPPEN	1241	1241	
12	INDEX UNIQUE SCAN	ANGRU_PK	1241	1241	
13	INDEX UNIQUE SCAN	FPFON_PK	1241	1241	
14	TABLE ACCESS BY INDEX ROWID	FX_BALANCE	1241	1241	81.43
15	INDEX RANGE SCAN	FXBAL_FPFON_FK	1241	1G	18.44
16	INDEX RANGE SCAN	KTVAN_KTABA_FK_I			
17	TABLE ACCESS BY INDEX ROWID	KT_ANVERMKONTEN			

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

  …
  14 - SEL$5DA710D3 / B@SEL$2
  …
Predicate Information (identified by operation id):
---------------------------------------------------

  …
  14 - filter(("B"."FXBAL_ID"=:B2 AND "RISKFONDS"."KTABA_ID"="B"."KTABA_ID"))
  15 - access("RISKFONDS"."FPFON_ID"="B"."FPFON_ID")
  …
```

Fig. 5.7 A nonselective index. Example 2

was taken from a SQL monitoring report. The runtime of the following execution plan was 770 seconds (Fig. 5.7)."

P.: "The differences between the SQL monitoring report and the output of dbms_xplan.display_cursor are not so great. 'Execs' means 'starts' and 'rows (actual)' is equivalent to 'A-rows.' Where did you get the predicates from?"

A.: "These predicates were obtained additionally with dbms_xplan.display_cursor because they are missing in the SQL monitoring report."

P.: "The step with the highest cardinality is 15. 1G rows with an index range scan of FXBAL_FPFON_FK were found in this step. I assume that this step is relatively expensive. Step 14, in which the table FX_BALANCE is read by rowid, is even more expensive, however (at least according to the information in the column 'Activity (%)'). The cardinality of this step falls from 1G to 1241. Therefore, the filter in step 14 should be very selective. I would extend the index FXBAL_FPFON_FK by the relevant columns. However, I don't know which columns from the filter belong to table FX_BALANCE."

A.: "You can see from the plan section 'Query Block Name/Object Alias' that the table FX_BALANCE has the alias B. Accordingly, one must extend the index to include the columns FXBAL_ID and KTABA_ID. As the index FXBAL_FPFON_FK only had a single column FPFON_ID, it was possible to extend this index by two columns without difficulty. This improved the runtime to 0.2 seconds. Your analysis is therefore correct."

5.1.3 An Index with a Large Clustering Factor

Author: "Peter, we discussed in [1] what the clustering factor is in detail. Could you tell us briefly what clustering factor means and why it's important for SQL tuning?"

Peter: "I can try. The clustering factor is an optimizer statistic which Oracle calculates with an index full scan for the relevant index as follows: When two successive index entries refer to rows which are located in different table blocks, the value of the clustering factor is increased by 1. If they belong to the same block, this value remains unchanged. The initial value of the clustering factor in this calculation is equal to 1."

A.: "How can one use the clustering factor in SQL tuning?"

P.: "This statistic shows how far apart index entries and corresponding table entries are."

A.: "I think I know what you mean. But your interpretation is not quite right. For which operations can the clustering factor be important?"

P.: "In [1] we see that the clustering factor is equal to the number of buffer gets in table accesses by rowid after a full index scan. With the clustering factor, one can assess the effectiveness of table accesses by rowid after a full or range index scan. The larger the clustering factor, the less effective such table accesses are."

A.: "Correct. What is a good value and what is a bad value for the clustering factor?"

P.: "A good value is close to the number of filled table blocks. A poor value is close to the number of index entries."

A.: "Very good, Peter. Can one recognize an index with a large clustering factor in the execution plan?"

P.: "To do that, I would compare the cardinality of the index scan with the number of buffer gets of the subsequent table access by rowid. If these values are close together, the clustering factor of the index is presumably large."

A.: "The extract from an execution plan demonstrates that (Fig. 5.8). We will describe this example in full in the section 'Joins with a Large Hit Quantity.'

P.: "The number of buffer gets in step 18 is even larger than the cardinality in step 19. How is that possible?"

A.: "You've forgotten that the runtime statistic 'buffers' is cumulative (see section 'Runtime Statistics'). This means that only 4252K – 162K = 4090K buffer gets have occurred with table access by rowid. This number is smaller than the cardinality of 4159K in step 19."

P.: "Can one reduce the clustering factor of an index?"

Id	Operation	Name	Starts	A-Rows	A-Time	Buffers
0	SELECT STATEMENT		1	28	00:09:22.19	5161K
	…					
18	TABLE ACCESS BY INDEX ROWID	PICKAUF	137K	4159K	00:07:58.00	4252K
* 19	INDEX RANGE SCAN	PI_PR_FK_I	137K	4159K	00:00:06.51	162K
	…					

Fig. 5.8 The index PI_PR_FK_I presumably has a large clustering factor

A.: "Yes, if one enters the data into the table assorted in the same way as in the index. This solution has some disadvantages and is not always practicable (see [1]). In some situations, one can extend the relevant index to render table access by rowid unnecessary ('index only' access). This eliminates the problematical step in the execution plan."

5.1.4 A Sparse Index

Author: "A sparse index has considerably more leaf blocks than necessary. Its leaf blocks are sparsely filled with data. Some blocks may even be empty. Sparse indexes, their identification, and performance problems which they cause are described in [1]. Peter, during which operations can sparse indexes cause performance problems?"

Peter: "During index scans, because Oracle has to read more blocks than necessary."

A.: "How can one identify whether an index has too many leaf blocks?"

P.: "That can be done by means of optimizer statistics. With these statistics one can calculate the number of leaf blocks necessary for the index and compare this number with the relevant index statistic 'leaf blocks.' The script sparse_norm_idx9i.sql, which one can download from the Internet website www.tutool.de/book, functions according to this principle. There one can also find the second script estimate_sparse_norm_idx10g.sql, which needs no optimizer statistics but obtains all necessary data for the calculation itself. This script is more precise than the first one but needs considerably more time and resources. Both scripts are described in [1]."

A.: "One has to be careful with the index statistic 'leaf blocks' because it only takes into account the filled index blocks. Leaf blocks which are totally empty are omitted from this statistic (see [1])."

P.: "When an index has a lot of empty leaf blocks, the difference between the number of all allocated index blocks from the view DBA_SEGMENTS for this index and the index statistic 'leaf blocks' must be considerable. One can identify such empty leaf blocks in this way."

Id	Operation	Name	Starts	A-Rows	Buffers
0	SELECT STATEMENT		1	119	1774
1	SORT ORDER BY		1	119	1774
* 2	FILTER		1	119	1774
* 3	TABLE ACCESS BY INDEX ROWID	PROVIS_TRANSACTION	1	119	1774
* 4	INDEX RANGE SCAN	PROVIS_TRANSACTION_1IX	1	326	1751

```
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(:B1>=:B0)
   3 - filter(("GROUP_ID">=:B0 AND "GROUP_ID"<=:B1))
   4 - access("PROCESS_IND"=:B2)
```

Fig. 5.9 The index PROVIS_TRANSACTION_1IX is presumably a sparse index

A.: "Correct, Peter. But let's return to the execution plan. How would you recognize a sparse index in an index scan there?"

P.: "When an index has too many leaf blocks, Oracle has to read a large number of blocks during the index scan. Normally, in this case, the runtime statistic 'buffers' is relatively large compared to the cardinality."

A.: "I have a little bit to add to what you have just said. The statistic cardinality contains the number of selected rows. It is also possible that many rows are accessed in an index scan but only a few are selected."

P.: "I don't completely understand that."

A.: "Please think back to the predicates in 'access' and in 'filter' from the section 'Predicate Information.' The data is accessed with the predicates in 'access.' If no filter predicates exist to the relevant index scan, then the cardinality is almost equal to the number of accessed rows. In this case, one can compare the cardinality and the statistic 'buffers' in order to identify a sparse index, as in a practical example shown in Fig. 5.9. Can you recognize a sparse index there?"

P.: "That's very easy. In step 4, 326 rows were found by the index range scan. Oracle did 1751 buffer gets for that. The index PROVIS_TRANSACTION_1IX seems to me to be a sparse index. Does one have to check every index scan in the execution plan? That could be quite a lot."

A.: "Not each one, of course. Only problematical execution plan steps are interesting for us. Those are steps with a large runtime (column 'A-Time' in an execution plan with runtime statistics) or with a large activity (column 'Activity (%)' in a SQL monitoring report)."

P.: "Has the index PROVIS_TRANSACTION_1IX been rebuilt?"

A.: "Yes, you can see the result in Fig. 5.10."

P.: "Can one rebuild every sparse index without problems?"

A.: "When there is competing access to an index, its rebuilding can cause serious waiting time on 'latch: cache buffers chains' (see [1]). This is not the only possible negative effect of a rebuilding of an index, but precisely this can have a severely adverse effect on performance. Peter, I have already indicated that a large number of buffer gets and a relatively low cardinality in an index range scan don't always indicate a sparse index."

Id	Operation	Name	Starts	A-Rows	Buffers
0	SELECT STATEMENT		1	119	24
1	SORT ORDER BY		1	119	24
* 2	FILTER		1	119	24
* 3	TABLE ACCESS BY INDEX ROWID	PROVIS_TRANSACTION	1	119	24
* 4	INDEX RANGE SCAN	PROVIS_TRANSACTION_1IX	1	326	1

Fig. 5.10 The execution plan after the index rebuilding of PROVIS_TRANSACTION_1IX

```
select /*+ index(t1 i1_t1) no_index_ss(t1) */ count(*) from t1 where
a=1 and c between 100 and 110

Plan hash value: 2778874372
```

Id	Operation	Name	Starts	A-Rows	Buffers
0	SELECT STATEMENT		1	1	160
1	SORT AGGREGATE		1	1	160
* 2	INDEX RANGE SCAN	I1_T1	1	11	160

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("A"=1 AND "C">=100 AND "C"<=110)
       filter(("C"<=110 AND "C">=100))
```

Fig. 5.11 Selective filter with index range scan

P.: "Unfortunately, I can't give you any other explanation."

A.: "When the relevant index scan also has filter predicates, precisely these (i.e., no predicates with access) can be selective and contribute a great deal to a low cardinality. In this case, it can have nothing to do with a sparse index."

P.: "What do you mean by that?"

A.: "I'll show you with an example. We'll create a table T1 and fill it with data. All values in column A are equal to 1. We will fill column C in a way that makes this column very selective. The values in the remaining columns are not important for us. Let's create an index for three columns:

```
SQL> create index i1_t1 on t1(a,b,c);

Index created.

SQL> select leaf_blocks from user_indexes where index_name='I1_T1';

LEAF_BLOCKS
-----------
        159
```

After that we will execute the following SQL statement (Fig. 5.11).

As you can see, all leaf blocks have been read by the index range scan. The number of selected rows is low thanks to the high selectivity of column C. The rows have been filtered out by the filter (($C \leq 110$ AND $C \geq 100$))."

P.: "It does actually look very similar to a sparse index. With the hint NO_INDEX_SS, you have suppressed an index skip scan. Why?"

A.: "We are mainly concentrating on index range scan, because it occurs more often than other index scans in practice. When the statistic 'buffers' is large in comparison to the cardinality of an index skip scan, this normally also indicates that either the relevant index is a sparse index or the index skip scan is simply not efficient, although the associated filter is selective. Because each index skip scan has a filter, one has to check both

```
select /*+ index(t1 i1_t1) */ count(*) from t1 where
a=1 and c between 100 and 110

Plan hash value: 2778874372
```

Id	Operation	Name	Starts	A-Rows	Buffers
0	SELECT STATEMENT		1	1	175
1	SORT AGGREGATE		1	1	175
* 2	INDEX RANGE SCAN	I1_T1	1	11	175

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("A"=1)
       filter((TO_NUMBER("C")>=100 AND TO_NUMBER("C")<=110))
```

Fig. 5.12 Selective filter with index range scan due to type conversion

```
select /*+ index(t1 i_t1) no_index_ss(t1 i_t1) gather_plan_statistics
*/  count(*) from t1 where a between 1 and 3 and b = 20000

Plan hash value: 3547404373
```

Id	Operation	Name	Starts	A-Rows	Buffers
0	SELECT STATEMENT		1	1	105
1	SORT AGGREGATE		1	1	105
* 2	INDEX RANGE SCAN	I_T1	1	3	105

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("A">=1 AND "B"=20000 AND "A"<=3)
       filter("B"=20000)
```

Fig. 5.13 Selective filter with index range scan with a wide "between" interval for the leading column in the index

these possibilities. At the end of this section, I will demonstrate by means of an example that, in some cases of the selective filter, an index skip scan is more effective than an index range scan."

P.: "Can one solve problems with a selective filter by creating an index for the filter predicates?"

A.: "It's possible that the selective index column can only be checked in filter and not in access, due to a type conversion. Let's change the type of column C to VARCHAR2 and fill table T1 with data so that the values in column C are numerical. In this example, index I1_T1 has only two columns: A and C. Let's execute the SQL statement (Fig. 5.12).

By creating an FBI (function-based index) for the filter predicates, one can no doubt also reduce the number of buffer gets. However, it makes more sense to change the problematical data type in the table. One has to analyze each problematical index scan with a selective filter separately and initiate appropriate improvement measures."

P.: "You wanted to show me that an index skip scan can be more effective than an index range scan."

A.: "Exactly. Look at Fig. 5.13. Table T1 has an index for columns A and B. Column A, which is the leading column in this index, is not selective and has only four different values: 1, 2, 3, 4. Can you say why so many index blocks have been scanned in comparison to cardinality?"

```
select /*+ index_ss(t1 i_t1) gather_plan_statistics */  count(*) from
t1 where a between 1 and 3  and b = 20000

Plan hash value: 812462047
```

Id	Operation	Name	Starts	A-Rows	Buffers
0	SELECT STATEMENT		1	1	4
1	SORT AGGREGATE		1	1	4
* 2	INDEX SKIP SCAN	I_T1	1	3	4

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("A">=1 AND "B"=20000 AND "A"<=3)
       filter("B"=20000)
```

Fig. 5.14 Index skip scan is more efficient than index range scan when there is a wide "between" interval of the leading column in the index, if this column only has a few different values

P.: "As column A has only four different values, the interval between 1 and 3 is relatively wide, so Oracle has to read a lot of leaf blocks when carrying out the index range scan. The data from these blocks are filtered out with the filter predicate 'B=20000.' The cardinality is presumably so low because this filter is selective. But I don't understand how an index skip scan can help here. I would have created an index for the column from the filter in such a situation."

A.: "Imagine that the index I_T1 is quasi-partitioned for column A. Each of these partitions has the same value for column A. In our example there are four partitions. With index skip scan, Oracle executes an index range scan in each of the partitions with 'access (A=<I> and B=20000),' whereby I = 1, 2, 3. As the condition B=20000 is selective, this access is very efficient. For index skip scan, Oracle only has to carry out three such index range scans. This explains why the index skip scan is very efficient (Fig. 5.14). I find this a much more elegant solution than to create a new index."

5.1.5 Nested Loop Join Instead of Hash Join and Vice Versa

Author: "In this section we will discuss how one can recognize in the execution plan that a hash join is more effective than an applied nested loop join and vice versa."

Peter: "Doesn't that happen automatically in Oracle?"

A.: "You are right. Oracle uses the 'adaptive plan' feature for this. Oracle already calculates both variants (hash join and nested loop join) during parsing and only decides on one of these variants during runtime. It is important to know that these join methods are not always an alternative to each other."

P.: "Why?"

A.: "In contrast to nested loop join, the hash join can only be used with equi-joins."

P.: "Now I understand: The hash join is based on a hash algorithm for which join conditions with equalities are the only option."

A.: "Correct. Before we analyze the nested loop join and the hash join in the execution plan, we have to agree on the terminology. A nested loop join consists of two cycles: Outer loop and inner loop. In the first cycle, the rows are read from the outer table. For each row selected in the first cycle, the second cycle is executed. The data in the second cycle is selected from the second table (inner table) with the join condition. In the case of the hash join, this functions as follows: Firstly, a hash table is formed for the join columns from the first table (build table) in the memory. Then all rows from the second table (probe table) are checked against the hash table. Peter, could you please tell me under which conditions a nested loop join makes sense."

P.: "I think the cardinality of the outer loop should be relatively low. Otherwise, too many inner loops will be executed."

A.: "That's correct, but not enough."

P.: "I think I know which condition is still missing. The cardinality of the inner loop must also be low. The lower it is, the more effective the nested loop join is."

A.: "Very good, Peter. When the cardinality of the outer or inner loop is high, the relevant nested loop join is not efficient. If it is an equi-join, one can try to use the hash join instead of the nested loop join. Let's take the following execution plan of a SQL statement from Oracle's SYSMAN schema as an example (Fig. 5.15).

The runtime of the relevant SQL statement with this execution plan was 573 seconds. Can you please analyze this plan, Peter?"

P.: "The problematical steps in this plan are 18 and 19. In step 19, 584 million rows were selected. For each of these rows, the table access to the table EM_METRIC_ITEMS was done by rowid in step 18. The cardinality sank to eight million. This step is the most expensive in the execution plan. We could try to reduce the cardinality in step 19."

A.: "Peter, I fear that your analysis is heading in the wrong direction. Even though you're right, we can't reduce the cardinality in step 19 because SYSMAN is a schema from Oracle itself, and we are not allowed to make any changes there, such as index extension. Please concentrate on nested loops."

P.: "OK. Steps 18 and 19 belong to the nested loop join in step 6. In this nested loop join, 966 rows from the join in step 7 were linked to the table EM_METRIC_ITEMS. According to the relevant predicates, that is an equi-join. The cardinality of the inner loop in this join was eight million (see step 18). As that is a large value, one could try to replace the nested loop by a hash join."

SQL Plan Monitoring Details (Plan Hash Value=2535171835)

Id	Operation	Name	Execs	Rows (Actual)	Activity (%)
0	SELECT STATEMENT		1	25	
1	VIEW	GC_METRIC_LATEST	1	25	
2	UNION-ALL		1	25	
3	FILTER		1	23	
4	NESTED LOOPS		1	23	
5	NESTED LOOPS		1	23	
6	NESTED LOOPS		1	8M	
7	NESTED LOOPS		1	966	
8	HASH JOIN		1	966	
9	HASH JOIN		1	966	
10	TABLE ACCESS FULL	EM_METRIC_COLUMNS	1	1625	
11	NESTED LOOPS		1	3136	
12	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUPS	1	12	
13	INDEX RANGE SCAN	EM_METRIC_GROUPS_PK	1	12	
14	INDEX RANGE SCAN	EM_METRIC_COLUMN_VER_PK	12	3136	
15	TABLE ACCESS FULL	EM_METRIC_GROUP_VER	1	61661	
16	TABLE ACCESS BY INDEX ROWID	EM_METRIC_KEYS	966	966	
17	INDEX RANGE SCAN	EM_METRIC_KEYS_PK	966	1932	
18	TABLE ACCESS BY INDEX ROWID	EM_METRIC_ITEMS	966	8M	41.36
19	INDEX RANGE SCAN	EM_METRIC_ITEMS_KEY_IDX	966	584M	23.73
20	INLIST ITERATOR		8M	23	0.52
21	TABLE ACCESS BY INDEX ROWID	EM_MANAGEABLE_ENTITIES	38M	23	1.40
22	INDEX RANGE SCAN	EM_MANAGEABLE_ENTITIES_PK	38M	3M	24.78
23	PARTITION RANGE ITERATOR		33	23	
24	INDEX UNIQUE SCAN	EM_METRIC_VALUES_PK	33	23	
25	NESTED LOOPS				
26	TABLE ACCESS BY INDEX ROWID	EM_MEXT_TARGET_ASSOC			
27	INDEX UNIQUE SCAN	MEXT_TARGET_ASSOC_UN			
28	INDEX UNIQUE SCAN	MEXT_COLUMNS_PK			
29	FILTER		1	2	
30	NESTED LOOPS		1	2	
31	NESTED LOOPS		1	2	
32	NESTED LOOPS		1	2	
33	NESTED LOOPS		1	652K	
34	NESTED LOOPS		1	89	
35	HASH JOIN		1	89	
36	HASH JOIN		1	89	
37	TABLE ACCESS FULL	EM_METRIC_COLUMNS	1	11	
38	NESTED LOOPS		1	3136	
39	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUPS	1	12	
40	INDEX RANGE SCAN	EM_METRIC_GROUPS_PK	1	12	
41	INDEX RANGE SCAN	EM_METRIC_COLUMN_VER_PK	12	3136	
42	TABLE ACCESS FULL	EM_METRIC_GROUP_VER	1	61661	
43	TABLE ACCESS BY INDEX ROWID	EM_METRIC_KEYS	89	89	
44	INDEX RANGE SCAN	EM_METRIC_KEYS_PK	89	178	
45	TABLE ACCESS BY INDEX ROWID	EM_METRIC_ITEMS	89	652K	4.19
46	INDEX RANGE SCAN	EM_METRIC_ITEMS_KEY_IDX	89	54M	1.22
47	INLIST ITERATOR		652K	2	0.17
48	TABLE ACCESS BY INDEX ROWID	EM_MANAGEABLE_ENTITIES	3M	2	0.35
49	INDEX RANGE SCAN	EM_MANAGEABLE_ENTITIES_PK	3M	230K	2.27
50	INDEX UNIQUE SCAN	EM_METRIC_STRING_LATEST_PK	2	2	
51	TABLE ACCESS BY INDEX ROWID	EM_METRIC_STRING_LATEST	2	2	
52	NESTED LOOPS				
53	TABLE ACCESS BY INDEX ROWID	EM_MEXT_TARGET_ASSOC			

Fig. 5.15 An inefficient nested loop join

54	INDEX UNIQUE SCAN	MEXT TARGET ASSOC UN			
55	INDEX UNIQUE SCAN	MEXT_COLUMNS_PK			

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
  …
  12 - SEL$F32B35FB / G@SEL$2
  …
  18 - SEL$F32B35FB / I@GMVL
  19 - SEL$F32B35FB / I@GMVL
  …
  21 - SEL$F32B35FB / ME@SEL$3
  …
  45 - SEL$E18A34F2 / I@GMSVL
  46 - SEL$E18A34F2 / I@GMSVL
  …

Predicate Information (identified by operation id):
---------------------------------------------------
  …
  18 - filter(("I"."METRIC_GROUP_ID">=1 AND "I"."METRIC_GROUP_ID"="G"."METRIC_GROUP_ID" AND
("I"."IS_CURRENT"='1' OR "G"."KEYS_FROM_MULT_COLS"=1)

))
  19 - access("I"."METRIC_KEY_ID"="K"."METRIC_KEY_ID")
  …
```

Fig. 5.15 (continued)

A.: "The hash join can eliminate both problematical steps 18 and 19. According to the optimizer statistics, table EM_METRIC_ITEMS had approximately 7.5 million rows. As the table was not inordinately large, access to this table with an FTS was tried. With these two optimizer hints, the problematical nested loop join is replaced by a hash join:

```
full(@SEL$F32B35FB I@GMVL) use_hash(@SEL$F32B35FB I@GMVL)
```

In the execution plan, there is another similar nested loop join in step 33. Although this join was not as suboptimal as the first one, the decision here was also to use a hash join. The following hints were used:

```
full(@SEL$E18A34F2 I@GMSVL) use_hash(@SEL$E18A34F2 I@GMSVL)
```

These hash joins reduced the runtime to one-tenth (i.e., to 30–50 seconds). The execution plan in question is shown in Fig. 5.16."

P.: "I see that the two execution plans only differ in two joins. How is this possible?"

A.: "For the second execution, the outlines from the first plan were used. In these outlines, only the hints relevant to the nested loop joins were replaced by the four hints listed above for hash joins. This example shows how easy it is to accelerate a SQL statement tenfold. In the next section, we will continue with the tuning of this SQL statement."

P.: "I now know how an inefficient nested loop join can be recognized and replaced by a hash join. How can you recognize an inefficient hash join?"

A.: "Try to answer the question yourself."

P.: "The cardinality in a hash join when accessing the build table must be relatively low so that the outer loop of the nested loop join will be efficient. The hits of the hash join must also be low, so that the

SQL Plan Monitoring Details (Plan Hash Value=1326981145)

Id	Operation	Name	Execs	Rows (Actual)	Activity (%)
0	SELECT STATEMENT		1	25	2.86
1	VIEW	GC_METRIC_LATEST	1	25	
2	UNION-ALL		1	25	
3	FILTER		1	23	
4	NESTED LOOPS		1	23	
5	HASH JOIN		1	23	45.71
6	HASH JOIN		1	8M	20.00
7	NESTED LOOPS		1	966	
8	NESTED LOOPS		1	1932	
9	HASH JOIN		1	966	
10	HASH JOIN		1	966	
11	TABLE ACCESS FULL	EM_METRIC_COLUMNS	1	1625	
12	NESTED LOOPS		1	3136	
13	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUPS	1	12	
14	INDEX RANGE SCAN	EM_METRIC_GROUPS_PK	1	12	
15	INDEX RANGE SCAN	EM_METRIC_COLUMN_VER_PK	12	3136	
16	TABLE ACCESS FULL	EM_METRIC_GROUP_VER	1	61661	
17	INDEX RANGE SCAN	EM_METRIC_KEYS_PK	966	1932	
18	TABLE ACCESS BY INDEX ROWID	EM_METRIC_KEYS	1932	966	
19	TABLE ACCESS FULL	EM_METRIC_ITEMS	1	8M	11.43
20	TABLE ACCESS FULL	EM_MANAGEABLE_ENTITIES	1	1	
21	PARTITION RANGE ITERATOR		23	23	
22	INDEX UNIQUE SCAN	EM_METRIC_VALUES_PK	23	23	
23	NESTED LOOPS				
24	TABLE ACCESS BY INDEX ROWID	EM_MEXT_TARGET_ASSOC			
25	INDEX UNIQUE SCAN	MEXT_TARGET_ASSOC_UN			
26	INDEX UNIQUE SCAN	MEXT_COLUMNS_PK			
27	FILTER		1	2	
28	NESTED LOOPS		1	2	
29	NESTED LOOPS		1	2	
30	NESTED LOOPS		1	2	
31	HASH JOIN		1	652K	2.86
32	NESTED LOOPS		1	89	
33	NESTED LOOPS		1	178	
34	HASH JOIN		1	89	
35	HASH JOIN		1	89	
36	TABLE ACCESS FULL	EM_METRIC_COLUMNS	1	11	
37	NESTED LOOPS		1	3136	
38	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUPS	1	12	
39	INDEX RANGE SCAN	EM_METRIC_GROUPS_PK	1	12	
40	INDEX RANGE SCAN	EM_METRIC_COLUMN_VER_PK	12	3136	
41	TABLE ACCESS FULL	EM_METRIC_GROUP_VER	1	61661	
42	INDEX RANGE SCAN	EM_METRIC_KEYS_PK	89	178	
43	TABLE ACCESS BY INDEX ROWID	EM_METRIC_KEYS	178	89	
44	TABLE ACCESS FULL	EM_METRIC_ITEMS	1	8M	
45	INLIST ITERATOR		652K	2	
46	TABLE ACCESS BY INDEX ROWID	EM_MANAGEABLE_ENTITIES	3M	2	
47	INDEX RANGE SCAN	EM_MANAGEABLE_ENTITIES_PK	3M	228K	14.29
48	INDEX UNIQUE SCAN	EM_METRIC_STRING_LATEST_PK	2	2	
49	TABLE ACCESS BY INDEX ROWID	EM_METRIC_STRING_LATEST	2	2	
50	NESTED LOOPS				
51	TABLE ACCESS BY INDEX ROWID	EM_MEXT_TARGET_ASSOC			
52	INDEX UNIQUE SCAN	MEXT_TARGET_ASSOC_UN			
53	INDEX UNIQUE SCAN	MEXT_COLUMNS_PK			

Fig. 5.16 The execution plan after replacing the nested loop joins by hash joins

cardinality of the inner join is also low. If these two conditions are fulfilled, one can replace the respective hash join by a nested loop."

A.: "Quite right. As this analysis is fairly simple, we don't need to give an example."

5.2 "Global" Problems in the Execution Plan

This section is dedicated to problems which normally have an adverse effect on several execution plan steps. As a rule, they are not as localized as the problems described in the previous section and therefore have a "global" character.

Author: "Peter, can you say which SQL statements or which execution plan steps are affected by 'global' problems?"

Peter: "I assume that these problems can come up in a join of several tables."

A.: "Absolutely correct. What can cause 'global' problems in a join of several tables?"

P.: "An inappropriate table order in the execution plan of this join?"

A.: "Right again. Could you be a bit more concrete about what a 'global' problem looks like?"

P.: "The number of hits in some execution plan steps is unnecessarily large. As this influences the performance of the subsequent execution plan steps, one can describe such a problem as 'global.'"

A.: "Thank you, Peter. In this section we will discuss problems which can occur due to an inappropriate table order in the execution plan of a join. We will learn how to recognize and change an inappropriate table order. I would like to start with a change of a table order in a join."

5.2.1 Formal Rules for Changing the Table Order in a Join

Author: "One must be careful when changing any table order in a join as one can easily create a Cartesian product."

Peter: "This happens when two tables are not joined."

A.: "Yes, there are no join predicates for two tables. Oracle then has to join each row of the first table with each row of the second table. The number of hits of the Cartesian product is $N1 \times N2$ rows, whereby N1 and N2 are the cardinalities of the first and second table, respectively."

P.: "Is a Cartesian product in an execution plan always a bad thing?"

A.: "Not always. The optimizer sometimes decides on a Cartesian product even if an execution plan would be possible without a Cartesian product."

P.: "In such a case, the cardinalities of the two tables involved in the Cartesian must be very low."

A.: "Correct. In most cases, however, a Cartesian product causes poor performance because its result is usually high or very high. For this reason, one should avoid this situation wherever possible. First let's look at inner joins and use a simple example to show how to move a table in the join sequence without giving rise to a Cartesian product. In the following SQL statement, table T4 is joined to T2 as well as T0 (Fig. 5.17).

```
SQL> explain plan set statement_id='TTT' into sys.plan_table for
  2  select /*+ leading(t0 t1 t2 t3 t4 t5) */ * from t0, t1, t2, t3, t4, t5
  3  where
  4  t0.a = t1.a and
  5  t1.b = t2.b and
  6  t2.c = t3.c and
  7  t2.d = t4.d and
  8  t4.e = t5.e and
  9  t4.h = t0.h and
 10  t0.f = :b1 and
 11  t4.j = :b2;

Plan hash value: 392760849
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	624	15 (20)	00:00:01
* 1	HASH JOIN		1	624	15 (20)	00:00:01
* 2	HASH JOIN		1	520	12 (17)	00:00:01
* 3	HASH JOIN		1	416	10 (20)	00:00:01
* 4	HASH JOIN		1	312	7 (15)	00:00:01
* 5	HASH JOIN		1	208	5 (20)	00:00:01
* 6	TABLE ACCESS FULL	T0	1	104	2 (0)	00:00:01
7	TABLE ACCESS FULL	T1	82	8528	2 (0)	00:00:01
8	TABLE ACCESS FULL	T2	82	8528	2 (0)	00:00:01
9	TABLE ACCESS FULL	T3	82	8528	2 (0)	00:00:01
* 10	TABLE ACCESS FULL	T4	1	104	2 (0)	00:00:01
11	TABLE ACCESS FULL	T5	82	8528	2 (0)	00:00:01

Fig. 5.17 Inner join without Cartesian product

```
SQL> explain plan set statement_id='TTT' into sys.plan_table for
  2  select /*+ leading(t0 t1 t4 t2 t3 t5) */ * from t0, t1, t2, t3, t4, t5
  3  where
  4  t0.a = t1.a and
  5  t1.b = t2.b and
  6  t2.c = t3.c and
  7  t2.d = t4.d and
  8  t4.e = t5.e and
  9  t4.h = t0.h and
 10  t0.f = :b1 and
 11  t4.j = :b2;

Plan hash value: 281742934
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	624	15 (20)	00:00:01
* 1	HASH JOIN		1	624	15 (20)	00:00:01
* 2	HASH JOIN		1	520	12 (17)	00:00:01
* 3	HASH JOIN		1	416	10 (20)	00:00:01
* 4	HASH JOIN		1	312	7 (15)	00:00:01
* 5	HASH JOIN		1	208	5 (20)	00:00:01
* 6	TABLE ACCESS FULL	T0	1	104	2 (0)	00:00:01
7	TABLE ACCESS FULL	T1	82	8528	2 (0)	00:00:01
* 8	TABLE ACCESS FULL	T4	1	104	2 (0)	00:00:01
9	TABLE ACCESS FULL	T2	82	8528	2 (0)	00:00:01
10	TABLE ACCESS FULL	T3	82	8528	2 (0)	00:00:01
11	TABLE ACCESS FULL	T5	82	8528	2 (0)	00:00:01

Fig. 5.18 Movement of a table in a join without a Cartesian product

You see, Peter, there is no Cartesian product in the execution plan in Fig. 5.17. Can you please tell me why?”

P.: “Presumably because each table on the second, third position, etc. in this join is joined to one of the preceding tables.”

A.: “Very good. Let’s try to move table 4 without producing a Cartesian product. Can we move T4 to the third position (i.e., after table T1)?”

P.: “Yes, because it is joined to T0.”

A.: “The execution plan in Fig. 5.18 proves that you are right.”

P.: “I don’t think table sequences T0, T4, T1, T2, T3, T5 and T4, T0, T1, T2, T3, T5 should cause any Cartesian product either.”

A.: “That’s true. You can check that yourself. Are you now able to formulate a rule for the movement of a table in a join?”

P.: “I’ll try. One can move a table in a join without causing a Cartesian product if this table is joined to a preceding table in the new position. One can move a table to the first position if it is joined to the second table.”

A.: “Such a movement alone does not guarantee any improvement in performance. In the next sections, we will learn how to move a table in a join to achieve an improvement in performance.”

P.: “I have a question. In the case of a table chain in which each table is only joined to the next one, there are not so many variants for table movements. We can only reverse the table order. Is that right?”

A.: “Let’s look at that together. Let’s take the following table chain: T0 ⇒ T1 ⇒ T2 ⇒ T3 ⇒ T4. Here the symbol ‘⇒’ means a join. According to our rule, we can move T1 to the first position: T1, T0, T2, T3, T4. We can then either move T2 to the first position too or to the second position, i.e., T2, T1, T0, T3, T4 or T1, T2, T0, T3, T4, etc. As you see, there are quite a lot of variants for a table movement in a table chain. I therefore see no reason to regard table chains as a special case. Figure 5.19 shows such a change in table sequence in a table chain.”

P.: “Agreed. Have we finished with inner joins?”

A.: “Yes, we can move on to outer joins. Can you remind us what those are please, Peter?”

P.: “The result of a left outer join (or a left join) of two tables T1 and T2 consists of the result of the inner join of these tables and the remaining rows of table T1, for which the inner join produces no result (in this case, the column values of table T2 contain null values). A right outer join (or a right join) of two tables T1 and T2 is a left join of T2 and T1. A full outer join of two tables T1 and T2 is a combination of a left and a right join of these tables.”

```
SQL> explain plan set statement_id='TTT' into sys.plan_table for
  2  select /*+ leading(t3 t2 t1 t0 t4) */ * from t0, t1, t2, t3, t4
  3  where
  4  t0.a = t1.a and
  5  t1.b = t2.b and
  6  t2.c = t3.c and
  7  t3.d = t4.d and
  8  t0.e = :b1 and
  9  t4.f = :b2;

Plan hash value: 3764228295
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	520	12 (17)	00:00:01
* 1	HASH JOIN		1	520	12 (17)	00:00:01
* 2	HASH JOIN		1	416	10 (20)	00:00:01
* 3	TABLE ACCESS FULL	T0	1	104	2 (0)	00:00:01
* 4	HASH JOIN		82	25584	7 (15)	00:00:01
5	TABLE ACCESS FULL	T1	82	8528	2 (0)	00:00:01
* 6	HASH JOIN		82	17056	5 (20)	00:00:01
7	TABLE ACCESS FULL	T3	82	8528	2 (0)	00:00:01
8	TABLE ACCESS FULL	T2	82	8528	2 (0)	00:00:01
* 9	TABLE ACCESS FULL	T4	1	104	2 (0)	00:00:01

Fig. 5.19 Change of table order in a table chain

A.: "So it's enough if we only consider left and full outer join. A left join is a rigid construction in which no change of table order is possible (Fig. 5.20)."

P.: "What about the full outer join?"

A.: "One can change the table order there because that is a symmetrical operation (see Fig. 5.21). This change doesn't do anything to improve performance, however."

P.: "Then the change of table order is no option for outer joins. It is either impossible or of no use."

A.: "In principle you're right. But if we have a combination of inner and outer joins, there is quite a lot we can do. For example, we can move the complete outer join in the join. This is demonstrated in Figs. 5.22 and 5.23.

It is also possible to insert a table between the tables of an outer join in the table sequence. In the SQL statement in Fig. 5.24, table T4 is joined

```
SQL> explain plan set statement_id='TTT' into sys.plan_table for
  2  select /*+ leading(t2 t1) */ t1.a, t2.a from t1 left join t2 on (t1.a=t2.a);

Plan hash value: 1823443478
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		82	2132	5 (20)	00:00:01
* 1	HASH JOIN OUTER		82	2132	5 (20)	00:00:01
2	TABLE ACCESS FULL	T1	82	1066	2 (0)	00:00:01
3	TABLE ACCESS FULL	T2	82	1066	2 (0)	00:00:01

Fig. 5.20 Left join. No change in table order is possible

```
SQL> explain plan set statement_id='TTT' into sys.plan_table for
  2  select /*+ LEADING(@"SEL$1" "T2"@"SEL$1" "T1"@"SEL$1") */ t1.a, t2.a from t1 full join t2 on
(t1.a=t2.a);

Plan hash value: 3807180574
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		82	2132	5 (20)	00:00:01
1	VIEW	VW_FOJ_0	82	2132	5 (20)	00:00:01
* 2	HASH JOIN FULL OUTER		82	2132	5 (20)	00:00:01
3	TABLE ACCESS FULL	T2	82	1066	2 (0)	00:00:01
4	TABLE ACCESS FULL	T1	82	1066	2 (0)	00:00:01

Fig. 5.21 Full outer join

```
SQL> explain plan set statement_id='TTT' into sys.plan_table for
  2  select /*+ leading(t1 t2 t3) */ * from t1 inner join t2 on (t1.a = t2.a) left join t3 on
(t2.b = t3.b) where t1.c = :b1 and t2.c = :b2;

Plan hash value: 133157483
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	312	7 (15)	00:00:01
* 1	HASH JOIN OUTER		1	312	7 (15)	00:00:01
* 2	HASH JOIN		1	208	5 (20)	00:00:01
* 3	TABLE ACCESS FULL	T1	1	104	2 (0)	00:00:01
* 4	TABLE ACCESS FULL	T2	1	104	2 (0)	00:00:01
5	TABLE ACCESS FULL	T3	82	8528	2 (0)	00:00:01

Fig. 5.22 Movement of outer join (1)

```
SQL> explain plan set statement_id='TTT' into sys.plan_table for
  2  select /*+ leading(t2 t3 t1) */ * from t1 inner join t2 on (t1.a = t2.a) left join t3 on
(t2.b = t3.b) where t1.c = :b1 and t2.c = :b2;

Plan hash value: 910709849
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	312	7 (15)	00:00:01
* 1	HASH JOIN		1	312	7 (15)	00:00:01
* 2	HASH JOIN OUTER		1	208	5 (20)	00:00:01
* 3	TABLE ACCESS FULL	T2	1	104	2 (0)	00:00:01
4	TABLE ACCESS FULL	T3	82	8528	2 (0)	00:00:01
* 5	TABLE ACCESS FULL	T1	1	104	2 (0)	00:00:01

Fig. 5.23 Movement of outer join (2)

```
SQL> explain plan set statement_id='TTT' into sys.plan_table for
  2  select /*+ leading(t1 t2 t3 t4) */ * from t1 inner join t2 on (t1.a = t2.a) left join t3 on
(t2.b = t3.b) inner join t4 on (t1.c = t4.c) where t1.c = :b1 and t2.c = :b2;

Plan hash value: 2335041112
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	416	10 (20)	00:00:01
* 1	HASH JOIN		1	416	10 (20)	00:00:01
* 2	HASH JOIN OUTER		1	312	7 (15)	00:00:01
* 3	HASH JOIN		1	208	5 (20)	00:00:01
* 4	TABLE ACCESS FULL	T1	1	104	2 (0)	00:00:01
* 5	TABLE ACCESS FULL	T2	1	104	2 (0)	00:00:01
6	TABLE ACCESS FULL	T3	82	8528	2 (0)	00:00:01
* 7	TABLE ACCESS FULL	T4	1	104	2 (0)	00:00:01

Fig. 5.24 Placement of a table between the tables of an outer join (1)

```
SQL> explain plan set statement_id='TTT' into sys.plan_table for
  2  select /*+ leading(t1 t2 t4 t3) */ * from t1 inner join t2 on (t1.a = t2.a) left join t3 on
(t2.b = t3.b) inner join t4 on (t1.c = t4.c) where t1.c = :b1 and t2.c = :b2;

Plan hash value: 1892448498
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	416	10 (20)	00:00:01
* 1	HASH JOIN OUTER		1	416	10 (20)	00:00:01
* 2	HASH JOIN		1	312	7 (15)	00:00:01
* 3	HASH JOIN		1	208	5 (20)	00:00:01
* 4	TABLE ACCESS FULL	T1	1	104	2 (0)	00:00:01
* 5	TABLE ACCESS FULL	T2	1	104	2 (0)	00:00:01
* 6	TABLE ACCESS FULL	T4	1	104	2 (0)	00:00:01
7	TABLE ACCESS FULL	T3	82	8528	2 (0)	00:00:01

Fig. 5.25 Placement of a table between the tables of an outer join (2)

to table T1 with an inner join. We can therefore insert this table between tables T2 and T3 (which are joined together with an outer join) without producing a Cartesian product (see Fig. 5.25). If table T4 has a high cardinality, however, I would advise against this movement: After table T4 has been joined with T1 and T2, the outer join with T3 is executed and produces at least as many rows as the cardinality of T4 in step 6. In this way, we also obtain a high cardinality in a further step of the execution plan (in step 7 in Fig. 5.25). This is precisely what happens in a practical example in the section ‘Joins with a Large Number of Hits.’

To conclude this section, I would like to mention that Oracle replaces outer joins with inner joins if possible (see an example in Fig. 5.26).”

```
SQL> explain plan set statement_id='TTT' into sys.plan_table for
  2  select t1.a, t2.a from t1 left join t2 on (t1.a=t2.a) where t2.b is not null;

Plan hash value: 2959412835
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		4	104	5 (20)	00:00:01
* 1	HASH JOIN		4	104	5 (20)	00:00:01
* 2	TABLE ACCESS FULL	T2	4	52	2 (0)	00:00:01
3	TABLE ACCESS FULL	T1	82	1066	2 (0)	00:00:01

Fig. 5.26 Oracle replaces outer joins with inner joins if possible (1)

```
SQL> explain plan set statement_id='TTT' into sys.plan_table for
  2  select t1.a, t2.a, t2.b, t3.b from t1 left join t2 on (t1.a=t2.a) inner join t3 on
(t2.b=t3.b);

Plan hash value: 261998084
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		82	4264	7 (15)	00:00:01
* 1	HASH JOIN		82	4264	7 (15)	00:00:01
* 2	HASH JOIN		82	3198	5 (20)	00:00:01
3	TABLE ACCESS FULL	T1	82	1066	2 (0)	00:00:01
4	TABLE ACCESS FULL	T2	82	2132	2 (0)	00:00:01
5	TABLE ACCESS FULL	T3	82	1066	2 (0)	00:00:01

Fig. 5.27 Oracle replaces outer joins with inner joins if possible (2)

P.: "The condition 't2.b is not null' changes a left join to an inner join."

A.: "Quite right. There is another example shown in Fig. 5.27. The replacement of outer joins by inner joins gives the optimizer more table sequence variants to consider during parsing. This is also advantageous for SQL tuning."

5.2.2 Joins with a Low Number of Hits

Author: "First let's consider joins with an inappropriate table order in the execution plan and with a low number of hits. The important thing is that this low number of hits does not result from an aggregation or similar operation."

Peter: "And what if an aggregation reduces the number of hits of the join?"

A: "Then one has to analyze the number of hits immediately before this aggregation."

P.: "Which operation is to reduce the cardinality then?"

A.: "The cardinality is to be reduced by a join."

P.: "Can a join with a small number of hits be accelerated?"

A.: "One can at least try. The chances of obtaining a considerable acceleration are fairly good in this case."

P.: "What do you have to do to achieve that?"

Id	Operation	Name	Starts	A-Rows
0	SELECT STATEMENT		1	1
1	SORT AGGREGATE		1	1
2	NESTED LOOPS		1	1
3	NESTED LOOPS		1	1
* 4	TABLE ACCESS FULL	T1	1	8001
* 5	INDEX RANGE SCAN	T2_I1	8001	1
* 6	TABLE ACCESS BY INDEX ROWID	T2	1	1

```
Predicate Information (identified by operation id):
---------------------------------------------------

   4 - filter("T1"."B"=10)
   5 - access("T1"."A"="T2"."A")
   6 - filter("T2"."B"=40)
```

Fig. 5.28 Inappropriate table order in a join of two tables

Id	Operation	Name	Starts	A-Rows
0	SELECT STATEMENT		1	1
1	SORT AGGREGATE		1	1
2	NESTED LOOPS		1	1
3	NESTED LOOPS		1	1
* 4	TABLE ACCESS FULL	T2	1	11
* 5	INDEX RANGE SCAN	T1_I1	11	1
* 6	TABLE ACCESS BY INDEX ROWID	T1	1	1

Fig. 5.29 Change of table order in the join reduces the cardinality of the execution plan steps

A.: "Change the table order in the join in such a way that the cardinality is as low as possible in each execution plan step."

P.: "That sounds a bit too general, though. According to what criterion should one change the table order?"

A.: "I would like to propose a simple heuristic method for this. Let's start with a nested loop join of two tables. Here we take a nested loop join in particular because fast joins mainly use this kind of join. What do you notice in the execution plan in Fig. 5.28?"

P.: "The cardinality in step 4 is 8001. In step 5, it falls to 1."

A.: "Can one achieve a low cardinality in each execution plan step (at least considerably lower than 8001) by changing the table order in this join?"

P.: "I don't know."

A.: "That's the right answer. The execution plan doesn't provide us with enough information to be able to say that for sure. We have to check. How would you do that, Peter?"

P.: "If we change the table sequence, table T2 will be queried first. I would check how high the cardinality of filter 'T2.B=40' is. If it is low, one can change the table order."

A.: "As a rule, I prefer checking directly with the hint LEADING."

P.: "Why?"

A.: "A direct check is easier and, for that reason, less error-prone for the tuner. It also provides more information which one can use during SQL tuning. So let's check directly with the hint LEADING(T2 T1) to see if the changed table order reduces the cardinality (Fig. 5.29)."

P.: "The cardinality is considerably lower."

A.: "Changing the table sequence in the join has been beneficial here. If the cardinality of the filter 'T2.B=40' had been high, the low cardinality of the join could have been explained by a very high selectivity of the join predicate 'T1.A=T2.A.' In this case, changing the table sequence would not have achieved anything."

P.: "OK, nested loop join of two tables is clear now."

A.: "Now let's consider an inner join of several tables. Let's look for the table access in the execution plan with the highest cardinality. We can call this one table A. If the cardinality of the join in which table A is involved is also high and only falls in the next step or over a number of steps, one can assume that the table order in this join is suboptimal. The first table in the subsequent course of the execution plan in which the cardinality falls when this table is joined can then be called table B. If one inserts table B before table A in the join order, one can often reduce the cardinalities of the relevant execution plan steps."

P.: "You're assuming that the cardinality of the joins will fall at some point. Why?"

A.: "It must fall at some point because the number of hits for the whole join is low."

P.: "I don't understand why the change in the table order described above should reduce the cardinalities of the execution plan steps."

A.: "Inner join's hits remain unaffected by any change to the table order. Let's look at a part of the join from the beginning until the join to table B. If we insert table B before table A, we don't change the cardinality of this join because we are remaining within the same join. If the cardinality of table B (and the cardinality of the subsequent tables in the join) is low after this rearrangement, we have achieved the desired result. This is very similar to the case of nested loop join of two tables, which we have already discussed."

P.: "In an inner join of two tables, one can change the table order without causing a Cartesian product. In a join of several tables, this is not always the case."

A.: "Yes, we must therefore check whether this rearrangement is possible without producing a Cartesian product."

P.: "Could it be that we do not only have to move a table B, but several tables, in order to avoid a Cartesian product?"

A.: "That's possible. In practice, however, table B often follows immediately after table A in the join, so that one only has to move table B."

P.: "I hope I understand how to handle an inner join now. What about outer joins?"

A.: "When outer joins occur in the join of several tables, we regard each outer join as an entity and take this into consideration when changing the table order as described in the section 'Formal Rules for Changing the Table Order in a Join.' I have prepared three typical examples which clearly demonstrate the heuristic method. Let's start with the example

we have already used in the section ‘Nested Loop Join Instead of Hash Join and Vice Versa’ (see Fig. 5.15). Below is an extract from the execution plan, with a list of the corresponding predicates:”

SQL Plan Monitoring Details (Plan Hash Value=2535171835)

Id	Operation	Name	Execs	Rows (Actual)	Activity (%)
0	SELECT STATEMENT		1	25	
1	VIEW	GC_METRIC_LATEST	1	25	
2	UNION-ALL		1	25	
3	FILTER		1	23	
4	NESTED LOOPS		1	23	
5	NESTED LOOPS		1	23	
6	NESTED LOOPS		1	8M	
7	NESTED LOOPS		1	966	
8	HASH JOIN		1	966	
9	HASH JOIN		1	966	
10	TABLE ACCESS FULL	EM_METRIC_COLUMNS	1	1625	
11	NESTED LOOPS		1	3136	
12	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUPS	1	12	
13	INDEX RANGE SCAN	EM_METRIC_GROUPS_PK	1	12	
14	INDEX RANGE SCAN	EM_METRIC_COLUMN_VER_PK	12	3136	
15	TABLE ACCESS FULL	EM_METRIC_GROUP_VER	1	61661	
16	TABLE ACCESS BY INDEX ROWID	EM_METRIC_KEYS	966	966	
17	INDEX RANGE SCAN	EM_METRIC_KEYS_PK	966	1932	
18	TABLE ACCESS BY INDEX ROWID	EM_METRIC_ITEMS	966	8M	41.36
19	INDEX RANGE SCAN	EM_METRIC_ITEMS_KEY_IDX	966	584M	23.73
20	INLIST ITERATOR		8M	23	0.52
21	TABLE ACCESS BY INDEX ROWID	EM_MANAGEABLE_ENTITIES	38M	23	1.40
22	INDEX RANGE SCAN	EM_MANAGEABLE_ENTITIES_PK	38M	3M	24.78
		…			

```
Predicate Information (identified by operation id):
---------------------------------------------------
   …
   13 - access("G"."TARGET_TYPE"=:41)
     filter(("G"."METRIC_GROUP_NAME"=:2 OR "G"."METRIC_GROUP_NAME"=:4 OR
"G"."METRIC_GROUP_NAME"=:9 OR "G"."METRIC_GROUP_NAME"=:11 OR "G"."METRIC_GROUP_NAME"=:14 OR
"G"."METRIC_GROUP_NAME"=:18 OR "G"."METRIC_GROUP_NAME"=:23 OR "G"."METRIC_GROUP_NAME"=:26 OR
"G"."METRIC_GROUP_NAME"=:29
   …
   18 - filter(("I"."METRIC_GROUP_ID">=1 AND "I"."METRIC_GROUP_ID"="G"."METRIC_GROUP_ID" AND
("I"."IS_CURRENT"='1' OR "G"."KEYS_FROM_MULT_COLS"=1)
   19 - access("I"."METRIC_KEY_ID"="K"."METRIC_KEY_ID")
   …
   21 - filter(("ME"."ENTITY_TYPE"=:41 AND "ME"."ENTITY_NAME"=:40 AND
"GV"."TYPE_META_VER"="ME"."TYPE_META_VER" AND ("GV"."CATEGORY_PROP_1"=' ' OR
 "GV"."CATEGORY_PROP_1"="ME"."CATEGORY_PROP_1") AND ("GV"."CATEGORY_PROP_2"=' ' OR
"GV"."CATEGORY_PROP_2"="ME"."CATEGORY_PROP_2") AND ("GV"."CATEGORY_PROP_3"=' ' OR
"GV"."CATEGORY_PROP_3"="ME"."CATEGORY_PROP_3") AND ("GV"."CATEGORY_PROP_4"=' ' OR
"GV"."CATEGORY_PROP_4"="ME"."CATEGORY_PROP_4") AND ("GV"."CATEGORY_PROP_5"=' ' OR
"GV"."CATEGORY_PROP_5"="ME"."CATEGORY_PROP_5")))
   …
  22 - access("I"."TARGET_GUID"="ME"."ENTITY_GUID" AND (("ME"."MANAGE_STATUS"=0 OR
"ME"."MANAGE_STATUS"=1 OR "ME"."MANAGE_STATUS"=2 OR "ME"."MANAGE_STATUS"=3 OR
"ME"."MANAGE_STATUS"=5)))
   …
```

P.: “I’m sorry, but I can’t follow that.”

A.: “Wait a minute, Peter. First let’s find tables A and B. Would you do that please.”

P.: “The highest cardinality of 584M occurs in step 19 with the join to table EM_METRIC_ITEMS (alias I@GMVL). Although it falls to 8M during access by rowid to table EM_METRIC_ITEMS in step 18, it still remains high. Table EM_METRIC_ITEMS is therefore table A. In step 21, the cardinality falls to 23 with the join to table EM_MANAGEABLE_ENTITIES (alias ME@SEL3). This is table B.”

A.: "Let's try inserting table EM_MANAGEABLE_ENTITIES before table EM_METRIC_ITEMS in the join. Is that possible without a Cartesian product?"

P.: "For this, table EM_MANAGEABLE_ITEMS has to be joined to a preceding table, but it isn't."

A.: "This table is implicitly joined to table EM_METRIC_GROUPS (alias G@SEL@2)). In step 13, table EM_METRIC_GROUPS is accessed via the predicate 'G.TARGET_TYPE=:41.' Table EM_MANAGEABLE_ENTITIES also has a filter 'ME. ENTITY_TYPE=:41' (see the predicates to step 21). These two tables are therefore joined via the bind variable 41."

P.: "After that, we can insert table EM_MANAGEABLE_ENTITIES immediately before table EM_MANAGEABLE_ITEMS in the join."

A.: "Presumably this would have been enough for an adequate improvement in performance. I took a different approach, however. As the cardinality falls from 8M to 23 with the join to table EM_MANAGEABLE_ENTITIES, I thought that the predicates 'ME. ENTITY_TYPE=:41' and 'ME.ENTITY_NAME=:40' (see the predicates to step 21) were so selective that we could begin the join directly with access to the table EM_MANAGEABLE_ENTITIES. I then joined table EM_MANAGEABLE_ENTITIES to table EM_METRIC_GROUPS to demonstrate how Oracle converts an implicit join to an explicit join. Only then was it the turn of table EM_METRIC_ITEMS. As this table is joined to table EM_METRIC_KEYS (see the predicates to step 19), which, in turn, is joined to the others, no Cartesian products were to be expected. This was confirmed by an explain plan, which I generated for the SQL statement with the hint LEADING(@"SEL$F32B35FB" "ME"@"SEL$3" "G"@"SEL$2" "I"@"GMVL"). I almost forgot that the relevant SQL statement is a union of two joins which have an identical structure and only differ in terms of bind variables. For this reason, I added a second hint LEADING(@"SEL$E18A34F2" "ME"@"SEL$6" "G"@"SEL$5" "I"@"GMSVL") and executed the SQL statement. The result can be seen in Fig. 5.30."

P.: "Great! The runtime is now approximately 14 seconds instead of 573 before."

A.: "But I still wasn't satisfied with this execution plan. Two unnecessarily expensive hash joins (in steps 12 and 38) took 13.24 of the 13.89 seconds of runtime. It made sense to replace these hash joins with the respective nested loop joins. For these nested loop joins to be efficient, the relevant inner loops had to use index scans. For this reason, I checked whether an index existed for the columns TARGET_GUID and METRIC_GROUP_ID of table EM_METRIC_ITEMS:"

Plan hash value: 1717020805

Id	Operation	Name	Starts	A-Rows	A-Time
0	SELECT STATEMENT		1	25	00:00:13.89
1	VIEW	GC_METRIC_LATEST	1	25	00:00:13.89
2	UNION-ALL		1	25	00:00:13.89
* 3	FILTER		1	23	00:00:08.32
4	NESTED LOOPS		1	23	00:00:08.32
5	NESTED LOOPS		1	23	00:00:08.32
6	NESTED LOOPS		1	966	00:00:08.17
* 7	HASH JOIN		1	23	00:00:08.08
* 8	TABLE ACCESS FULL	EM_METRIC_COLUMNS	1	1625	00:00:00.38
* 9	HASH JOIN		1	11	00:00:07.70
10	TABLE ACCESS BY INDEX ROWID	EM_METRIC_KEYS	1	2	00:00:00.01
* 11	INDEX RANGE SCAN	EM_METRIC_KEYS_PK	1	2	00:00:00.01
* 12	HASH JOIN		1	11	00:00:07.69
* 13	HASH JOIN		1	12	00:00:00.01
14	TABLE ACCESS BY INDEX ROWID	EM_MANAGEABLE_ENTITIES	1	1	00:00:00.01
* 15	INDEX RANGE SCAN	EM_MANAGEABLE_ENTITIES_UK1	1	1	00:00:00.01
16	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUPS	1	12	00:00:00.01
* 17	INDEX RANGE SCAN	EM_METRIC_GROUPS_PK	1	12	00:00:00.01
* 18	TABLE ACCESS FULL	EM_METRIC_ITEMS	1	7572K	00:00:04.56
* 19	INDEX RANGE SCAN	EM_METRIC_COLUMN_VER_PK	23	966	00:00:00.09
* 20	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUP_VER	966	23	00:00:00.15
* 21	INDEX UNIQUE SCAN	EM_METRIC_GROUP_VER_U1	966	966	00:00:00.03
22	PARTITION RANGE ITERATOR		23	23	00:00:00.01
* 23	INDEX UNIQUE SCAN	EM_METRIC_VALUES_PK	23	23	00:00:00.01
24	NESTED LOOPS		0	0	00:00:00.01
25	TABLE ACCESS BY INDEX ROWID	EM_MEXT_TARGET_ASSOC	0	0	00:00:00.01
* 26	INDEX UNIQUE SCAN	MEXT_TARGET_ASSOC_UN	0	0	00:00:00.01
* 27	INDEX UNIQUE SCAN	MEXT_COLUMNS_PK	0	0	00:00:00.01
* 28	FILTER		1	2	00:00:05.56
29	NESTED LOOPS		1	2	00:00:05.56
30	NESTED LOOPS		1	2	00:00:05.56
31	NESTED LOOPS		1	2	00:00:05.56
32	NESTED LOOPS		1	84	00:00:05.56
* 33	HASH JOIN		1	2	00:00:05.56
* 34	TABLE ACCESS FULL	EM_METRIC_COLUMNS	1	11	00:00:00.02
* 35	HASH JOIN		1	11	00:00:05.55
36	TABLE ACCESS BY INDEX ROWID	EM_METRIC_KEYS	1	2	00:00:00.01
* 37	INDEX RANGE SCAN	EM_METRIC_KEYS_PK	1	2	00:00:00.01
* 38	HASH JOIN		1	11	00:00:05.55
* 39	HASH JOIN		1	12	00:00:00.01
40	TABLE ACCESS BY INDEX ROWID	EM_MANAGEABLE_ENTITIES	1	1	00:00:00.01
* 41	INDEX RANGE SCAN	EM_MANAGEABLE_ENTITIES_UK1	1	1	00:00:00.01
42	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUPS	1	12	00:00:00.01
* 43	INDEX RANGE SCAN	EM_METRIC_GROUPS_PK	1	12	00:00:00.01
* 44	TABLE ACCESS FULL	EM_METRIC_ITEMS	1	7572K	00:00:02.36
* 45	INDEX RANGE SCAN	EM_METRIC_COLUMN_VER_PK	2	84	00:00:00.01
* 46	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUP_VER	84	2	00:00:00.01
* 47	INDEX UNIQUE SCAN	EM_METRIC_GROUP_VER_U1	84	84	00:00:00.01
* 48	INDEX UNIQUE SCAN	EM_METRIC_STRING_LATEST_PK	2	2	00:00:00.01
49	TABLE ACCESS BY INDEX ROWID	EM_METRIC_STRING_LATEST	2	2	00:00:00.01
50	NESTED LOOPS		0	0	00:00:00.01
51	TABLE ACCESS BY INDEX ROWID	EM_MEXT_TARGET_ASSOC	0	0	00:00:00.01
* 52	INDEX UNIQUE SCAN	MEXT_TARGET_ASSOC_UN	0	0	00:00:00.01
* 53	INDEX UNIQUE SCAN	MEXT_COLUMNS_PK	0	0	00:00:00.01

```
Predicate Information (identified by operation id):
---------------------------------------------------
…
  12 - access("I"."METRIC_GROUP_ID"="G"."METRIC_GROUP_ID" AND
"I"."TARGET_GUID"="ME"."ENTITY_GUID")
       filter(("I"."IS_CURRENT"='1' OR "G"."KEYS_FROM_MULT_COLS"=1))
…
  38 - access("I"."METRIC_GROUP_ID"="G"."METRIC_GROUP_ID" AND
"I"."TARGET_GUID"="ME"."ENTITY_GUID")
       filter(("I"."IS_CURRENT"='1' OR "G"."KEYS_FROM_MULT_COLS"=1))
```

Fig. 5.30 Example 1. The first improvement after changing the table order in the join

```
SQL> select INDEX_NAME, TABLE_NAME, COLUMN_NAME, COLUMN_POSITION from dba_ind_columns where table_name =
'EM_METRIC_ITEMS' order by index_name, COLUMN_POSITION;

INDEX_NAME                     TABLE_NAME                     COLUMN_NAME                    COLUMN_POSITION

------------------------------ ------------------------------ ------------------------------ ---------------

EM_METRIC_ITEMS_KEY_IDX        EM_METRIC_ITEMS                METRIC_KEY_ID                                1

EM_METRIC_ITEMS_PK             EM_METRIC_ITEMS                TARGET_GUID                                  1

EM_METRIC_ITEMS_PK             EM_METRIC_ITEMS                METRIC_GROUP_ID                              2

EM_METRIC_ITEMS_PK             EM_METRIC_ITEMS                METRIC_KEY_ID                                3

EM_METRIC_ITEMS_UN1            EM_METRIC_ITEMS                METRIC_ITEM_ID                               1
```

P.: "Where did you find these columns?"

A.: "Those are the columns from the predicates to the two hash joins. The above output shows that the index EM_METRIC_ITEMS_PK was of use for the inner loop."

P.: "Wasn't it also necessary to check that these columns are selective?"

A.: "Why? The cardinalities of the two joins were already known and were both 11 (see steps 12 and 38 in Fig. 5.30). I have added the following hints:

- LEADING(@"SEL$F32B35FB" "ME"@"SEL$3" "G"@"SEL$2" "I"@"GMVL")
- INDEX(@SEL$F32B35FB I@GMVL EM_METRIC_ITEMS_PK)
- USE_NL(@SEL$F32B35FB I@GMVL)
- LEADING(@"SEL$E18A34F2" "ME"@"SEL$6" "G"@"SEL$5" "I"@"GMSVL")
- INDEX(@SEL$E18A34F2 I@GMSVL EM_METRIC_ITEMS_PK)
- USE_NL(@SEL$E18A34F2 I@GMSVL)

and executed the SQL statement (Fig. 5.31)."

P.: "Fantastic! The SQL statement has become approximately 1000 times faster."

A.: "You will no doubt have noticed how easily we managed to do that. Let's take another example (see Fig. 5.32). Try to analyze it."

P.: "I first have to look for tables A and B. In this example, that's not so easy because several have the maximum cardinality of 24826. I don't know which step to take."

A.: "You should take the first step in which this cardinality occurs."

P.: "When you say the first you mean the first step that is executed."

A.: "That's right, Peter."

P.: "The cardinality of 24826 first occurs in step 6, in which an index range scan takes place via the index IQE_SV_CLIENT. As this index belongs to the table QUEUE_ENTRY (alias A0), it is table A in the join. In step 8, the cardinality falls to 846 when joined with the table PROCESSING_INFO (alias C0). This table is table B in the join."

A.: "Can we put table B before table A in the join?"

Plan hash value: 2260856612

Id	Operation	Name	Starts	A-Rows	A-Time
0	SELECT STATEMENT		1	25	00:00:00.59
1	VIEW	GC_METRIC_LATEST	1	25	00:00:00.59
2	UNION-ALL		1	25	00:00:00.59
* 3	FILTER		1	23	00:00:00.57
4	NESTED LOOPS		1	23	00:00:00.57
5	NESTED LOOPS		1	23	00:00:00.56
6	NESTED LOOPS		1	966	00:00:00.49
* 7	HASH JOIN		1	23	00:00:00.39
* 8	TABLE ACCESS FULL	EM_METRIC_COLUMNS	1	1625	00:00:00.38
* 9	HASH JOIN		1	11	00:00:00.01
10	TABLE ACCESS BY INDEX ROWID	EM_METRIC_KEYS	1	2	00:00:00.01
* 11	INDEX RANGE SCAN	EM_METRIC_KEYS_PK	1	2	00:00:00.01
12	NESTED LOOPS		1	11	00:00:00.01
13	NESTED LOOPS		1	11	00:00:00.01
* 14	HASH JOIN		1	12	00:00:00.01
15	TABLE ACCESS BY INDEX ROWID	EM_MANAGEABLE_ENTITIES	1	1	00:00:00.01
* 16	INDEX RANGE SCAN	EM_MANAGEABLE_ENTITIES_UK1	1	1	00:00:00.01
17	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUPS	1	12	00:00:00.01
* 18	INDEX RANGE SCAN	EM_METRIC_GROUPS_PK	1	12	00:00:00.01
* 19	INDEX RANGE SCAN	EM_METRIC_ITEMS_PK	12	11	00:00:00.01
* 20	TABLE ACCESS BY INDEX ROWID	EM_METRIC_ITEMS	11	11	00:00:00.01
* 21	INDEX RANGE SCAN	EM_METRIC_COLUMN_VER_PK	23	966	00:00:00.10
* 22	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUP_VER	966	23	00:00:00.07
* 23	INDEX UNIQUE SCAN	EM_METRIC_GROUP_VER_U1	966	966	00:00:00.01
24	PARTITION RANGE ITERATOR		23	23	00:00:00.01
* 25	INDEX UNIQUE SCAN	EM_METRIC_VALUES_PK	23	23	00:00:00.01
26	NESTED LOOPS		0	0	00:00:00.01
27	TABLE ACCESS BY INDEX ROWID	EM_MEXT_TARGET_ASSOC	0	0	00:00:00.01
* 28	INDEX UNIQUE SCAN	MEXT_TARGET_ASSOC_UN	0	0	00:00:00.01
* 29	INDEX UNIQUE SCAN	MEXT_COLUMNS_PK	0	0	00:00:00.01
* 30	FILTER		1	2	00:00:00.02
31	NESTED LOOPS		1	2	00:00:00.02
32	NESTED LOOPS		1	2	00:00:00.02
33	NESTED LOOPS		1	2	00:00:00.02
34	NESTED LOOPS		1	84	00:00:00.02
* 35	HASH JOIN		1	2	00:00:00.02
* 36	TABLE ACCESS FULL	EM_METRIC_COLUMNS	1	11	00:00:00.01
* 37	HASH JOIN		1	11	00:00:00.01
38	TABLE ACCESS BY INDEX ROWID	EM_METRIC_KEYS	1	2	00:00:00.01
* 39	INDEX RANGE SCAN	EM_METRIC_KEYS_PK	1	2	00:00:00.01
40	NESTED LOOPS		1	11	00:00:00.01
41	NESTED LOOPS		1	11	00:00:00.01
* 42	HASH JOIN		1	12	00:00:00.01
43	TABLE ACCESS BY INDEX ROWID	EM_MANAGEABLE_ENTITIES	1	1	00:00:00.01
* 44	INDEX RANGE SCAN	EM_MANAGEABLE_ENTITIES_UK1	1	1	00:00:00.01
45	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUPS	1	12	00:00:00.01
* 46	INDEX RANGE SCAN	EM_METRIC_GROUPS_PK	1	12	00:00:00.01
* 47	INDEX RANGE SCAN	EM_METRIC_ITEMS_PK	12	11	00:00:00.01
* 48	TABLE ACCESS BY INDEX ROWID	EM_METRIC_ITEMS	11	11	00:00:00.01
* 49	INDEX RANGE SCAN	EM_METRIC_COLUMN_VER_PK	2	84	00:00:00.01
* 50	TABLE ACCESS BY INDEX ROWID	EM_METRIC_GROUP_VER	84	2	00:00:00.01
* 51	INDEX UNIQUE SCAN	EM_METRIC_GROUP_VER_U1	84	84	00:00:00.01
* 52	INDEX UNIQUE SCAN	EM_METRIC_STRING_LATEST_PK	2	2	00:00:00.01
53	TABLE ACCESS BY INDEX ROWID	EM_METRIC_STRING_LATEST	2	2	00:00:00.01
54	NESTED LOOPS		0	0	00:00:00.01
55	TABLE ACCESS BY INDEX ROWID	EM_MEXT_TARGET_ASSOC	0	0	00:00:00.01
* 56	INDEX UNIQUE SCAN	MEXT_TARGET_ASSOC_UN	0	0	00:00:00.01
* 57	INDEX UNIQUE SCAN	MEXT_COLUMNS_PK	0	0	00:00:00.01

Fig. 5.31 Example 1. Improvement in execution plan after replacement of hash joins by nested loop joins

Id	Operation	Name	Starts	A-Rows	A-Time
1	SORT ORDER BY		1	0	00:00:00.39
* 2	FILTER		1	0	00:00:00.39
3	NESTED LOOPS		1	846	00:00:00.39
4	NESTED LOOPS OUTER		1	24826	00:00:00.15
5	TABLE ACCESS BY INDEX ROWID	QUEUE_ENTRY	1	24826	00:00:00.05
* 6	INDEX RANGE SCAN	IQE_SV_CLIENT	1	24826	00:00:00.01
* 7	INDEX UNIQUE SCAN	PK_QUEUE	24826	24826	00:00:00.09
* 8	TABLE ACCESS BY INDEX ROWID	PROCESSING_INFO	24826	846	00:00:00.21
* 9	INDEX UNIQUE SCAN	PK_PROCESSING_INFO	24826	24826	00:00:00.11
10	NESTED LOOPS		14	0	00:00:00.01
* 11	INDEX UNIQUE SCAN	PK_QUEUE_RECIPIENTS	14	0	00:00:00.01
* 12	INDEX UNIQUE SCAN	PK_USER_	0	0	00:00:00.01
13	NESTED LOOPS		14	0	00:00:00.01
* 14	INDEX UNIQUE SCAN	PK_QUEUE_RECIPIENTS_EXTERNAL	14	0	00:00:00.01
* 15	INDEX UNIQUE SCAN	PK_USER_	0	0	00:00:00.01
16	NESTED LOOPS		14	0	00:00:00.01
17	NESTED LOOPS		14	42	00:00:00.01
18	NESTED LOOPS		14	42	00:00:00.01
* 19	INDEX UNIQUE SCAN	PK_USER_	14	14	00:00:00.01
20	TABLE ACCESS BY INDEX ROWID	GROUP_ALL_USERS	14	42	00:00:00.01
* 21	INDEX RANGE SCAN	IDX_GRP_LL_SRS_BSTRCT_ACTOR_ID	14	42	00:00:00.01
* 22	INDEX UNIQUE SCAN	PK_GROUP_	42	42	00:00:00.01
* 23	INDEX UNIQUE SCAN	PK_QUEUE_GROUPS	42	0	00:00:00.01

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(("A0"."OWNER_ID"=:B1 OR ("A0"."OWNER_ID" IS NULL AND ( IS NOT NULL OR  IS NOT NULL
OR  IS NOT NULL))))
   6 - access("A0"."SERVICE_CLIENT_ID"=:B5)
   7 - access("A0"."QUEUE_ID"="B0"."ABSTRACT_ACTOR_ID")
   8 - filter("C0"."PROCESSING_STATE"=:B6)
   9 - access("A0"."PROCESSING_INFO_ID"="C0"."PROCESSING_INFO_ID")
  11 - access("B0_SUB"."ABSTRACT_ACTOR_ID"=:B1 AND "B0_SUB"."RECIPIENT_ID"=:B2)
  12 - access("A0_SUB"."ABSTRACT_ACTOR_ID"=:B2)
  14 - access("B0_SUB"."ABSTRACT_ACTOR_ID"=:B1 AND "B0_SUB"."RECIPIENT_ID"=:B3)
  15 - access("A0_SUB"."ABSTRACT_ACTOR_ID"=:B3)
  19 - access("A0_SUB_SUB"."ABSTRACT_ACTOR_ID"=:B4)
  21 - access("B0_SUB_SUB"."ABSTRACT_ACTOR_ID"=:B4)
  22 - access("B0_SUB_SUB"."GROUP_ID"="A0_SUB"."GROUP_ID")
  23 - access("B0_SUB"."ABSTRACT_ACTOR_ID"=:B1 AND "B0_SUB"."GROUP_ID"="A0_SUB"."GROUP_ID")
```

Fig. 5.32 The second example of an inappropriate table order in a join with a small number of hits

P.: "Doesn't that hinder the outer join in step 4?"

A.: "That's not a problem because the table PROCESSING_INFO does not belong to this join."

P.: "Agreed. As table A is the first in the join, we have to move table B to the first position. Table A is then the second in the join. These tables have to be joined so that no Cartesian product occurs when the table order is changed."

A.: "Are they joined?"

P.: "According to the predicates to step 9, tables QUEUE_ENTRY and PROCESSING_INFO are joined by the column PROCESSING_INFO_ID. We can therefore change the table order accordingly with the hint LEADING(C0 A0)."

A.: "One moment please. The relevant SQL statement is fast. It only runs for 0.39 seconds with the poor execution plan. For this reason it makes sense to use a nested loop join to join the tables QUEUE_ENTRY and PROCESSING_INFO. For the inner loop to run this join optimally, the

Plan hash value: 471755289

Id	Operation	Name	Starts	A-Rows	A-Time
1	SORT ORDER BY		1	0	00:00:00.07
* 2	FILTER		1	0	00:00:00.07
3	NESTED LOOPS OUTER		1	840	00:00:00.07
* 4	HASH JOIN		1	840	00:00:00.06
5	TABLE ACCESS BY INDEX ROWID	PROCESSING_INFO	1	840	00:00:00.01
* 6	INDEX RANGE SCAN	IPROC_INFO_PS	1	840	00:00:00.01
7	TABLE ACCESS BY INDEX ROWID	QUEUE_ENTRY	1	24816	00:00:00.03
* 8	INDEX RANGE SCAN	IQE_SV_CLIENT	1	24816	00:00:00.01
* 9	INDEX UNIQUE SCAN	PK_QUEUE	840	840	00:00:00.01
10	NESTED LOOPS		14	0	00:00:00.01
* 11	INDEX UNIQUE SCAN	PK_QUEUE_RECIPIENTS	14	0	00:00:00.01
* 12	INDEX UNIQUE SCAN	PK_USER_	0	0	00:00:00.01
13	NESTED LOOPS		14	0	00:00:00.01
* 14	INDEX UNIQUE SCAN	PK_QUEUE_RECIPIENTS_EXTERNAL	14	0	00:00:00.01
* 15	INDEX UNIQUE SCAN	PK_USER_	0	0	00:00:00.01
16	NESTED LOOPS		14	0	00:00:00.01
17	NESTED LOOPS		14	42	00:00:00.01
18	NESTED LOOPS		14	42	00:00:00.01
* 19	INDEX UNIQUE SCAN	PK_USER_	14	14	00:00:00.01
20	TABLE ACCESS BY INDEX ROWID	GROUP_ALL_USERS	14	42	00:00:00.01
* 21	INDEX RANGE SCAN	IDX_GRP_LL_SRS_BSTRCT_ACTOR_ID	14	42	00:00:00.01
* 22	INDEX UNIQUE SCAN	PK_GROUP_	42	42	00:00:00.01
* 23	INDEX UNIQUE SCAN	PK_QUEUE_GROUPS	42	0	00:00:00.01

Fig. 5.33 Example 2. The first improvement

column PROCESSING_INFO_ID of the table QUEUE_ENTRY has to be indexed. I checked this and discovered that the index IQE_PROCINFO exists for this column. I then executed the SQL statement with the following two hints: LEADING(C0 A0) and INDEX(A0 IQE_POCINFO). The result can be seen in Fig. 5.33."

P.: "The runtime has improved from 0.39 to 0.07 seconds. However, I see that the optimizer has not taken the index IQE_PROCINFO. Instead of the desired nested loop join, it has used a hash join."

A.: "That's true. With the additional hint USE_NL(C0 A0), I managed to force the optimizer to do what I wanted it to do (Fig. 5.34)."

P.: "I am amazed once again. You have reduced the runtime to 0.02 seconds and improved the original runtime of 0.39 seconds approximately 20-fold."

A.: "Now let's take the last example in this section (see Fig. 5.35). The relevant SQL statement ran for 10,044 seconds. What do you think of the execution plan, Peter?"

P.: "Presumably that is an interim result because the arrows in column 'Id' indicate active execution plan steps. The relevant SQL statement was not yet finished when the SQL monitoring report was created."

A.: "You have interpreted that correctly. However, the runtime statistics in this report are representative and adequate for the analysis."

P.: "With the best will in the world, I don't think that 15,069 rows are a small number of hits."

Plan hash value: 2113343020

Id	Operation	Name	Starts	A-Rows	A-Time
1	SORT ORDER BY		1	0	00:00:00.02
* 2	FILTER		1	0	00:00:00.02
3	NESTED LOOPS OUTER		1	841	00:00:00.02
4	NESTED LOOPS		1	841	00:00:00.02
5	TABLE ACCESS BY INDEX ROWID	PROCESSING_INFO	1	841	00:00:00.01
* 6	INDEX RANGE SCAN	IPROC_INFO_PS	1	841	00:00:00.01
* 7	TABLE ACCESS BY INDEX ROWID	QUEUE_ENTRY	841	841	00:00:00.01
* 8	INDEX RANGE SCAN	IQE_PROCINFO	841	841	00:00:00.01
* 9	INDEX UNIQUE SCAN	PK_QUEUE	841	841	00:00:00.01
10	NESTED LOOPS		14	0	00:00:00.01
* 11	INDEX UNIQUE SCAN	PK_QUEUE_RECIPIENTS	14	0	00:00:00.01
* 12	INDEX UNIQUE SCAN	PK_USER_	0	0	00:00:00.01
13	NESTED LOOPS		14	0	00:00:00.01
* 14	INDEX UNIQUE SCAN	PK_QUEUE_RECIPIENTS_EXTERNAL	14	0	00:00:00.01
* 15	INDEX UNIQUE SCAN	PK_USER_	0	0	00:00:00.01
16	NESTED LOOPS		14	0	00:00:00.01
17	NESTED LOOPS		14	42	00:00:00.01
18	NESTED LOOPS		14	42	00:00:00.01
* 19	INDEX UNIQUE SCAN	PK_USER_	14	14	00:00:00.01
20	TABLE ACCESS BY INDEX ROWID	GROUP_ALL_USERS	14	42	00:00:00.01
* 21	INDEX RANGE SCAN	IDX_GRP_LL_SRS_BSTRCT_ACTOR_ID	14	42	00:00:00.01
* 22	INDEX UNIQUE SCAN	PK_GROUP_	42	42	00:00:00.01
* 23	INDEX UNIQUE SCAN	PK_QUEUE_GROUPS	42	0	00:00:00.01

Fig. 5.34 Example 2. A 20-fold improvement in runtime

SQL Plan Monitoring Details (Plan Hash Value=3720417339)

Id	Operation	Name	Execs	Rows (Actual)	Activity (%)
0	SELECT STATEMENT		1	15069	
1	FILTER		1	15069	
2	NESTED LOOPS		1	15069	
3	NESTED LOOPS		1	15139	
-> 4	NESTED LOOPS		1	15504	
-> 5	NESTED LOOPS		1	1M	0.01
-> 6	PARTITION RANGE ITERATOR		1	1M	
-> 7	TABLE ACCESS BY LOCAL INDEX ROWID	BCA_CN_LINK	74	1M	64.48
-> 8	INDEX RANGE SCAN	BCA_CN_LINK~S01	74	780M	25.20
-> 9	PARTITION RANGE ITERATOR		1M	1M	0.05
-> 10	TABLE ACCESS BY LOCAL INDEX ROWID	BCA_CNSP_ACCT	1M	1M	0.97
-> 11	INDEX RANGE SCAN	BCA_CNSP_ACCT~0	1M	1M	0.09
12	PARTITION RANGE ITERATOR		1M	15504	0.04
-> 13	TABLE ACCESS BY LOCAL INDEX ROWID	BCA_CONTRACT	1M	15504	6.41
-> 14	INDEX RANGE SCAN	BCA_CONTRACT~0	1M	4M	2.73
15	INDEX UNIQUE SCAN	BCA_PAYREF~0	29135	15139	0.02
-> 16	TABLE ACCESS BY INDEX ROWID	BCA_PAYREF	24056	15069	

```
Predicate Information (identified by operation id) :
---------------------------------------------------
    1 - filter(:A6>=:A5)
    7 - filter(("T_02"."FUNCTION"=:A9 AND "T_02"."OBJECT_TYP"=:A8))
    8 - access("T_02"."CLIENT"=:A1 AND "T_02"."VALID_TO_REAL"=:A7)
    8 - filter("T_02"."VALID_TO_REAL"=:A7)
   10 - filter("T_01"."VALID_TO_REAL"=:A7)
   11 - access("T_01"."CLIENT"=:A0 AND "T_02"."CONTRACT_INT"="T_01"."CONTRACT_INT")
   13 - filter(("T_00"."VALID_TO_REAL"=:A7 AND "T_00"."PRODINT"=:A4 AND "T_00"."STATUS">=:A5 AND
"T_00"."STATUS"<=:A6))
   14 - access("T_00"."CLIENT"=:A3 AND "T_01"."CONTRACT_INT"="T_00"."CONTRACT_INT")
   15 - access("T_03"."CLIENT"=:A2 AND "T_03"."PAYREF_INT"="T_02"."OBJECT_ID")
```

Fig. 5.35 The third example of an inappropriate table order in a join with a low number of hits

A.: "Compared to 780M in step 8, this number of hits is relatively small. Now I really think it's time you started your analysis."

Id	Operation	Name	Starts	A-Rows	A-Time
0	SELECT STATEMENT		1	24731	00:04:53.90
* 1	FILTER		1	24731	00:04:53.90
2	NESTED LOOPS		1	24731	00:04:53.88
3	NESTED LOOPS		1	24731	00:04:17.84
4	NESTED LOOPS		1	25255	00:03:38.67
5	NESTED LOOPS		1	27884	00:01:00.26
6	PARTITION RANGE ITERATOR		1	27884	00:00:50.95
* 7	TABLE ACCESS FULL	BCA_CONTRACT	120	27884	00:00:50.93
8	PARTITION RANGE ITERATOR		27884	27884	00:00:09.27
* 9	TABLE ACCESS BY LOCAL INDEX ROWID	BCA_CNSP_ACCT	27884	27884	00:00:09.14
* 10	INDEX RANGE SCAN	BCA_CNSP_ACCT~0	27884	27884	00:00:00.45
11	PARTITION RANGE ITERATOR		27884	25255	00:02:38.38
* 12	TABLE ACCESS BY LOCAL INDEX ROWID	BCA_CN_LINK	27884	25255	00:02:38.28
* 13	INDEX RANGE SCAN	BCA_CN_LINK~S01	27884	625K	00:01:07.13
* 14	INDEX UNIQUE SCAN	BCA_PAYREF~0	25255	24731	00:00:39.13
15	TABLE ACCESS BY INDEX ROWID	BCA_PAYREF	24731	24731	00:00:36.01

Fig. 5.36 Example 3. Improvement after moving the two tables in the join

P.: "The high cardinality of 780M occurs in step 8 with an index access. In the next step, 7, it falls to 1M with the table access by rowid for the table BCA_CN_LINK (alias T_02). The nested loop join in step 5 has the same cardinality. This table is table A in this plan. In step 13, the cardinality falls to 15504 with the join to table BCA_CONTRACT (alias T_00). That is table B. However, I cannot place table BCA_CONTRACT before table BCA_CN_LINK in the table order, without causing a Cartesian product, because they don't have any join conditions."

A.: "That's no problem. The tables in this join are joined together as follows: T_00 ⇒ T_01 ⇒ T_02 ⇒ T_03. This is a table chain. The table order in the join is T_02, T_01, T_00, T_03. Previously, we saw that it may not only be necessary to move just one table B, but several tables. We have precisely such a case here. Which table do we also have to move?"

P.: "Table T_01, which is joined to T_02."

A.: "Exactly. With the hint LEADING(T_00 T_01 T_02 T_03), I defined the desired table order and executed the SQL statement again (Fig. 5.36)."

P.: "Why did you mark three lines in red in the execution plan in Fig. 5.36?"

A.: "I wanted to point out that this execution plan has further potential for improvement. In step 7, one could try using an index range scan instead of an FTS. Steps 12 and 13 show that the selectivity of the index BCA_CN_LINK~S01 can be improved if one extends this index with the columns from the predicates to the table access by rowid. I didn't do any further tuning because the runtime of approximately 5 minutes was perfectly acceptable."

5.2.3 Joins with a Large Hit Quantity

Author: "In this section we will discuss joins with a large hit quantity. The relevant SQL statement has to process a large amount of data, which requires a certain amount of work. One can, therefore, not expect that

such a SQL statement will run as fast as lightning after tuning. Let's begin again with an inappropriate table order. How would you recognize this in the execution plan of a join with a large hit quantity, Peter?"

Peter: "Some joins with especially high cardinalities can indicate this."

A.: "You have joins with high as well as relatively low cardinality in the execution plan?"

P.: "Exactly."

A.: "In the section 'Joins with a Low Number of Hits,' we've already seen what an adverse effect a high cardinality of some joins can have on the performance of a join of several tables, and this was mainly using nested loop joins. Could you tell me how severely this can affect the hash joins?"

P.: "As the build table of a hash join is built in the memory, if possible it has to fit into the memory. If it is larger, it complicates processing severely and increases the runtime of the hash join. In a hash join of several tables, the hits of a hash join are, at the same time, the build table of the next hash join. For this reason, I would smooth out the existing cardinality peaks."

A.: "How would you do that?"

P.: "Can't one use the same heuristic method that we discussed in the section 'Joins with a Low Number of Hits'?"

A.: "Yes, you can. We have already shown enough examples of how to work with this method. Assuming we have used this method and have no more serious peaks, can we now be sure that the table order in the join is more or less correct?"

P.: "Difficult to say."

A.: "I don't have any general suggestions in connection with formal SQL tuning either. In any case, it's worth examining the execution plan again in detail. Our first example shows how one can recognize an inappropriate table order in a join without cardinality peaks."

P.: "When we have eliminated all 'brakes' in the execution plan, what else can we do to improve performance if it is necessary to improve it further?"

A.: "Oracle offers a range of methods for this purpose: Materialized views, star schemas, etc. If we don't want to intervene so deeply in the existing database processes and SQL statements, we can very successfully use parallelization (parallel query) for the problematical SQL statements. One can leave the decision regarding parallelization completely to Oracle (using the feature 'Automatic Degree of Parallelism'), or one can define the degree of parallelism for the entire SQL statement oneself. In this case, Oracle decides which objects are to be parallelized in which execution plan steps. If one sees from the runtime statistics in the execution plan that some steps are particularly expensive, then it makes sense to parallelize precisely these steps manually with the relevant hints. One can use parallelization both after and before formal SQL tuning (if the relevant performance problem is so acute that one needs a solution immediately and the SQL tuning can be performed

later). One has to be careful with parallelization because this feature can take up a lot of resources. However, as we do not wish to focus on parallelization in this book, we won't investigate this interesting topic any further."

P.: "Can you please show me how one can use parallelization in connection with the runtime statistics in the execution plan."

A.: "I'll show you with the second example in this section. We'll start with the first example. An execution plan of a join with a large hit quantity is shown in Fig. 5.37. One can see that the cardinality in step 19 rises to 4159K and increases the cardinality of the nested loop join in step 12 accordingly. This cardinality remains unchanged in outer joins in steps 10 and 11."

Peter: "Is that an example from the section 'An Index with a Large Clustering Factor'?"

A.: "What a mammoth memory you've got! Yes, I did actually use a part from this execution plan."

P.: "Doesn't this example come from the section 'Joins with a Low Hit Quantity'? The number of hits of the join is small, only 28 rows."

A.: "Your question proves that it wasn't a compliment after all when I said you had a mammoth memory. Mammoths didn't become extinct without reason, did they? Do you remember that cardinality mustn't fall as a result of an aggregation? But in step 9 it does fall after an aggregation, so we have to consider the cardinality before this operation. That is 4159K. In this execution plan, we don't see any cardinality peaks with individual joins."

P.: "The cardinality of 4159K initially occurs with access to the table PICKAUF. This table isn't the last in the join. It is followed by the tables QUANTEN and PRUEFGRUENDE. These two tables do not have any join predicates for the table PICKAUF, however."

A.: "That's true. They are joined to the table PICKRUND (alias R) with an outer join. As they come after the table PICKAUF in the join, the cardinality of the relevant joins must be at least as high as the cardinality of the table PICKAUF, i.e., 4159K."

P.: "We can, however, place the tables QUANTEN and PRUEFGRUENDE before the table PICKAUF in the join."

A.: "Correct. This shows us that the table order in this join is not quite optimal after all. One could have changed this table order as you have just suggested. This change in the table order would have improved the runtime by approximately 45 seconds at best (30.05 + 15.40 ~ 45). I noticed that the table access by rowid for the table PICKAUF took approx. 8 minutes. This table access was so expensive because the index PI_PR_FK_I had a large clustering factor. Initially, I wanted to try to do without table access by rowid. To achieve this, I had to extend the index PI_PR_FK_I by a few columns. Peter, can you remember how to find these columns?"

Plan hash value: 2555052906

Id	Operation	Name	Starts	A-Rows	A-Time	Buffers
0	SELECT STATEMENT		1	28	00:09:22.19	5161K
1	NESTED LOOPS		1	1	00:00:00.01	0
2	FIXED TABLE FULL	X$KCCDI2	1	1	00:00:00.01	0
* 3	FIXED TABLE FULL	X$KCCDI	1	1	00:00:00.01	0
4	MERGE JOIN		1	28	00:09:22.19	5161K
5	SORT JOIN		1	28	00:09:22.01	5161K
6	VIEW	V_CRA_002_STATUS_KO_EINH	1	28	00:09:22.01	5161K
7	SORT GROUP BY		1	28	00:09:22.01	5161K
8	VIEW		1	96	00:09:22.01	5161K
9	SORT GROUP BY		1	96	00:09:21.91	5161K
10	NESTED LOOPS OUTER		1	4159K	00:09:07.70	5161K
11	NESTED LOOPS OUTER		1	4159K	00:08:48.53	4696K
12	NESTED LOOPS		1	4159K	00:08:13.51	4278K
13	NESTED LOOPS		1	137K	00:00:13.45	25696
* 14	TABLE ACCESS BY INDEX ROWID	PICKLISTEN	1	122K	00:00:05.19	7210
* 15	INDEX RANGE SCAN	PIL_PK	1	122K	00:00:00.39	311
16	TABLE ACCESS BY INDEX ROWID	PICKRUND	122K	137K	00:00:08.05	18486
* 17	INDEX RANGE SCAN	PR_PL_FK_I	122K	137K	00:00:01.77	2788
18	TABLE ACCESS BY INDEX ROWID	PICKAUF	137K	4159K	00:07:58.00	4252K
* 19	INDEX RANGE SCAN	PI_PR_FK_I	137K	4159K	00:00:06.51	162K
* 20	TABLE ACCESS BY INDEX ROWID	QUANTEN	4159K	832K	00:00:30.05	418K
* 21	INDEX RANGE SCAN	QT_LE1_FK_I	4159K	832K	00:00:10.86	405K
* 22	TABLE ACCESS BY INDEX ROWID	PRUEFGRUENDE	4159K	4277	00:00:15.40	465K
* 23	INDEX RANGE SCAN	PG_NR_LE_I	4159K	359K	00:00:07.91	396K
* 24	SORT JOIN		28	28	00:00:00.18	119
25	VIEW	V_CRA_002_STATUS_KO_EINH_NLS	1	1	00:00:00.18	119
26	FAST DUAL		1	1	00:00:00.01	0

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
  …
  16 - SEL$10        / R@SEL$10
  17 - SEL$10        / R@SEL$10
  18 - SEL$10        / P@SEL$10
  19 - SEL$10        / P@SEL$10
  20 - SEL$10        / Q@SEL$10
  21 - SEL$10        / Q@SEL$10
  22 - SEL$10        / PG@SEL$10
  23 - SEL$10        / PG@SEL$10
  …
Predicate Information (identified by operation id):
---------------------------------------------------
  …
  17 - access("R"."LAGER_PICKL"="PL"."LAGER" AND "R"."NR_PICKL"="PL"."NR_PICKL")
  19 - access("P"."LAGER_RUNDF"="R"."LAGER" AND "P"."NR_RUNDF"="R"."NR_RUNDF")
  20 - filter(("Q"."ID_ARTIKEL" NOT LIKE 'VST%' AND "Q"."ID_ARTIKEL"<>'LEER'))
  21 - access("Q"."LAGER_NR_LE_1"="R"."LAGER_PACK" AND "Q"."NR_LE_1"="R"."NR_LE_PACK")
  22 - filter("PG"."STAT"='10')
  23 - access("PG"."NR_LE"="R"."NR_LE_PACK")
  …
Column Projection Information (identified by operation id):
-----------------------------------------------------------
  …
  18 - "P"."STAT"[VARCHAR2,2], "P"."MNG_SOLL"[NUMBER,22]
  …
```

Fig. 5.37 A join with a large hit quantity and an index with a large clustering factor

Plan hash value: 355660978

Id	Operation	Name	Starts	A-Rows	A-Time
0	SELECT STATEMENT		1	28	00:00:15.32
1	NESTED LOOPS		1	1	00:00:00.01
2	FIXED TABLE FULL	X$KCCDI2	1	1	00:00:00.01
* 3	FIXED TABLE FULL	X$KCCDI	1	1	00:00:00.01
4	MERGE JOIN		1	28	00:00:15.32
5	SORT JOIN		1	28	00:00:15.32
6	VIEW	V_CRA_002_STATUS_KO_EINH	1	28	00:00:15.32
7	SORT GROUP BY		1	28	00:00:15.32
8	VIEW		1	96	00:00:15.32
9	SORT GROUP BY		1	96	00:00:15.31
10	NESTED LOOPS		1	4159K	00:00:06.80
11	NESTED LOOPS OUTER		1	137K	00:00:02.95
12	NESTED LOOPS OUTER		1	137K	00:00:02.13
13	NESTED LOOPS		1	137K	00:00:01.17
* 14	TABLE ACCESS BY INDEX ROWID	PICKLISTEN	1	122K	00:00:00.27
* 15	INDEX RANGE SCAN	PIL_PK	1	122K	00:00:00.06
16	TABLE ACCESS BY INDEX ROWID	PICKRUND	122K	137K	00:00:00.77
* 17	INDEX RANGE SCAN	PR_PL_FK_I	122K	137K	00:00:00.43
* 18	TABLE ACCESS BY INDEX ROWID	QUANTEN	137K	13350	00:00:00.79
* 19	INDEX RANGE SCAN	QT_LE1_FK_I	137K	13362	00:00:00.57
* 20	TABLE ACCESS BY INDEX ROWID	PRUEFGRUENDE	137K	102	00:00:00.69
* 21	INDEX RANGE SCAN	PG_NR_LE_I	137K	14055	00:00:00.50
* 22	INDEX RANGE SCAN	PI_PR_FK_LR_I	137K	4159K	00:00:02.32
* 23	SORT JOIN		28	28	00:00:00.01
24	VIEW	V_CRA_002_STATUS_KO_EINH_NLS	1	1	00:00:00.01
25	FAST DUAL		1	1	00:00:00.01

Fig. 5.38 "Index-Only" access optimizes the table order in the join

P.: "I would look for these columns in predicates and in projections belonging to step 18. As there are no predicates to this step, we only have projections. There we can find two columns, STAT and MNG_SOLL."

A.: "As the index PI_PR_FK_I only had two columns, it was possible to add two columns without the index becoming too wide. For test purposes, a new index, PI_PR_FK_LR_I, was created with these four columns parallel to index PI_PR_FK_I and the SQL statement executed again (Fig. 5.38)."

P.: "The runtime is now 15.32 seconds. That really is an improvement! I notice, however, that the table order has been changed by the optimizer: The table PICKAUF is now in the last position, where it belongs in my opinion. Do you have an explanation for this optimization of the table order?"

A.: "Clustering factor has a strong weighting as far as optimizer costs are concerned. The appropriate formulae can be found in [2]. This seriously affects the table order. We have eliminated the clustering factor in the optimizer costs for an index because we have dispensed with table access by rowid (precisely in this operation, the optimizer takes the clustering factor into account as far as costs are concerned). Consequently, the optimizer has changed the table order."

P.: "Was it really worthwhile analyzing whether the table order in the join was optimal? The change in this order only reduced the runtime by 45 seconds."

A.: "Yes, that really isn't a very large proportion of the runtime, which was originally over 9 minutes. However, it could have been different if the table order had remained unchanged after the index extension. Now let's

SQL Plan Monitoring Details (Plan Hash Value=1597604708)

Id	Operation	Name	Time Active(s)	Execs	Rows (Actual)
0	SELECT STATEMENT		574	1	559K
1	NESTED LOOPS OUTER		574	1	559K
2	VIEW	V_M_F_SUBTR_AGG_DETAIL_2	574	1	559K
3	WINDOW SORT		715	1	559K
4	WINDOW SORT		294	1	559K
5	WINDOW SORT		250	1	559K
6	WINDOW SORT		252	1	559K
7	HASH JOIN		2744	1	559K
8	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_SERVICETIME	1	1	413
9	HASH JOIN		350	1	559K
10	HASH JOIN		2723	1	559K
11	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_COUNTRY	1	1	1
12	HASH JOIN		2723	1	559K
13	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_COUNTRY_REGION	1	1	10
14	HASH JOIN		2715	1	559K
15	VIEW		1	1	508
16	HASH GROUP BY		1	1	508
17	NESTED LOOPS		1	1	110K
18	NESTED LOOPS		1	1	110K
19	VIEW		1	1	508
20	HASH GROUP BY		9	1	508
21	HASH JOIN OUTER		9	1	9M
22	INDEX STORAGE FAST FULL SCAN	IX_TR2AG_ROOTUUID_STATUS	1	1	508
23	PARTITION RANGE ALL		9	1	14M
24	PARTITION RANGE SINGLE		9	97	14M
25	TABLE ACCESS STORAGE FULL	M_F_SUBTRANSACTION_DETAIL	12	97	14M
26	PARTITION RANGE ALL		1	898	110K
27	INDEX RANGE SCAN	IX_FACT_LAST_DML_DATE	1	10550	110K
28	TABLE ACCESS BY LOCAL INDEX ROWID	I_ITCAMTM_I_SUBTRANS_FACT	2	111K	110K
29	HASH JOIN		2715	1	350M
30	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_SUBTRANSACTION	1	1	1911
31	HASH JOIN		2715	1	3G
32	HASH JOIN		1	1	6877
33	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_LOCATIONS	1	1	173
34	HASH JOIN		1	1	6877
35	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_AGENTS	1	1	243
36	HASH JOIN		1	1	6877
37	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_TRANSACTION2AGENT	1	1	832
38	HASH JOIN		1	1	441
39	HASH JOIN		1	1	59
40	HASH JOIN		1	1	42
41	NESTED LOOPS		1	1	13
42	NESTED LOOPS		1	1	13
43	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_CUSTOMER	1	1	1
44	INDEX RANGE SCAN	ITCAMTM_CUST2SUBCUST_FK	1	1	13
45	TABLE ACCESS BY INDEX ROWID	I_ITCAMTM_I_SUBCUSTOMER	1	13	13
46	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_APPLICATION	1	1	42
47	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_TRANSACTION	1	1	59
48	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_SUBTRANS_GROUP	1	1	441
49	PARTITION RANGE ALL		2715	1	351M
50	TABLE ACCESS STORAGE FULL	I_ITCAMTM_I_SUBTRANS_FACT	2715	20	351M
51	TABLE ACCESS STORAGE FULL	B_DATEDIMENSION	22	1	69762
52	TABLE ACCESS BY GLOBAL INDEX ROWID	M_F_SUBTRANSACTION_DETAIL	574	559K	507K
53	INDEX UNIQUE SCAN	M_F_SUBTRANS_DEATIL_PK	574	559K	507K

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
…
  50 - SEL$C8360722 / FACT@SEL$4
…
```

Fig. 5.39 Parallelization based on the runtime statistics in the execution plan

discuss the second example in Fig. 5.39. The runtime of the relevant SQL statement was 3837 seconds and had to be improved urgently. What can you recognize in this execution plan, Peter?"

P.: "Step 50 is problematical. 351M rows are found there. This causes an expensive hash join with a cardinality of 3G in step 31. Initially, this cardinality falls to 350M in step 29 and then to 559K with the next join in step 14. I think the table order in this join is suboptimal."

A.: "Correct. It was possible to tune this plan. However, as a quick solution was required and there wasn't a heavy load on the system, I decided to parallelize the problematical step 50. For this purpose, I used the hint PARALLEL(@SEL$C8360722 FACT@SEL$4 8), which reduced the runtime to 315 seconds."

5.3 Summary

Table 5.1 Problem categories: identification and solution

Global/ local	Problem category	Identification in execution plan	Solution
Local	FTS due to a missing index	FTS with a large number of buffer gets or disk reads and a low cardinality	Creation of the relevant index
Local	FTS on a sparse table	FTS with a large number of buffer gets or disk reads and a low cardinality	Table reorganization. As a temporary solution, the creation of the relevant index is possible
Local	Index scan with a nonselective index	An index scan with a high cardinality which falls in the following step with table access by rowid	Extend the index by the addition of selective columns from the filter from table access by rowid
Local	Index scan with an index with a large clustering factor	The cardinality of an index scan is comparable with the number of buffer gets in the following table access by rowid	Check the optimizer statistics "clustering factor" and, if the figure really is large, try the following: – Extend the index with the columns from the filter and from the projection belonging to table access by rowid to render this table access superfluous, or – Enter the data into the relevant table in the same order as in the index
Local	Index scan with a sparse index or a selective filter in the index scan	The number of buffer gets resulting from the index scan is large and the cardinality relatively low. If there is no filter in the index scan, this is most likely a sparse index. If	Rebuild the sparse index. If a selective filter belongs to the index scan, one must clarify the reasons why selective predicates land in the filter instead of in the access. If

(continued)

		a filter belongs to the index scan, this can be either a sparse index or a selective filter. One then has to check both possibilities	necessary, create an index for the selective columns from the filter
Local	Nested loop join instead of hash join in an equi-join	The cardinality of the outer or inner loop is high	Use hash join instead of nested loop join in the execution plan. For this purpose, one can use the hint USE_HASH
Local	Hash join instead of nested loop join in an equi-join	Both the cardinality of access to the build table and the hit quantity of the hash join are low	Use nested loop join instead of hash join in the execution plan. For this purpose, one can use the hint USE_NL (normally in combination with the hint LEADING)
Global	Inappropriate table order in the join with a small number of hits	Some table accesses and joins with which the relevant tables are involved have a high cardinality	Good chances of achieving a significant improvement in performance. Try to change the table order in such a way that the cardinality is low in every execution plan step. The following heuristic method often helps here: – Find the table A in the execution plan which, when accessed, has a high cardinality. The cardinality of the relevant join must also be high – Find the table B in the subsequent course of the execution plan for which the cardinality falls when it is joined – Try inserting table B before table A in the table order When changing the table order, take care to ensure that no Cartesian products occur
Global	Inappropriate table order in the join with a large hit quantity	High cardinality peaks in some joins in the join	Try to eliminate the cardinality peaks with the heuristic method described above. If the runtime is still not acceptable after this, parallelize the relevant SQL statement

Procedure of Formal SQL Tuning

6

In this chapter we will discuss what is, in our view, a sensible formal SQL tuning process. Some steps of this process can facilitate or even eliminate the need for tuning. We also feel it is important to clarify which problems have a higher priority during tuning if several problems occur in an execution plan at the same time.

Author: "Peter, where would you begin with SQL tuning?"

Peter: "That's a very good question. After so much new information, I am a bit confused."

A.: "Then we'll clarify the matter together and organize this information systematically. Before I begin with SQL tuning, I always check whether SQL tuning is really necessary."

P.: "How do you do that?"

A.: "It may be possible to find a good execution plan for the relevant SQL statement, either in the AWR or in the SQL area. If this is the case, one can fix this plan (the methods for this can be found in [1]). If, however, one has to improve a bad execution plan, then I would hesitate a moment."

P.: "And wait for a miracle?"

A.: "I appreciate your humor, Peter. It makes sense to check if the relevant SQL statement may be badly programmed."

P.: "So you suggest analyzing the SQL statement. But that's exactly what I try to avoid doing because I'm mostly not able to do it."

A.: "Some other authors recommend familiarizing oneself with the SQL statement first and getting to know its structure. I don't go that far because I don't think it's necessary. On the other hand, it is wrong to approach tuning with your eyes closed. Some serious errors are very conspicuous. It is better to correct these with a developer instead of tuning the SQL statement. With the next example, I will show you that this is not at all difficult in some cases. The following SQL statement

© Springer-Verlag Berlin Heidelberg 2016

L. Nossov et al., *Formal SQL Tuning for Oracle Databases*,
DOI 10.1007/978-3-662-50417-8_6

SQL Plan Monitoring Details (Plan Hash Value=2648737676)

Id	Operation	Name	Time Active(s)	Execs	Rows (Actual)
0	INSERT STATEMENT			1	
1	LOAD TABLE CONVENTIONAL			1	
2	HASH UNIQUE		1514	1	0
3	HASH JOIN		1528	1	42296
4	TABLE ACCESS FULL	CO_PRM	1	1	4727
5	HASH JOIN		1528	1	79305
6	INDEX FULL SCAN	TE_PRM_PRM_PRDV_RULE_PK	1	1	4727
7	HASH JOIN		1614	1	79305
8	INDEX RANGE SCAN	TE_001_PROM_INTERVAL_PK001	1	1	25225
9	HASH JOIN		1614	1	12M
10	TABLE ACCESS FULL	TE_ITEM_CO_MRHRC_GP	15	1	145K
11	HASH JOIN		456	1	7M
12	TABLE ACCESS FULL	CO_EL_MRST_PRDV	1	1	4448
13	HASH JOIN		456	1	18M
14	TABLE ACCESS FULL	TE_PROMOTION_PRDV_RULE	1	1	4718
15	HASH JOIN		456	1	18M
16	TABLE ACCESS FULL	RU_PRDV	1	1	4244
17	MERGE JOIN CARTESIAN		456	1	19M
18	TABLE ACCESS FULL	CO_EL_PRDV	456	1	5287
19	BUFFER SORT		456	5287	19M
20	TABLE ACCESS FULL	RU_PRDV_ITM	1	1	3671

Fig. 6.1 A poor execution plan with MERGE JOIN CARTESIAN

was running relatively well. From time to time, however, it changed to a suboptimal execution plan, as shown in Fig. 6.1."

P.: "In step 17, a Cartesian product is created with a cardinality of 19M. This is probably the reason for the poor performance. What did you do with this Cartesian product?"

A.: "Nothing at all. I noticed that the good plan also had a Cartesian product in it. Table CO_EL_PRDV was also involved in the relevant join. For this reason, I assumed that this table had no join predicates. I verified this directly in the SQL text. You can try that too, Peter."

```
/* B2230 */ insert into TE_001_PROM_INTERVAL (TYPECODE, ITEM_ID, CO_PRM__ID_PRM,
MERCHANDISE_STRUCTURE_ID, CO_PRM__DC_PRM_EF, CO_PRM__DC_PRM_EP, CO_PRM__DESCRIPTION,
CO_PRM__NM_PRM_PRT, CO_PRM__ORIGIN, CO_PRM__PROMOTION_TYPE, CO_PRM__PROMOTION_TYPE_NAME,
CO_PRM__EXTERNAL_PROMOTION_ID, RU_PRDV__DE_RU_PRDV, RU_PRDV__EXTERNAL_PRDV_RULE_ID,
RU_PRDV__ROUNDING_METHOD_CODE, RU_PRDV__DECIMAL_PLACES_COUNT, RU_PRDV__ROUND_DESTINATION_VAL,
PROMOTION_PRDV_RULE__I_D_C_V, PROMOTION_PRDV_RULE__R_PR_NM,PROMOTION_PRDV_RULE__TYPECODE,
THRESHOLD_QUANTITY, INTERVAL_QUANTITY, LIMIT_QUANTITY, REDUCTION_METHOD_CODE, REDUCTION_AMOUNT,
REDUCTION_PERCENT, NEW_PRICE_AMOUNT, EFFECTIVE_DATE_TIME, EXPIRATION_DATE_TIME,
ITEM_IS_SET_HEADER)
select distinct :1 , TE_ITEM_CO_MRHRC_GP.ITEM_ID, CO_PRM.ID_PRM, :2 , CO_PRM.DC_PRM_EF,
CO_PRM.DC_PRM_EP, CO_PRM.DESCRIPTION, CO_PRM.NM_PRM_PRT, CO_PRM.ORIGIN, CO_PRM.PROMOTION_TYPE,
CO_PRM.PROMOTION_TYPE_NAME, CO_PRM.EXTERNAL_PROMOTION_ID, RU_PRDV.DE_RU_PRDV,
RU_PRDV.EXTERNAL_PRDV_RULE_ID, RU_PRDV.ROUNDING_METHOD_CODE, RU_PRDV.DECIMAL_PLACES_COUNT,
RU_PRDV.ROUND_DESTINATION_VALUE, TE_PROMOTION_PRDV_RULE.ITEM_DISCOUNT_CONTROL_VECTOR,
TE_PROMOTION_PRDV_RULE.RECEIPT_PRINTER_NAME, TE_PROMOTION_PRDV_RULE.TYPECODE,
CO_EL_MRST_PRDV.QU_TH, CO_EL_MRST_PRDV.QU_INTV, CO_EL_MRST_PRDV.QU_LM, RU_PRDV_ITM.CD_RDN_MTH,
RU_PRDV_ITM.MO_RDN, RU_PRDV_ITM.PE_RDN, RU_PRDV_ITM.MO_PRC,
TE_ITEM_CO_MRHRC_GP.EFFECTIVE_DATE_TIME, TE_ITEM_CO_MRHRC_GP.EXPIRATION_DATE_TIME, :3
from RU_PRDV, TE_PROMOTION_PRDV_RULE, RU_PRDV_ITM, CO_PRM, TE_PRM_PRM_PRDV_RULE,
TE_001_PROM_INTERVAL templateTable, CO_EL_PRDV, CO_EL_MRST_PRDV, TE_ITEM_CO_MRHRC_GP
where RU_PRDV.ID_RU_PRDV = TE_PROMOTION_PRDV_RULE.PRDV_RULE_ID and RU_PRDV_ITM.ID_RU_PRDV =
RU_PRDV.ID_RU_PRDV and RU_PRDV.ID_RU_PRDV = TE_PRM_PRM_PRDV_RULE.PRDV_RULE_ID and
TE_PRM_PRM_PRDV_RULE.PROMOTION_ID = CO_PRM.ID_PRM and TE_PROMOTION_PRDV_RULE.SALE_RETURN_TYPECODE
<> :4 and TE_PROMOTION_PRDV_RULE.AMENDMENT_TYPECODE <> :5 and
RU_PRDV.LU_CBRK_PRDV_TRN in (:6 , :7 ) and ( RU_PRDV.SC_RU_PRDV is null or RU_PRDV.SC_RU_PRDV =
:8 ) and RU_PRDV.TY_RU_PRDV = :9 and RU_PRDV.BONUSPOINTS_FLAG = :10 and
RU_PRDV.EXTERNAL_PRDV_RULE_ID in ('1', '2', '8') and ( RU_PRDV.REBATE_METHOD_CODE is null or
RU_PRDV.REBATE_METHOD_CODE = :11 ) and RU_PRDV_ITM.CD_RDN_MTH in (:12 , :13 , :14 ) and
templateTable.TYPECODE = :15 and ( templateTable.CO_PRM__ID_PRM = :16 ) and
CO_EL_MRST_PRDV.ID_EL_PRDV = TE_PROMOTION_PRDV_RULE.PRDV_RULE_EL_ID and (
( CO_EL_MRST_PRDV.TY_TH in (:17 , :18 ) and CO_EL_MRST_PRDV.QU_TH <= 1) or (
CO_EL_MRST_PRDV.TY_TH in (:19 ) and CO_EL_MRST_PRDV.MO_TH = 0.01)) and CO_EL_PRDV.TY_EL_PRDV =
:20 and (CO_EL_MRST_PRDV.ID_MRHRC_GP like :21 ) and (
TE_ITEM_CO_MRHRC_GP.MERCHANDISE_HIERARCHY_GROUP_ID = CO_EL_MRST_PRDV.ID_MRHRC_GP and (
templateTable.ITEM_ID <> :22 and TE_ITEM_CO_MRHRC_GP.ITEM_ID = templateTable.ITEM_ID )) and (
TE_ITEM_CO_MRHRC_GP.EXPIRATION_DATE_TIME is null or TE_ITEM_CO_MRHRC_GP.EXPIRATION_DATE_TIME
>= :23 ) and ( TE_ITEM_CO_MRHRC_GP.STATUS_CODE is null or TE_ITEM_CO_MRHRC_GP.STATUS_CODE = :24 )
```

P.: “This table only has one predicate CO_EL_PRDV.TY_EL_PRDV = :20. It is not actually joined to any other table. Nor do I notice any columns from this table in the select list. Its only function in the join is to multiply the result. With the operator DISTINCT, these multiplied rows are then removed again. That really is a strange SQL statement!”

A.: “That’s clearly a badly programmed SQL statement. After removal of the table CO_EL_PRDV from the join, there were no further problems with this SQL statement. Did you find this analysis complicated?”

P.: “I must admit it was very easy. But I was wondering why you didn’t fix the good plan.”

A.: “It wasn’t possible. You probably noticed the comment at the beginning of the SQL text. That is the name of the relevant database schema. As there were hundreds of identically designed database schemas in this database, there were, accordingly, a large number of problematical SQL statements which only differ in these comments. One would therefore have had to fix hundreds of execution plans of these SQL statements.”

P.: “I see.”

A.: “Now we’ll discuss

- how one can identify bottlenecks in an execution plan,
- which problems one should eliminate first, and which solutions are preferable.”

P.: "The bottlenecks are very easy to find. The execution plan step with the longest runtime is also a bottleneck, isn't it?"

A.: "Not always. For example, if a step with an FTS instead of an index access has the longest runtime, then this FTS is a bottleneck in the execution plan. Often a problem in a long runtime only becomes visible in the subsequent execution plan steps. We have already discussed this in the section "A Non-selective Index."

P.: "How should one proceed, then? Look for the highest cardinality in the execution plan?"

A.: "One can look for the step with the longest runtime, as you suggested earlier. But then you must try to find out what kind of problem it is. If the problem belongs to one of the following categories:

- FTS due to a missing index
- FTS on a sparse table
- Index scan with a sparse index
- A selective filter in an index scan

Then the step with the longest runtime is a bottleneck in the execution plan. A long runtime is a fairly precise indication of a problematical join from the category 'nested loop instead of hash join and vice-versa.' If a problem occurs regarding a high cardinality, this problem is not normally caused in the step with the longest runtime, but in one of the preceding steps where a large volume of data has to be processed (i.e., in a step with a high cardinality). These are problems from the following categories:

- Index scan with a nonselective index
- Index scan with an index with a large clustering factor
- Inappropriate table order in the join

One can also proceed differently: First of all, find the steps with a high cardinality and then identify and solve the associated problems. According to our statistics on the problem cases in the section 'Statistics on Problem Categories,' one can solve most problems in this way. After that, in any case, one must check the steps with long runtimes. This is the approach I prefer. 'Cardinality' is a magic word in SQL tuning (Fig. 6.2). It plays a decisive role when analyzing problems from the second group of categories. In conjunction with the other runtime statistics, it is also extremely important for the first group."

P.: "What should I do if I have several problems in an execution plan?"

A.: "If these problems occur at different points in the execution plan, you can solve them one after another. It is far more interesting when these problems occur at practically the same point. Then one must decide how to start."

P.: "I would start with the problem which poses the greatest threat to performance."

Fig. 6.2 If "Open Sesame!" doesn't work, try "Cardinality"

A.: "So you're acting on the basis of runtime again: The problem which causes the longest runtime is solved first. In many cases, this is the best way. In Fig. 5.37, for example, we had two problems: an inappropriate table order in the join and an index scan with an index with a large clustering factor and subsequent table access by rowid. The first caused a runtime of approx. 45 seconds and the second approx. 8 minutes. For this reason, I solved the second problem first, and this improved the table order at the same time. However, there are cases in which the decision is not so easy to make because it depends on a number of factors."

P.: "Have we already had an example of this?"

A.: "Of course. That's the example in Fig. 5.15. There you can see several problems, for example, an inefficient nested loop join. It was, however, difficult to say in advance what this nested loop join would cost exactly. Only after using hash join instead of nested loop join did it become clear that the nested loop join was fairly expensive (the runtime was reduced from 573 to approx. 30–50 seconds). In the second problem which is recognizable, there is a nonselective index in step 19. We have already discussed this problem in the section 'Nested Loop Join Instead of Hash Join and Vice-Versa.' Could you analyze this problem again please, Peter?"

P.: "The cardinality in the case of index scan with the index EM_METRIC_ITEMS_KEY_IDX is 584M in step 19. It falls to 8M in the next step, step 18, with table access by rowid. This points to a selective filter in step 18. If one extends the index EM_METRIC_ITEMS_KEY_IDX by the

addition of selective columns from this filter, the cardinality is already reduced in step 19 and the runtime is improved."

A.: "Such a change is not allowed in one of Oracle's schemas (SYSMAN). But let us assume that this is not one of Oracle's schemas and that the relevant index extension is legitimate. It is, again, difficult to say how much this would improve performance (I would anticipate a runtime of 1–2 minutes with this solution). However, we have a third problem in the execution plan: an inappropriate table order in the join. After this problem was solved, the runtime fell to 0.59 seconds. When analyzing the problem, one could already assume that this might be the fastest solution. A fall in cardinality from 8M in step 6 to 23 in step 5 looked very promising. Which of these three solutions is the best, Peter?"

P.: "The fastest, obviously."

A.: "That's not so obvious. If one requires a solution very quickly, one must concentrate on a solution which can be implemented as quickly as possible. The solution with a runtime of 1–2 minutes might have been perfectly adequate."

P.: "So what's wrong with the fastest solution?"

A.: "As usual with formal SQL tuning, in this case one needs a hint for the relevant change in the execution plan. One can explicitly add this hint to the SQL text, but this involves a program code change. It is also possible to add this hint implicitly as a hidden hint. For this, a database administrator uses either stored outlines (before Oracle 12c), SQL profile, SQL patch, or SQL plan baselines (see [1]). There are companies where you need authorization for this. This is precisely the situation that Hanno Ernst mentions in the section 'Hanno's Experience'. In such cases, he often uses an index extension. In the case of index extensions, the clustering factor of the index changes, and this can also change the table order in the join. In his systems, Hanno observes that such changes in table order are usually advantageous, and, for this reason, he prefers index extensions to other solutions where possible."

P.: "So you mean that the best solution is one that has to suit the situation in hand."

A.: "That's right."

7 Practical Experience with Formal SQL Tuning

7.1 Hanno's Experience

The "formal method" developed from an idea of how one can successfully help someone tune a very complex execution plan in a blog using pure text ping-pong.

"Formal method"—what does that actually mean? That's what I asked myself when I first heard of it. "Formal"—referring to form not content! If we draw parallels between the meaning of the word and the approach, then I would describe it as follows: The form defines the rules. Regardless of the data model or data content, we apply these rules and achieve our goal in a very efficient manner.

In order to be able to carry out tuning effectively, one must understand execution plans—to some extent at least. Chapter 3 is relatively detailed, as it should be.

As someone with personal practical experience, I can say that the use of runtime statistics opens up totally new possibilities. They show you what is really happening. One can also compare them with optimizer estimations. In the past, to find the causes of problems, we often carried out analyses by counting value distributions or uniqueness and comparing them with optimizer statistics. To some extent, we had to put ourselves in the position of a developer, i.e., to acquire knowledge of data models. This takes quite a long time, however, and this book is specifically designed to provide assistance here. Runtime statistics are what I always missed, and, in retrospect, I'm not surprised that analysis was often so difficult and based on "trial and error."

I'll just give you a brief outline of my personal approach, Peter.

Hanno: "Let's assume that we have found a problematical cursor. This is described in detail in [1]. Now the problem has to be made reproducible, without changing the productive data, of course."

Peter: "You find the SQL text and, if present, the bind variables. Then you execute the SQL statement in order to obtain the important runtime statistics."

© Springer-Verlag Berlin Heidelberg 2016

L. Nossov et al., *Formal SQL Tuning for Oracle Databases*,
DOI 10.1007/978-3-662-50417-8_7

H.: “That’s right. I prepare all the basic conditions for execution. The SQL statement is then executed and the plan with the function DBMS_STATS.DISPLAY_CURSOR, including further sections, is displayed.”

P.: “Then we have everything we need for the analysis?!”

H.: “Normally, yes. But there are a few exceptions. For example, with ‘bind peeking,’ if the binds are not representative because they were originally used for parsing the cursor, and Oracle is using the cursor again.”

P.: “That means it is the same plan but with completely different cardinalities in each step?”

H.: “Exactly. It’s quite possible that that can happen. With the data I have obtained, I then begin the analysis as discussed in the chapters ‘Bottlenecks in the Execution Plan’ and ‘Approaches to Formal SQL Tuning’. I identify the bottleneck and think about a solution. As licenses for EM packs are seldom available in my working environment, I am often unable to access the AWR or use SQL monitoring. Nor am I able to create any SQL profiles. If possible, therefore, I try to make structural changes to schema objects. For example, extending an index can solve some of our problem categories.”

7.1.1 Statistics on Problem Categories

During the preparation phase for this book, for statistical purposes, the authors recorded real cases involving acute performance problems. This data from practical applications resulted in several problem categories and their percentage distributions (Fig. 7.1).

Problem category	Frequency
a missing index	<5%
a non-selective index	47%
an index with a large clustering factor	10%
a sparse index	<5%
a sparse table	<5%
an inefficient nested loop join instead of a hash join	<5%
an inappropriate table order in a join of several tables	20%
a badly programmed SQL statement. No need for SQL tuning	<5%
other problems	7%

Fig. 7.1 Problem categories and their occurrence in practice

These statistics were collected from both OLTP and DWH systems. They confirm that formal SQL tuning covers most practical problems. Only 7 % belong to the "other problems" category, and even these were successfully analyzed and eliminated using the same formal principle.

There are usually enough indexes in the schemata of the applications. There is, therefore, only a small percentage of "missing indexes" in the statistics.

There are often a lot of single columns indexed, and, therefore, most cases belong to the problem category "nonselective index." This means that two columns are singly indexed, are not very selective, but are often queried together in SQL. A combined index of these columns can increase selectivity and solve the problem. It is therefore essential for a DBA to be able to analyze and remedy these problems. Below is a simple test case, which anyone can try out for themselves.

The second biggest problem category is "an inappropriate table order in a join." I would just briefly like to mention my own experience here. As this is a global problem, the solution is not quite so simple. One reason for the incorrect order is the fact that the optimizer wrongly estimates the cardinality of a table join (or of a table access). In my experience, one also often finds a nonselective index in such a case. If one improves (extends) the index, the order of the join also often changes for the better.

Sparse objects only occur when the application is in production. The objects are fragmented due to inserts, updates, and deletes, possibly resulting in large gaps which still have to be read. Indexes are very susceptible to this, for example, in the case of columns with sequential numbers, status change, or time fields.

The two largest categories in our table account for 67 % of all cases. A DBA can be fairly successful with SQL tuning if he can cope with these problems.

7.1.2 A Small Synthetic Test Case in Respect of a Nonselective Index

This case occurs very often in practice and is very easy to recognize. The solution is easy to implement following the rules of formal SQL tuning and without changing the SQL statement.

Hanno: "Could you comment on this, Peter…."

```
alter session set cursor_sharing=exact;
alter session set statistics_level=all;
drop table tab1;
create table tab1 as
select mod(level,2000) a,mod(level,666) b,mod(level,5) c from dual connect by level <= 2000000;
exec dbms_stats.gather_table_stats('SYS','TAB1');

select a,b,c from tab1 where a=1 and b between 40 and 300;
```

Id	Operation	Name	Cost (%CPU)	A-Rows	Buffers
0	SELECT STATEMENT		1190 (100)	390	4339
* 1	TABLE ACCESS FULL	TAB1	1190 (2)	390	4339

```
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(("A"=1 AND "B"<=300 AND "B">=40))

Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - "A"[NUMBER,22], "B"[NUMBER,22], "C"[NUMBER,22]
```

Peter: “There’s nothing to say here. The table without an index is read with an FTS.”

H.: “Then let’s create an index. What do you say now?”

```
create index tab1_i on tab1 (a);
select a,b,c from tab1 where a=1 and b between 40 and 300;
```

Id	Operation	Name	Cost (%CPU)	A-Rows	Buffers
0	SELECT STATEMENT		1005 (100)	390	1031
* 1	TABLE ACCESS BY INDEX ROWID	TAB1	1005 (0)	390	1031
* 2	INDEX RANGE SCAN	TAB1_I	5 (0)	1000	31

```
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(("B"<=300 AND "B">=40))
   2 - access("A"=1)

Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - "A"[NUMBER,22], "B"[NUMBER,22], "C"[NUMBER,22]
   2 - "TAB1".ROWID[ROWID,10], "A"[NUMBER,22]
```

P.: “The predicates show ‘access’ by the index. After that, the 1000 rows in the table are filtered. In the process, 1000 table blocks are accessed (1031−31). Here, however, one can also see that the clustering factor is very high. 1000 rows—1000 buffers. . ..”

H.: “Yes, that’s true. A poor clustering factor + a poor optimizer estimation of cardinality or a low selectivity of the index = a big performance problem.”

```
create index tab1_ii on tab1 (a,b);
select a,b,c from tab1 where a=1 and b between 40 and 300;
```

Id	Operation	Name	Cost (%CPU)	A-Rows	Buffers
0	SELECT STATEMENT		398 (100)	390	420
1	TABLE ACCESS BY INDEX ROWID	TAB1	398 (0)	390	420
* 2	INDEX RANGE SCAN	TAB1_II	4 (0)	390	30

```
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("A"=1 AND "B">=40 AND "B"<=300)

Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - "A"[NUMBER,22], "B"[NUMBER,22], "C"[NUMBER,22]
   2 - "TAB1".ROWID[ROWID,10], "A"[NUMBER,22], "B"[NUMBER,22]
```

P.: "Now the FILTER on 'b' has disappeared. All predicates are used in access with the index. That means that the table is now only accessed due to the projection of column 'c.' As you said, the index extension resulted in a reduction of the cardinality (nonselective index) in step 2 and in a reduction of the buffers in step 1."

H.: "Now let's add column 'c' to the index."

```
create index tab1_iii on tab1 (a,b,c);
select a,b,c from tab1 where a=1 and b between 40 and 300;
```

Id	Operation	Name	Cost (%CPU)	A-Rows	Buffers
0	SELECT STATEMENT		4 (100)	390	30
* 1	INDEX RANGE SCAN	TAB1_III	4 (0)	390	30

```
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("A"=1 AND "B">=40 AND "B"<=300)

Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - "A"[NUMBER,22], "B"[NUMBER,22], "C"[NUMBER,22]
```

P.: "That's great! Now the 390 blocks (420−30) are no longer necessary for table access either, and only 30 remain. The improvements are quite considerable. Is that only possible in the example or is it the same in practice?"

H.: "In my personal experience, it often works in practice. I use it very often, at least up to the first index extension in the test (tab1_ii). In order to squeeze the last drop out of it and use an index-only access, the conditions must be right."

P.: "And what are they. . .?"

H.: "(1) Only a few or very narrow columns, so that the index doesn't grow very much. (2) If the index increases into the double-digit percent range due to the change, there have to be good reasons, for example, if the SQL statements which undergo the improvements shown above are executed thousands of times per minute."

7.1.3 Practical Example

This example shows the evaluation by a technical administrator with two potential improvements and one special feature that he hadn't expected. As the data is mostly not in the cache at the moment of execution, I also simulate that accordingly with the command "alter system flush buffer_cache" before each execution.

The SQL statement:

```
select
tp.produktionsplan_typname,
tp.tai_typname,
to_char(t.letzteausfuehrung,'YYYY-MM-DD HH24') std ,
count(*)
from
tai t, tai_produktionsplan tp
where
t.letzteausfuehrung > to_timestamp('2014.10.27 00:00:00', 'YYYY-MM-DD HH24:MI:SS')
and t.letzteausfuehrung <= to_timestamp('2014.11.02 23:59:59', 'YYYY-MM-DD HH24:MI:SS')
and t.kennung = tp.tai_kennung
group by tp.produktionsplan_typname,tp.tai_typname,to_char(t.letzteausfuehrung,'YYYY-MM-DD HH24')
order by tp.produktionsplan_typname,tp.tai_typname,to_char(t.letzteausfuehrung,'YYYY-MM-DD
HH24');
```

We see the following conspicuous features in the execution plan.

– The “index range scan” in step 5 has a very high cardinality.
– After that, in step 4, access by rowid occurs almost four million times, and a large number of blocks are read in the process.

The conclusion is therefore as follows. A poor clustering factor + high cardinality on index access = a very big performance problem (Fig. 7.2).

Let’s return briefly to the section “Sections of the Execution Plan.”

If “access” were used in an "INDEX RANGE SCAN" operation, and it was then still necessary to use a further filter by rowid, there would be an asterisk in the plan step for "TABLE ACCESS BY ROWID" before the “Id.” In this case, it would be possible to extend the index by column(s) which belong to the filter. To avoid the access by rowid, one can expand the index by column(s) from the section “Column Projection Information” (these columns are queried in select).

Ideally, I would now try to prevent table access (step 4), so that Oracle can only work with index-only access. This assumes that all columns of the table which are required as filter criteria and projection are present in the index! Let’s check which columns are already present in the index.

```
SQL> select COLUMN_NAME,COLUMN_POSITION from dba_ind_columns where
index_name='IDX_TAI_LETZTEAUSF_KENN' order by COLUMN_POSITION;

COLUMN_NAME     COLUMN_POSITION
--------------- ---------------
SYS_NC00037$                  1
KENNUNG                       2
```

Now we come to a special feature. Let’s look at the details for the plan above. As expected, we have an asterisk in step 5. We don’t have an asterisk in step 4. Consequently, step 4 can only have a projection (no predicates).

Id	Operation	Name	A-Rows	A-Time	Buffers	Reads
0	SELECT STATEMENT		21916	01:04:52.62	6035K	5069K
1	SORT GROUP BY		21916	01:04:52.62	6035K	5069K
* 2	FILTER		3955K	01:04:34.66	6035K	5069K
* 3	HASH JOIN		3955K	01:04:33.41	6035K	5069K
4	TABLE ACCESS BY INDEX ROWID	TAI	3955K	00:57:03.52	3849K	2883K
* 5	INDEX RANGE SCAN	IDX_TAI_LETZTEAUSF_KENN	3955K	00:00:30.94	27601	31626
6	TABLE ACCESS FULL	TAI_PRODUKTIONSPLAN	160M	00:09:14.98	2185K	2185K

Fig. 7.2 Practical example: the problematical execution plan

```
Predicate Information:
   2 - filter(SYS_EXTRACT_UTC(TIMESTAMP' 2014-11-02
23:59:59.000000000')>SYS_EXTRACT_UTC(TIMESTAMP' 2014-10-27 00:00:00.000000000'))
   3 - access("T"."KENNUNG"="TP"."TAI_KENNUNG")
   5 - access("T"."SYS_NC00037$">SYS_EXTRACT_UTC(TIMESTAMP' 2014-10-27 00:00:00.000000000') AND
"T"."SYS_NC00037$"<=SYS_EXTRACT_UTC(TIMESTAMP' 2014-11-02 23:59:59.000000000'))

Column Projection Information:
   4 - "T"."KENNUNG"[NUMBER,22], "LETZTEAUSFUEHRUNG"[TIMESTAMP WITH TIME ZONE,13]
   5 - "T".ROWID[ROWID,10], "T"."SYS_NC00037$"[TIMESTAMP,11], "T"."KENNUNG"[NUMBER,22]
```

What is conspicuous is the use of the function SYS_EXTRACT_UTC, because it is not even specified in the SQL statement. In addition, the column name "SYS_NC00037$" indicates an FBI (function-based index).

Here we have a special case. The column "LETZTEAUSFUEHRUNG" is of the type "TIMESTAMP WITH TIME ZONE." As this is not an absolute value, Oracle stores it in UTC format (and uses the function SYS_EXTRACT_UTC for this purpose). An index on such a column is automatically generated as an FBI.

This is also demonstrated by a short test (Fig. 7.3).

The values to be displayed in the SQL statement TO_CHAR ("LETZTEAUSFUEHRUNG",'YYYY-MM-DD HH24') cannot then be calculated from the index. It is therefore also necessary to access rowid in step 4 to enable projection.

Note: An FBI with SYS_EXTRACT_UTC is implicitly generated only for TIMESTAMP WITH TIME ZONE (not for local time zone).

7.1.3.1 The First Improvement

The existing index is extended with the column TO_CHAR ("LETZTEAUSFUEHRUNG",'YYYY-MM-DD HH24'). This completely dispenses with the need to access the table.

```
CREATE INDEX IDX_TAI_LETZTEAUSF_KENN2 ON TAI ("LETZTEAUSFUEHRUNG", "KENNUNG",
TO_CHAR("LETZTEAUSFUEHRUNG",'YYYY-MM-DD HH24'));
```

As a result, table access to TAI is totally eliminated. The projection of the timestamp now occurs in step 4. In this example, the column T.SYS_NC00040$ belongs to the FBI (Fig. 7.4).

In principle, the improvement in runtime from 1 hour 4 minutes to 11 minutes was great progress, but I also wanted to try to accelerate the FTS.

7.1.3.2 The Second Improvement

An access to the table TAI_PRODUKTIONSPLAN only with index would have meant adding three columns to the index.

```
SQL> create table tab1 (column1 timestamp with time zone);
Table created.

SQL> create index ind1 on tab1 (column1);
Index created.

SQL> select dbms_metadata.get_ddl('INDEX','IND1','SYS') from dual;
DBMS_METADATA.GET_DDL('INDEX','IND1','SYS')
--------------------------------------------------------------------------------


  CREATE INDEX "SYS"."IND1" ON "SYS"."TAB1" (SYS_EXTRACT_UTC("COLUMN1"))  PCTFREE 10 INITRANS 2
MAXTRANS 255 COMPUTE STATISTICS  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
2147483645 PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT
CELL_FLASH_CACHE DEFAULT) TABLESPACE "SYSTEM"
```

Fig. 7.3 Test case: index on a column of the type "TIMESTAMP WITH TIME ZONE"

```
Predicate Information (identified by operation id):
---------------------------------------------------
3 - access("T"."KENNUNG"="TP"."TAI_KENNUNG")

Column Projection Information (identified by operation id):
-----------------------------------------------------------
5 - "TP"."TAI_KENNUNG"[NUMBER,22], "TP"."TAI_TYPNAME"[VARCHAR2,200],
"TP"."PRODUKTIONSPLAN_TYPNAME"[VARCHAR2,200]
```

This index would have become too large, and, as it was not a case of a high-frequency SQL query but of a kind of ad hoc report, I decided on parallelizing the FTS by means of the hint PARALLEL(TP 8) (Fig. 7.5).

My colleague was amazed to see the result of approximately 1-minute runtime instead of the 1 hour he had expected.

Id	Operation	Name	A-Rows	A-Time	Buffers	Reads
0	SELECT STATEMENT		21916	00:11:33.09	2218K	2218K
1	SORT GROUP BY		21916	00:11:33.09	2218K	2218K
* 2	FILTER		3955K	00:11:20.52	2218K	2218K
* 3	HASH JOIN		3955K	00:11:19.29	2218K	2218K
* 4	INDEX RANGE SCAN	IDX_TAI_LETZTEAUSF_KENN2	3955K	00:00:10.04	32165	32448
5	TABLE ACCESS FULL	TAI_PRODUKTIONSPLAN	160M	00:08:55.33	2185K	2185K

```
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(SYS_EXTRACT_UTC(TIMESTAMP' 2014-11-02
23:59:59.000000000')>SYS_EXTRACT_UTC(TIMESTAMP' 2014-10-27 00:00:00.000000000'))
   3 - access("T"."KENNUNG"="TP"."TAI_KENNUNG")
   4 - access("T"."SYS_NC00037$">SYS_EXTRACT_UTC(TIMESTAMP' 2014-10-27 00:00:00.000000000') AND
"T"."SYS_NC00037$"<=SYS_EXTRACT_UTC(TIMESTAMP' 2014-11-02 23:59:59.000000000'))

Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - (#keys=3) "TP"."PRODUKTIONSPLAN_TYPNAME"[VARCHAR2,200], "TP"."TAI_TYPNAME"[VARCHAR2,200],
"T"."SYS_NC00040$"[VARCHAR2,13], COUNT(*)[22]
   2 - "T"."SYS_NC00040$"[VARCHAR2,13], "TP"."PRODUKTIONSPLAN_TYPNAME"[VARCHAR2,200],
"TP"."TAI_TYPNAME"[VARCHAR2,200]
   3 - (#keys=1) "T"."SYS_NC00040$"[VARCHAR2,13], "TP"."PRODUKTIONSPLAN_TYPNAME"[VARCHAR2,200],
"TP"."TAI_TYPNAME"[VARCHAR2,200]
   4 - "T"."KENNUNG"[NUMBER,22], "T"."SYS_NC00040$"[VARCHAR2,13]
   5 - "TP"."TAI_KENNUNG"[NUMBER,22], "TP"."TAI_TYPNAME"[VARCHAR2,200],
"TP"."PRODUKTIONSPLAN_TYPNAME"[VARCHAR2,200]
```

Fig. 7.4 Practical example: the first improvement

Id	Operation	Name	A-Rows	A-Time
0	SELECT STATEMENT		15337	00:01:16.53
* 1	PX COORDINATOR		15337	00:01:16.53
2	PX SEND QC (ORDER)	:TQ10003	0	00:00:00.01
3	SORT ORDER BY		14525	00:00:00.09
4	PX RECEIVE		13503	00:00:00.06
5	PX SEND RANGE	:TQ10002	0	00:00:00.01
6	SORT GROUP BY		15337	00:00:00.21
7	PX RECEIVE		57299	00:00:00.06
8	PX SEND HASH	:TQ10001	0	00:00:00.01
9	HASH GROUP BY		42271	00:10:04.35
* 10	FILTER		2888K	00:10:00.34
* 11	HASH JOIN		2888K	00:09:59.18
12	BUFFER SORT		25M	00:05:43.39
13	PX RECEIVE		22M	00:04:06.98
14	PX SEND BROADCAST	:TQ10000	0	00:00:00.01
* 15	INDEX RANGE SCAN	IDX TAI LETZTEAUSF KENN2	3224K	00:00:32.82
16	PX BLOCK ITERATOR		13M	00:03:05.19
* 17	TABLE ACCESS FULL	TAI_PRODUKTIONSPLAN	10M	00:03:27.23

Fig. 7.5 Practical example: the second improvement

7.2 Victor's Experience

I had been working with Oracle database operations at a large company for a long time when Leonid came to give a talk on formal SQL tuning at the beginning of 2013. As my special area was precisely the field of optimization and troubleshooting, I was all ears. I knew full well how much effort was involved in optimization by the "trial and error" method. I often spent many hours unsuccessfully trying to optimize complex queries because my assumptions were incorrect. Even in relatively simple execution plans, I was not always sure if an index access was really a good idea at that point. I was only able to improve execution after a detailed, time-consuming data distribution analysis. The method Leonid presented promised to save a lot of time and I was quite excited about it. I have developed some auxiliary programs for use with this method, and I always employ them for SQL tuning.

Whenever an acute database incident has to be dealt with, or a repeated or constant performance issue occurs, formal SQL evaluation is employed sooner or later. It's amazing how much time this saves! Sometimes optimum plans are found for problematical queries, and sometimes index structures can be improved. If no improvement is possible without changing the relevant SQL statement, concrete optimization recommendations are prepared for development. Sometimes it is evident that the problem cannot be solved either by SQL optimization or by redesigning the application (data model design error). Thanks to the formal method, the analysis can be done very quickly.

Inefficient plans occur suddenly and have various causes, such as massive data changes, increase in data volume, introduction of new software, outdated or missing optimizer statistics, etc. In case of strong fluctuations in data volume, the optimizer generates a large number of different plans for the same SQL statement. That's why I first of all check whether it is possible to find another, better execution plan for the problematical SQL statement in the AWR. If so, the better plan can be activated. The optimization process is often completed after this step.

If it is not possible to find a suitable plan in the AWR, I prepare test versions of the SQL statement for the purpose of SQL tuning. With my auxiliary programs, I find the relevant SQL text in the database; I draw up a summary of the segments and index structures involved. (This can be useful for the analysis.) If necessary, I correct the SQL text so that my tests do not create or delete any database objects or change or lock their data. For this purpose, I extract the relevant selects from the DDL and DML commands and remove any existing "FOR UPDATE" clauses from the commands "select for update." It is worth formatting the SQL text. (There are a lot of tools available for this free of charge.)

I then check the parameter settings which are relevant for the optimizer (primarily those not documented). An inappropriately set parameter of this sort can hinder the optimizer in the search for an optimum plan. When testing, I reset some of these parameters to their default values and check the execution plan. The optimization process is often successfully completed after this step.

www.allitebooks.com

In practice, I use the SQL Tuning Advisor (SQLT advisor) from Oracle. If the SQLT advisor finds an alternative plan, then it is a good reason to test this plan. However, I would not advise activating an automatically generated plan without testing it. Even if the SQLT advisor promises a substantial improvement, this does not always materialize. If the SQLT advisor does a good job and saves me a lot of work, I am very pleased.

If manual SQL tuning is necessary, I follow the formal method as described in this book. I find this method very effective and hope that it will also help our readers. In order to change the order of accesses and access methods, I almost exclusively use hints from outlines as a template. Compared to "classical" hints, they take some getting used to at first. They refer directly to query blocks and can therefore be placed in any query block (e.g., at the beginning of the SQL statement). It is no longer necessary to distribute hints throughout the SQL text. This makes hints easier to use. In the case of views in SQL statements, this is the only possibility to address the relevant query blocks without changing these views.

I use mostly hidden hints for SQL tuning because it is not usually possible to change the SQL statement. As a rule, I employ the OSP method for this (Outlines in SQL Profiles) as described in [1].

Once the new plan has been activated, I monitor its effect over a lengthy period. If this plan does not prove to be as effective as expected, I have a "drop profile" command ready. This happens from time to time when certain nonrelevant bind values have been used in tests, for example, when they refer to yesterday's data (instead of up-to-date data). One has to be careful with bind values when tuning.

7.2.1 The First Practical Example

A small example of formal SQL tuning. A SQL statement was running on a number of instances and was using the lion's share of CPU resources, mostly without returning a single row. Below is the formatted text of this SQL statement.

```
SELECT   business_activity_id, type_rd, status_rd, NAME, priority_rd, root_business_activity_id
    FROM business_activity t
   WHERE workflow_template_id = '.bF6exCI1PAi079H'
     AND NAME = 'Deaktivieren'
     AND status_rd = 'InProg'
     AND external_system_indicator_rd = 'ERROR'
     AND service_order_stp_id IN (
           SELECT so.service_order_stp_id
             FROM service_order so
                , service_property sp
                , property_value pv
            WHERE so.service_order_stp_id = sp.service_order_stp_id
              AND sp.service_property_id = pv.service_property_id
              AND pv.value_string = 'ffmaems2')
ORDER BY business_activity_id
```

Figure 7.6 shows the problematical execution plan.

598K rows are read from the table PROPERTY_VALUE. The joining of this table with the table SERVICE_PROPERTY (alias SP) results in no hits. This makes a case for changing the table order in this join. The table SERVICE_PROPERTY has an index SP_IDX1 for the column SERVICE_ORDER_STP_ID which can be used for this change.

```
Table Name                     Index Name      Column List
------------------------------ --------------- -----------------------------------
SERVICE_PROPERTY               SP_IDX1         SERVICE_ORDER_STP_ID
SERVICE_PROPERTY               SP_PK1          SERVICE_PROPERTY_ID,SERVICE_ORDER_STP_ID
```

If we change the order with the hint LEADING (in this case, exceptionally, I didn't use any outlines), we get a completely different execution schema (Fig. 7.7).

Id	Operation	Name	Starts	A-Rows	A-Time	Buffers
0	SELECT STATEMENT		1	0	00:44:14.49	3321K
1	SORT ORDER BY		1	0	00:44:14.49	3321K
2	NESTED LOOPS SEMI		1	0	00:44:14.49	3321K
* 3	TABLE ACCESS BY INDEX ROWID	BUSINESS_ACTIVITY	1	1	00:00:00.12	19
* 4	INDEX RANGE SCAN	BA_NAME_IND	1	15	00:00:00.04	4
5	VIEW PUSHED PREDICATE	VW_NSO_1	1	0	00:44:14.37	3321K
6	NESTED LOOPS		1	0	00:44:14.37	3321K
7	TABLE ACCESS BY INDEX ROWID	PROPERTY_VALUE	1	598K	00:17:18.26	547K
* 8	INDEX RANGE SCAN	PV_IND2	1	598K	00:00:02.45	4476
* 9	INDEX UNIQUE SCAN	SP_PK1	598K	0	01:38:30.37	2773K

```
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(("WORKFLOW_TEMPLATE_ID"='.bF6exCIlPAi079H' AND
"EXTERNAL_SYSTEM_INDICATOR_RD"='ERROR' AND "STATUS_RD"='InProg'))
   4 - access("NAME"='Deaktivieren')
   8 - access("PV"."VALUE_STRING"='ffmaems2')
   9 - access("SP"."SERVICE_PROPERTY_ID"="PV"."SERVICE_PROPERTY_ID" AND
"SP"."SERVICE_ORDER_STP_ID"="SERVICE_ORDER_STP_ID")
```

Fig. 7.6 Example 1: A suboptimal execution plan

```
SELECT   business_activity_id, type_rd, status_rd, NAME, priority_rd, root_business_activity_id
    FROM business_activity t
   WHERE workflow_template_id = '.bF6exCIlPAi079H'
     AND NAME = 'Deaktivieren'
     AND status_rd = 'InProg'
     AND external_system_indicator_rd = 'ERROR'
     AND service_order_stp_id IN (
           SELECT /*+ leading(sp) use_nl(sp pv) */
                  so.service_order_stp_id
             FROM service_order so
                , service_property sp
                , property_value pv
            WHERE so.service_order_stp_id = sp.service_order_stp_id
              AND sp.service_property_id = pv.service_property_id
              AND pv.value_string = 'ffmaems2')
ORDER BY business_activity_id
```

Id	Operation	Name	Starts	A-Rows	A-Time	Buffers
0	SELECT STATEMENT		1	0	00:00:00.68	393
1	SORT ORDER BY		1	0	00:00:00.68	393
2	NESTED LOOPS SEMI		1	0	00:00:00.68	393
* 3	TABLE ACCESS BY INDEX ROWID	BUSINESS_ACTIVITY	1	1	00:00:00.01	19
* 4	INDEX RANGE SCAN	BA_NAME_IND	1	15	00:00:00.01	4
5	VIEW PUSHED PREDICATE	VW_NSO_1	1	0	00:00:00.67	374
6	NESTED LOOPS		1	0	00:00:00.67	374
7	NESTED LOOPS		1	177	00:00:00.61	319
8	TABLE ACCESS BY INDEX ROWID	SERVICE_PROPERTY	1	100	00:00:00.06	12
* 9	INDEX RANGE SCAN	SP_IDX1	1	100	00:00:00.03	6
* 10	INDEX RANGE SCAN	PV_PK	100	177	00:00:00.56	307
* 11	TABLE ACCESS BY INDEX ROWID	PROPERTY_VALUE	177	0	00:00:00.06	55

```
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(("WORKFLOW_TEMPLATE_ID"='.bF6exCIlPAi079H' AND
"EXTERNAL_SYSTEM_INDICATOR_RD"='ERROR' AND "STATUS_RD"='InProg'))
   4 - access("NAME"='Deaktivieren')
   9 - access("SP"."SERVICE_ORDER_STP_ID"="SERVICE_ORDER_STP_ID")
  10 - access("SP"."SERVICE_PROPERTY_ID"="PV"."SERVICE_PROPERTY_ID")
  11 - filter("PV"."VALUE_STRING"='ffmaems2')
```

Fig. 7.7 Example 1: The execution plan after optimization

An interesting aspect of this example is the fact that the table SERVICE_ORDER does not appear once in the plan. This enables the foreign key SERVICE_PROPERTY. SERVICE_ORDER_STP_ID -> SERVICE_ORDER. SERVICE_ORDER_STP_ID. At Oracle, this type of optimization is called join elimination and is available from version 10.2 onward. One indication that Oracle uses this optimization is the hint ELIMINATE_JOIN in the outlines.

The small test case which is based on our practical example reproduces this behavior.

```
create table parent (
  a number, b number,
  constraint parent_pk primary key (a));

create table child (
  a number, b number,
  constraint child_pk primary key (a),
  constraint child_fk foreign key (a) references parent (a));
```

The query

```
select * from child where a in (select a from parent);
```

runs with the following plan:

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT				2
1	TABLE ACCESS FULL	CHILD	1	26	2

```
Outline Data
-------------

  /*+
      ...
      ELIMINATE_JOIN(@"SEL$5DA710D3" "PARENT"@"SEL$2")
      ...
  */
```

If the foreign key is a non-validated condition (e.g., after a table reorganization), this optimization can no longer be used by Oracle:

```
alter table child enable novalidate constraint child_fk;

select * from child where a in (select a from parent);
```

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT				2
1	NESTED LOOPS		1	39	2
2	INDEX FULL SCAN	PARENT_PK	1	13	1
3	TABLE ACCESS BY INDEX ROWID	CHILD	1	26	1
* 4	INDEX UNIQUE SCAN	CHILD_PK	1		1

7.2.2 The Second Practical Example

In this section I would like to present another interesting example. I would be very grateful if Peter Smith would help me.

Peter: "Of course I'll help you. I welcome any opportunity to practice SQL tuning."

Victor: "Fig. 7.8 shows an execution plan. This plan has caused a serious performance problem in a system. How would you improve this plan, Peter?"

```
Plan hash value: 931175730
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		55	00:01:50.54
* 1	HASH JOIN RIGHT SEMI		1	1	55	00:01:50.54
2	VIEW	VW_NSO_1	1	102	102	00:00:11.57
3	HASH GROUP BY		1	102	102	00:00:11.57
4	INDEX FAST FULL SCAN	SENSORDATA_IDX	1	8300K	8421K	00:00:03.99
5	NESTED LOOPS		1	70553	8016K	00:01:33.21
6	NESTED LOOPS		1	1	55	00:00:00.01
7	NESTED LOOPS		1	1	55	00:00:00.01
8	MERGE JOIN		1	1	55	00:00:00.01
* 9	TABLE ACCESS BY INDEX ROWID	HIERARCHY	1	1	7	00:00:00.01
10	INDEX FULL SCAN	HIERARCHY_PK	1	120	120	00:00:00.01
* 11	SORT JOIN		7	120	55	00:00:00.01
12	TABLE ACCESS FULL	HIERARCHY	1	120	120	00:00:00.01
13	TABLE ACCESS BY INDEX ROWID	SENSOR	55	1	55	00:00:00.01
* 14	INDEX UNIQUE SCAN	SENSOR_PK	55	1	55	00:00:00.01
15	TABLE ACCESS BY INDEX ROWID	ITEM	55	1	55	00:00:00.01
* 16	INDEX UNIQUE SCAN	ITEM_PK	55	1	55	00:00:00.01
* 17	INDEX FAST FULL SCAN	SENSORDATA_IDX	55	69170	8016K	00:01:30.29

```
Outline Data
-------------

  /*+
      …
      LEADING(@"SEL$CC7EC59E" "HIERARCHY"@"SEL$3" "HIERARCHY"@"SEL$1" "SENSOR"@"SEL$1"
"ITEM"@"SEL$1" "SENSORDATA"@"SEL$1" "VW_NSO_1"@"SEL$CC7EC59E")
      …
  */

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("SENSORDATA"."ID"="MAX(SENSORDATA.ID)")
   9 - filter("HIERARCHY"."PARENT"=52)
  11 - access("HIERARCHY"."PARENT"="HIERARCHY"."ID")
       filter("HIERARCHY"."PARENT"="HIERARCHY"."ID")
  14 - access("HIERARCHY"."ID"="SENSOR"."ID")
  16 - access("HIERARCHY"."ID"="ITEM"."ID")
  17 - filter("HIERARCHY"."ID"="SENSORDATA"."SENSORID")
```

Fig. 7.8 Example 2: The problematical execution plan

P.: "Both the highest cardinality of 8016K and the longest runtime of 1½ minutes occur in step 17. After the join with the view VW_NSO_1, the cardinality falls to 55. I would change the table order in the join in such a way that the table SENSORDATA follows the view VW_NSO_1. But. . .."

V.: "What?"

P.: "I've just noticed that that isn't possible because this view and the table SENSORDATA are joined with a right outer join."

V.: "You have fallen into the same trap as me. Where do you see an outer join?"

P.: "In step 1."

V.: "But that isn't a right outer join. It's a hash join right semi! The word 'semi' tells us that it is a join with a subquery as an inline view. The word 'right' indicates that this inline view plays the role of the build table in the hash join. As this is not an outer join, I have used the hint LEADING (@"SEL$CC7EC59E" "VW_NSO_1"@"SEL$CC7EC59E" "SENSORDATA"@"SEL$1") and obtained a runtime of approx. 5 seconds. As it was not possible to change the SQL statement. . .."

P.: "You probably used the OSP method and created a SQL profile?"

```
Plan hash value: 1387967720
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		55	00:00:05.19
* 1	HASH JOIN		1	33	55	00:00:05.19
* 2	HASH JOIN		1	33	55	00:00:05.19
* 3	HASH JOIN		1	39	55	00:00:05.19
* 4	HASH JOIN		1	102	102	00:00:05.18
5	NESTED LOOPS		1	102	102	00:00:05.18
6	VIEW	VW_NSO_1	1	102	102	00:00:05.18
7	HASH UNIQUE		1	102	102	00:00:05.18
8	HASH GROUP BY		1	102	102	00:00:05.17
9	INDEX FAST FULL SCAN	SENSORDATA_IDX	1	7718K	7744K	00:00:01.93
* 10	INDEX RANGE SCAN	SENSORDATA_IDX	102	1	102	00:00:00.01
11	TABLE ACCESS FULL	HIERARCHY	1	120	120	00:00:00.01
* 12	TABLE ACCESS FULL	HIERARCHY	1	7	7	00:00:00.01
13	TABLE ACCESS FULL	SENSOR	1	102	102	00:00:00.01
14	TABLE ACCESS FULL	ITEM	1	120	120	00:00:00.01

Fig. 7.9 Example 2: The execution plan after generation of histograms for the column PARENT

V.: “Yes, and I was immediately asked whether there was another solution without hidden hints.”

P.: “Why?”

V.: “SQL profiles (like SQL plan baselines) have to be maintained. If the database is transferred to another computer, for example, one mustn’t forget to transfer all created SQL profiles. Although I personally do not regard this as any great effort, I have tried to find another solution.”

P.: “What’s that?”

V.: “I noticed that the optimizer does not correctly estimate the cardinality when the table HIERARCHY is accessed in step 9. It assumes that this cardinality is 1. In fact, it was 7. As the predicate ‘HIERARCHY. PARENT=52’ was used for access to the table HIERARCHY, I checked if the column PARENT had histograms. I checked this in the view DBA_TAB_COL_STATISTICS.”

P.: “I presume you didn’t find any histograms.”

V.: “You’re right. After creating histograms for the column PARENT, the execution plan improved immediately (see Fig. 7.9). The optimizer selected the same plan as I forced with the hint LEADING.”

P.: “In this case, the creation of histograms was a good alternative to the formal method. Is that always the case? Can one always generate new or additional optimizer statistics instead of using formal SQL tuning?”

V.: “In some situations that is possible, but certainly not in all. For example, if the optimizer estimates the cardinality of a join badly with a skewed distribution of data, as a rule, no statistics will help.”

8 Closing Remarks

That is the end of the book. We would be pleased if you decide to add formal SQL tuning to your arsenal and use it in everyday practice. Do that and you will be amazed how easy and effective this method is. With this figure, we take our leave (Fig. 8.1).

In the appendix we use an example to describe how one can use the formal principle for the analysis of performance problems after an Oracle migration. Such problems are considerably more complicated than the SQL tuning of individual SQL statements because several SQL statements are usually affected. When analyzing these problems, one has to determine what has caused the degradation in performance (e.g., new optimizer features). The formal principle can be very helpful here. If you would like to study the formal method in more detail, the following material will be of interest to you.

Before you close the book and put it down, we would like to take a final opportunity to clarify any outstanding questions. Peter may be able to help us again here. He is full of questions and doesn't give up until he receives a satisfactory answer. We hope that his questions are also of interest to you.

Peter: "I must admit that I do have some questions. These don't refer directly to the formal method itself, which I have understood (at least I hope so). They are of a more peripheral nature. First I would like to ask what formal SQL tuning is. Is it a method at all or is it simply a collection of empirical rules, which one can use when tuning?"

Author: "If we define a method as a systematic procedure for achieving an objective, then formal SQL tuning is a method. In this book, we have tried to systematize formal SQL tuning. The objective is also clear. Where do you see a problem?"

P.: "Formal SQL tuning doesn't cover all problems that can occur during tuning."

A.: "That's true. Most practical cases of acute performance problems are dealt with, however. If necessary, one can develop the method further using the same principle."

© Springer-Verlag Berlin Heidelberg 2016

L. Nossov et al., *Formal SQL Tuning for Oracle Databases*,
DOI 10.1007/978-3-662-50417-8_8

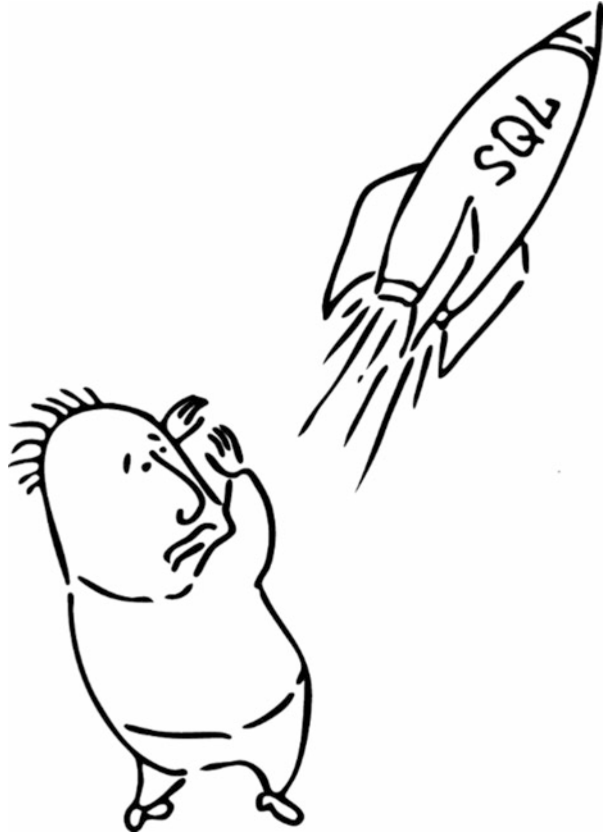


Fig. 8.1 Formal tuning exceeds your expectations

P.: "Can one algorithmize and program this procedure?"

A.: "Theoretically, one can create an algorithm for the formal method, but I would not program this procedure."

P.: "Why not?"

A: "I can give you a couple of reasons. The analysis of predicates plays a very important role in this method. Unfortunately, Oracle does not always generate predicates correctly in the execution plan. In most cases, one can obtain the correct predicates from the relevant explain plan. This complicates the program, however. Analysis of predicates is generally complicated. In [4] you can find scripts in which predicates are extracted and analyzed. Unfortunately, this does not always work because some cases are not covered."

P.: "I understand. The relevant program must be quite complicated. Are there any other reasons?"

A.: "Formal SQL tuning is a fairly simple method, which even beginners can master without difficulty. I really don't think that a program is necessary."

P.: “Are there cases where the formal method doesn’t work?”

A.: “Every method has its limits, Peter. I must say, however, that formal SQL tuning is very reliable, at least for the categories of problem described in this book.”

P.: “I am particularly interested in problems with an inappropriate table order in the join. Can it ever happen that no table order change made according to the rules described is capable of bringing about an appreciable improvement in performance?”

A.: “There are cases in which it is generally not possible to achieve any improvement by changing the table order in the join. This is normally due to the data model, which is unsuitable for the particular query.”

P.: “Do you mean that formal SQL tuning has absolutely no disadvantages?”

A.: “Not at all. For example, formal SQL tuning doesn’t consider any Oracle transformations and optimizations, because it is very problematical to formalize them in an easy manner and to incorporate them into the formal method. An experienced specialist may be able to take into account such Oracle features when tuning. An inexperienced person will generally not succeed in doing this. I must say, however, that transformations and optimizations seldom play a decisive role in SQL tuning (as, e.g., in the case in the appendix). For this reason, I do not see this as any major disadvantage.”

P.: “I have no further questions. Thank you very much.”

A.: “Then I wish you every success using the formal method.”

Appendix: Application of the Formal Principle for the Analysis of Performance Problems After an Oracle Migration

The method described in this book is helpful in most cases which a database specialist is likely to encounter in everyday practice. It is also relatively simple. With this method, we have tried to achieve a compromise between comprehensibility and usefulness. We will leave it to you, our readers, to judge how successful we have been.

Here we would like to present an example of how one can use the formal principle for analyzing performance problems after an Oracle migration.

Peter: "Why are you describing this in the appendix and not in a chapter?"

Author: "This is a special case because one relatively seldom encounters such problems (one doesn't migrate Oracle databases every day!). So, for practical reasons, we are putting this example in the appendix."

P.: "Why do you find this example so interesting?"

A.: "First of all, this example demonstrates that the formal method described here has its limitations; secondly, it shows how one can analyze relatively complex problems using the same principle. Shall we get started?"

P.: "Yes, all right. I hope I can follow you."

A.: "After an Oracle migration from 10.2.0.5 to 11.2.0.4, several SQL statements became suboptimal. They had a similar structure. Below is an example:

© Springer-Verlag Berlin Heidelberg 2016

L. Nossov et al., *Formal SQL Tuning for Oracle Databases*,
DOI 10.1007/978-3-662-50417-8

```
SELECT *
    FROM    SERVICE_AGREEMENT SA,SOC ,PROMOTION_TERMS PRMT
    WHERE   SA.BAN = 116
    AND     SA.SUBSCRIBER_NO = 'XXXXXXXXXXXX'
    AND     SA.EXPIRATION_DATE <= TO_DATE('20150102', 'YYYYMMDD')
    AND     SA.SOC_SEQ_NO = (SELECT --+ index(SA2 SERVICE_AGREEMENT_PK)
                 MAX(SA2.SOC_SEQ_NO)
                 FROM   SERVICE_AGREEMENT SA2
                 WHERE  SA2.BAN = SA.BAN
                 AND    SA2.SUBSCRIBER_NO = SA.SUBSCRIBER_NO
                 AND    SA2.SOC = SA.SOC
                 AND    SA2.EXPIRATION_DATE = SA.EXPIRATION_DATE)
    AND     SA.SOC = SOC.SOC
    AND     SA.TARIFF_OPTION IN ('VF_FUN_10', '000000000')
    AND     DECODE(RTRIM(SOC.TARGET_TARIFF),
           RTRIM('VF_FUN_10'), 0,
           RTRIM('VF_FUN'), 1,
           'ALLTO', 2) = (SELECT --+ index(S SOC_PK)
                  MIN(DECODE(RTRIM(S.TARGET_TARIFF),
                         RTRIM('VF_FUN_10'), 0,
                         RTRIM('VF_FUN'), 1,
                         'ALLTO', 2))
                  FROM  SOC S
                  WHERE S.SOC = SOC.SOC
                  AND  '20150102' >= TO_CHAR(S.EFFECTIVE_DATE, 'YYYYMMDD')
                  AND  TO_DATE('20150102', 'YYYYMMDD') < NVL(S.EXPIRATION_DATE,
TO_DATE('47001231', 'YYYYMMDD')))
    AND  '20150102' >= TO_CHAR(SOC.EFFECTIVE_DATE, 'YYYYMMDD')
    AND  TO_DATE('20150102', 'YYYYMMDD') < NVL(SOC.EXPIRATION_DATE, TO_DATE('47001231',
'YYYYMMDD'))
    AND  PRMT.SOC(+) = SOC.SOC
    AND  PRMT.SOC(+) = SOC.SOC
    AND  PRMT.TARGET_TARIFF(+) = SOC.TARGET_TARIFF
    AND  PRMT.EFFECTIVE_DATE(+) = SOC.EFFECTIVE_DATE
  ORDER BY DECODE(SA.SERVICE_TYPE,
            'P', 1,
            'M', 2, 3),
    SA.EFFECTIVE_DATE;
```

The relevant execution plan was as follows (Fig. A.1).

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		23	00:00:06.99
1	SORT ORDER BY		1	1	23	00:00:06.99
2	NESTED LOOPS		1	1	23	00:00:06.99
3	NESTED LOOPS		1	1	23	00:00:06.99
4	NESTED LOOPS OUTER		1	1	1980	00:00:00.04
5	NESTED LOOPS		1	1	1980	00:00:00.03
6	PARTITION RANGE SINGLE		1	1	23	00:00:00.01
* 7	TABLE ACCESS BY LOCAL INDEX ROWID	SERVICE_AGREEMENT	1	1	23	00:00:00.01
* 8	INDEX RANGE SCAN	SERVICE_AGREEMENT_3IX	1	1	36	00:00:00.01
* 9	MAT_VIEW ACCESS BY INDEX ROWID	SOC	23	1	1980	00:00:00.03
* 10	INDEX RANGE SCAN	SOC_99IX	23	18	9811	00:00:00.01
11	MAT_VIEW ACCESS BY INDEX ROWID	PROMOTION_TERMS	1980	1	0	00:00:00.01
* 12	INDEX UNIQUE SCAN	PROMOTION_TERMS_PK	1980	1	0	00:00:00.01
* 13	VIEW PUSHED PREDICATE	VW_SQ_2	1980	1	23	00:00:06.95
* 14	FILTER		1980		1980	00:00:06.94
15	SORT AGGREGATE		1980	1	1980	00:00:06.94
* 16	MAT_VIEW ACCESS BY INDEX ROWID	SOC	1980	1	370K	00:00:06.76
* 17	INDEX RANGE SCAN	SOC_PK	1980	1	1861K	00:00:02.74
* 18	VIEW PUSHED PREDICATE	VW_SQ_1	23	1	23	00:00:00.01
* 19	FILTER		23		23	00:00:00.01
20	SORT AGGREGATE		23	1	23	00:00:00.01
* 21	FILTER		23		23	00:00:00.01
22	PARTITION RANGE SINGLE		23	1	23	00:00:00.01
* 23	TABLE ACCESS BY LOCAL INDEX ROWID	SERVICE_AGREEMENT	23	1	23	00:00:00.01
* 24	INDEX RANGE SCAN	SERVICE_AGREEMENT_PK	23	1	36	00:00:00.01

Fig. A.1 The suboptimal execution plan after the Oracle migration

Can you see anything special about this execution plan, Peter?"

P.: "I assume that Oracle has transformed the two subqueries into inline views and joined them to the main query."

A.: "Quite right, Peter. Oracle has generated two names for these views: VW_SQ_1 and VW_SQ_2. The abbreviation 'SQ' means subquery. Notice that Oracle uses the operation 'VIEW PUSHED PREDICATE' for these two views. This means that the join predicates are pushed into the inline view (predicate push down). In our case, this happens in a natural way because these predicates are already contained in the subqueries. The view VW_SQ_2 corresponds to the second subquery. The join predicate which is pushed into this view is 'S.SOC=SOC.SOC.' We find this predicate in the section 'Predicate Information':

```
17 - access("S"."SOC"="SOC"."SOC")
     filter(TO_CHAR(INTERNAL_FUNCTION("S"."EFFECTIVE_DATE"),'YYYYMMDD')<='20150102')
```

We also have this predicate in the second subquery."

P.: "The problematical step in the execution plan above is step 17, in which 1861K rows were found. The optimizer estimated the cardinality for this step as 1. Actually, however, it was approx. 1000 (1861K/1980). I would have checked the optimizer statistics of the table SOC."

A.: "Wait a minute, Peter. I don't think that this measure would have reduced the runtime to 0.07 seconds (that's what it was in 10.2.0.5). In addition, all

optimizer statistics on this database were locked. There was, therefore, no possibility of solving the problem by changing the statistics. The real problem was that several SQL statements became suboptimal after the migration. It was necessary to find out the reason."

P.: "I would have tried the parameter setting optimizer_features_enable ='10.2.0.5'."

A.: "This parameter setting brought performance to the status of 10.2.0.5. The parameter setting "_optimizer_push_pred_cost_based"=false produced the same result. These parameter settings could only have been used as a temporary solution, however, because the first is too hard (this would mean forgoing optimizer features from Oracle 11) and the second can negatively

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		23	00:00:00.07
1	SORT ORDER BY		1	1	23	00:00:00.07
* 2	FILTER		1		23	00:00:00.07
3	NESTED LOOPS OUTER		1	1	1980	00:00:00.04
4	NESTED LOOPS		1	1	1980	00:00:00.03
5	PARTITION RANGE SINGLE		1	1	23	00:00:00.01
* 6	TABLE ACCESS BY LOCAL INDEX ROWID	SERVICE_AGREEMENT	1	1	23	00:00:00.01
* 7	INDEX RANGE SCAN	SERVICE_AGREEMENT_3IX	1	1	36	00:00:00.01
* 8	MAT_VIEW ACCESS BY INDEX ROWID	SOC	23	1	1980	00:00:00.03
* 9	INDEX RANGE SCAN	SOC_99IX	23	18	9811	00:00:00.01
10	MAT_VIEW ACCESS BY INDEX ROWID	PROMOTION_TERMS	1980	1	0	00:00:00.01
* 11	INDEX UNIQUE SCAN	PROMOTION_TERMS_PK	1980	1	0	00:00:00.01
12	SORT AGGREGATE		23	1	23	00:00:00.01
13	PARTITION RANGE SINGLE		23	1	23	00:00:00.01
* 14	TABLE ACCESS BY LOCAL INDEX ROWID	SERVICE_AGREEMENT	23	1	23	00:00:00.01
* 15	INDEX RANGE SCAN	SERVICE_AGREEMENT_PK	23	1	36	00:00:00.01
16	SORT AGGREGATE		18	1	18	00:00:00.03
* 17	MAT_VIEW ACCESS BY INDEX ROWID	SOC	18	1	1398	00:00:00.03
* 18	INDEX RANGE SCAN	SOC_PK	18	1	7281	00:00:00.01

```
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(("SA"."SOC_SEQ_NO"= AND
DECODE(RTRIM("SOC"."TARGET_TARIFF"),'VF_FUN_10',0,'VF_FUN',1,'ALLTO',2)=))
   6 - filter(("SA"."EXPIRATION_DATE"<=TO_DATE(' 2015-01-02 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
AND INTERNAL_FUNCTION("SA"."TARIFF_OPTION")))
   7 - access("SA"."SUBSCRIBER_NO"=' XXXXXXXXXXXX' AND "SA"."BAN"=116)
   8 - filter((TO_CHAR(INTERNAL_FUNCTION("SOC"."EFFECTIVE_DATE"),'YYYYMMDD')<='20150102' AND
NVL("SOC"."EXPIRATION_DATE",TO_DATE(' 4700-12-31 00:00:00', 'syyyy-mm-dd
              hh24:mi:ss'))>TO_DATE(' 2015-01-02 00:00:00', 'syyyy-mm-dd hh24:mi:ss')))
   9 - access("SA"."SOC"="SOC"."SOC")
  11 - access("PRMT"."SOC"="SOC"."SOC" AND "PRMT"."TARGET_TARIFF"="SOC"."TARGET_TARIFF" AND
"PRMT"."EFFECTIVE_DATE"="SOC"."EFFECTIVE_DATE")
       filter("PRMT"."SOC"="SOC"."SOC")
  14 - filter("SA2"."EXPIRATION_DATE"=:B1)
  15 - access("SA2"."BAN"=:B1 AND "SA2"."SUBSCRIBER_NO"=:B2 AND "SA2"."SOC"=:B3)
  17 - filter(NVL("S"."EXPIRATION_DATE",TO_DATE(' 4700-12-31 00:00:00', 'syyyy-mm-dd
hh24:mi:ss'))>TO_DATE(' 2015-01-02 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
  18 - access("S"."SOC"=:B1)
       filter(TO_CHAR(INTERNAL_FUNCTION("S"."EFFECTIVE_DATE"),'YYYYMMDD')<='20150102')
```

Fig. A.2 The execution plan before the Oracle migration

influence other execution plans. For this reason, I requested an execution plan from 10.2.0.5, which looked like this (Fig. A.2):

Do you see the difference between these two plans, Peter?"

P.: "In the second plan, subqueries are processed as subqueries. But I don't understand why the first subquery was only executed 23 times, at least

according to execution plan step 12. It should have been executed as many times as the cardinality in step 3, i.e., 1980 times."

A.: "In step 5, 23 rows from table SERVICE_AGREEMENT (alias SA) were found. In the next step, the cardinality increased to 1980 after the join with the table SOC. The subsequent outer join did not change the number of hits. What is important for us is the fact that this number of hits only contains 23, or even fewer, rows with various values from SA.SUBSCRIBER_NO, SA.BAN, SA.EXPIRATION_DATE, SA.SOC. Precisely, these values are queried in the first subquery (see steps 14 and 15). I assume that Oracle is using a kind of subquery caching here and has thereby optimized the number of executions of this subquery or reduced them to 23. In [2], the author refers to this as filter optimization. We will discuss this optimization later. I have noticed another peculiarity in the second subquery."

P.: "What's that? I don't notice anything."

A.: "In the execution plan in Fig. A.1, the view VW_SQ_2, which corresponds to the second subquery, was executed 1980 times (see step 13). Only 23 rows were found there."

P.: "Is this subquery so selective?"

A.: "I assumed that the reason would be different. The function DECODE, the result of which is compared with the subquery in the SQL statement, returns a null value if the value of the column TARGET_TARIFF differs from 0,1,2."

P.: "I don't quite understand that."

A.: "An 'ELSE' condition is missing in this function. This simple example shows that:

```
SQL> select nvl(decode(dummy,'A','a','B','b'),'NULL') from dual;

NVL(
----
NULL
```

I assumed (and a direct check confirmed this) that the result of the function DECODE consisted mainly of null values. In the case of a null value, one can spare the execution of the subquery because an equality condition for a null value is always wrong. Oracle uses such optimization for subqueries. There is also a similar optimization for joins. We will discuss these optimizations (let's call them filter 'IS NOT NULL') in detail later."

P.: "Why didn't Oracle use this optimization in the first execution plan, i.e., in the case of nested loop join?"

A.: "That's a fair question. At the moment we do not know enough to be able to answer this question. First we have to examine the optimization in detail. We will do that and then we'll give you an answer."

P.: "In what way did the fact that the DECODE function mostly returns null values actually help you?"

A.: “I extended the SQL statement by adding a condition, requesting that the result of the DECODE function should be ‘not null’:

```
SELECT
    FROM    SERVICE_AGREEMENT SA,SOC ,PROMOTION_TERMS PRMT
    WHERE   SA.BAN = 116
    AND     SA.SUBSCRIBER_NO = ' XXXXXXXXXXXX'
    AND     SA.EXPIRATION_DATE <= TO_DATE('20150102', 'YYYYMMDD')
    AND     SA.SOC_SEQ_NO = (SELECT --+ index(SA2 SERVICE_AGREEMENT_PK)
                 MAX(SA2.SOC_SEQ_NO)
                 FROM   SERVICE_AGREEMENT SA2
                 WHERE  SA2.BAN = SA.BAN
                 AND    SA2.SUBSCRIBER_NO = SA.SUBSCRIBER_NO
                 AND    SA2.SOC = SA.SOC
                 AND    SA2.EXPIRATION_DATE = SA.EXPIRATION_DATE)
    AND     SA.SOC = SOC.SOC
    AND     SA.TARIFF_OPTION IN ('VF_FUN_10', '000000000')
    AND     DECODE(RTRIM(SOC.TARGET_TARIFF),
           RTRIM('VF_FUN_10'), 0,
           RTRIM('VF_FUN'), 1,
           'ALLTO', 2) = (SELECT --+ index(S SOC_PK)
                  MIN(DECODE(RTRIM(S.TARGET_TARIFF),
                         RTRIM('VF_FUN_10'), 0,
                         RTRIM('VF_FUN'), 1,
                         'ALLTO', 2))
                  FROM  SOC S
                  WHERE S.SOC = SOC.SOC
                  AND  '20150102' >= TO_CHAR(S.EFFECTIVE_DATE, 'YYYYMMDD')
                  AND  TO_DATE('20150102', 'YYYYMMDD') < NVL(S.EXPIRATION_DATE,
TO_DATE('47001231', 'YYYYMMDD')))
    AND  '20150102' >= TO_CHAR(SOC.EFFECTIVE_DATE, 'YYYYMMDD')
    AND  TO_DATE('20150102', 'YYYYMMDD') < NVL(SOC.EXPIRATION_DATE, TO_DATE('47001231',
'YYYYMMDD'))
    AND  PRMT.SOC(+) = SOC.SOC
    AND  PRMT.SOC(+) = SOC.SOC
    AND  PRMT.TARGET_TARIFF(+) = SOC.TARGET_TARIFF
    AND  PRMT.EFFECTIVE_DATE(+) = SOC.EFFECTIVE_DATE
AND DECODE(RTRIM(SOC.TARGET_TARIFF),
           RTRIM('VF_FUN_10'), 0,
           RTRIM('VF_FUN'), 1,
           'ALLTO', 2) is not null
  ORDER BY DECODE(SA.SERVICE_TYPE,
            'P', 1,
            'M', 2, 3),
    SA.EFFECTIVE_DATE;
```

The relevant execution plan then became almost exactly as fast as the ‘good’ plan in Fig. A.2 (Fig. A.3).”

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		23	00:00:00.08
1	SORT ORDER BY		1	1	23	00:00:00.08
2	NESTED LOOPS		1	1	23	00:00:00.08
3	NESTED LOOPS		1	1	23	00:00:00.08
4	NESTED LOOPS OUTER		1	1	25	00:00:00.03
5	NESTED LOOPS		1	1	25	00:00:00.03
6	PARTITION RANGE SINGLE		1	1	23	00:00:00.01
* 7	TABLE ACCESS BY LOCAL INDEX ROWID	SERVICE_AGREEMENT	1	1	23	00:00:00.01
* 8	INDEX RANGE SCAN	SERVICE_AGREEMENT_3IX	1	1	36	00:00:00.01
* 9	MAT_VIEW ACCESS BY INDEX ROWID	SOC	23	1	25	00:00:00.03
* 10	INDEX RANGE SCAN	SOC_99IX	23	18	9811	00:00:00.01
11	MAT_VIEW ACCESS BY INDEX ROWID	PROMOTION_TERMS	25	1	0	00:00:00.01
* 12	INDEX UNIQUE SCAN	PROMOTION_TERMS_PK	25	1	0	00:00:00.01
* 13	VIEW PUSHED PREDICATE	VW_SQ_2	25	1	23	00:00:00.05
* 14	FILTER		25		25	00:00:00.05
15	SORT AGGREGATE		25	1	25	00:00:00.05
* 16	MAT_VIEW ACCESS BY INDEX ROWID	SOC	25	1	2178	00:00:00.05
* 17	INDEX RANGE SCAN	SOC_PK	25	1	11065	00:00:00.02
* 18	VIEW PUSHED PREDICATE	VW_SQ_1	23	1	23	00:00:00.01
* 19	FILTER		23		23	00:00:00.01
20	SORT AGGREGATE		23	1	23	00:00:00.01
* 21	FILTER		23		23	00:00:00.01
22	PARTITION RANGE SINGLE		23	1	23	00:00:00.01
* 23	TABLE ACCESS BY LOCAL INDEX ROWID	SERVICE_AGREEMENT	23	1	23	00:00:00.01
* 24	INDEX RANGE SCAN	SERVICE_AGREEMENT_PK	23	1	36	00:00:00.01

Fig. A.3 An additional condition "is not null" improved the execution plan

P.: "Was this solution implemented?"

A.: "No. There were too many SQL statements of this kind, which would have necessitated a major code change."

P.: "How was this problem solved then?"

A.: "Tracing with the event 10053 showed that Oracle didn't even try to transform the subqueries into inline views in 10.2.0.5 and to push the relevant join predicates into these inline views. This gave me the idea of looking to see what had been changed in Oracle 11 at this point. I very quickly discovered that Oracle can only push join predicates into the subqueries with DISTINCT, GROUP BY, etc. from 11.1.0.6 onward. As our two subqueries contain aggregations (MIN and MAX functions), this feature should also be applied to them. I immediately verified this with the parameter setting "_optimizer_extend_jppd_view_types"=false, which disables this Oracle feature. Oracle generated the old plan from 10.2.0.5 with this parameter setting. It was precisely this workaround which was implemented."

P.: "You said that this example showed that the formal method was limited. Where does it show it?"

A.: "Let's return to the suboptimal execution plan in Fig. A.1. Formal SQL tuning helps us to identify the problematical execution plan step 17 with the highest cardinality. This cardinality is so high because this step is executed 1980 times. This causes the cardinality of 1980 in step 4. The formal method does not tell us how we can improve the runtime of this SQL statement from 7 to 0.07 seconds. At best you notice that the hits in step 4 can be reduced by an additional condition 'is not null.'"

P.: "Where do you see a limitation here?"

A.: "Formal tuning tries to improve the existing execution plan. It doesn't consider any transformations and optimizations which could lead to a completely different execution plan. It should be noted that it is problematical to extend formal tuning in this way here. But I also have to say that the cases in which transformations and optimizations play a decisive role for tuning are relatively rare in practice. I think we can now examine the two optimizations subquery caching and filter 'IS NOT NULL' in greater depth. For this purpose, we use the script test_case_subquery_caching_filter_is_not_null.sql, which you can download from the website www.tutool.de/book, Peter. There you can also find all the test cases relating to [1]. In this test case, three tables, T1, T2, and T3, are created and filled with data. No column statistics are generated. The first test demonstrates subquery caching (Fig. A.4).

```
select count(*) from t1 where decode(t1.b,1,-1,2,-2,-100) = (select /*+
no_unnest */ max(decode(t2.b,10,-1,15,-2)) from t2 where t2.a=t1.a) or
t1.c =  (select /*+ no_unnest index(t3) */ max(t3.b) from t3 where
t3.a=t1.c)

Plan hash value: 1147310957
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		1	00:00:00.03
1	SORT AGGREGATE		1	1	1	00:00:00.03
* 2	FILTER		1		440	00:00:00.03
3	TABLE ACCESS FULL	T1	1	1000	1000	00:00:00.01
4	SORT AGGREGATE		70	1	70	00:00:00.01
5	TABLE ACCESS BY INDEX ROWID	T3	70	33	1000	00:00:00.01
* 6	INDEX RANGE SCAN	I_T3	70	33	1000	00:00:00.01
7	SORT AGGREGATE		20	1	20	00:00:00.01
8	TABLE ACCESS BY INDEX ROWID	T2	20	25	500	00:00:00.01
* 9	INDEX RANGE SCAN	I_T2	20	25	500	00:00:00.01

Fig. A.4 Subquery caching (filter optimization)

You have no doubt noticed that the function decode(t1.b,1,-1,2,-2,-100) doesn't return any null values. I have done that especially so that only an optimization (namely, subquery caching) is used. Can you explain why the subqueries were executed 70 and 20 times, respectively, although 1000 rows were found in step 3?"

P.: "Both subqueries correlate to the main query. I assume that table T1 has only 70 different values in column C and 20 different values in column A. Oracle stores the results of these two subqueries for the different values of columns C and A in the subquery cache. The subquery is only executed when the relevant result is missing in the subquery cache."

A.: "Your assumption is correct:

```
SQL> select count(distinct c) from t1;

COUNT(DISTINCTC)
----------------
              70

SQL> select count(distinct a) from t1;

COUNT(DISTINCTA)
----------------
              20
```

I must say, however, that Oracle uses a hash algorithm for subquery caching (see [2]). When hash collisions occur, the relevant subquery is executed more often. The parameter '_query_execution_cache_max_size' determines the size of the subquery cache. Let's deactivate subquery caching with the parameter setting "_query_execution_cache_max_size"=0:

```
SQL> alter session set "_query_execution_cache_max_size"=0;

Session altered.
```

Then we will execute the SQL statement again (Fig. A.5)."

```
select count(*) from t1 where decode(t1.b,1,-1,2,-2,-100) = (select /*+
no_unnest index(t2) */ max(decode(t2.b,10,-1,15,-2)) from t2 where
t2.a=t1.a) or t1.c =  (select /*+ no_unnest index(t3) */ max(t3.b) from
t3 where t3.a=t1.c)

Plan hash value: 1147310957
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		1	00:00:00.46
1	SORT AGGREGATE		1	1	1	00:00:00.46
* 2	FILTER		1		440	00:00:00.46
3	TABLE ACCESS FULL	T1	1	1000	1000	00:00:00.01
4	SORT AGGREGATE		1000	1	1000	00:00:00.26
5	TABLE ACCESS BY INDEX ROWID	T3	1000	33	14670	00:00:00.21
* 6	INDEX RANGE SCAN	I_T3	1000	33	14670	00:00:00.11
7	SORT AGGREGATE		560	1	560	00:00:00.18
8	TABLE ACCESS BY INDEX ROWID	T2	560	25	14000	00:00:00.14
* 9	INDEX RANGE SCAN	I_T2	560	25	14000	00:00:00.05

Fig. A.5 Subquery caching (filter optimization) is deactivated

P.: "I don't understand why there is a difference in the frequency with which subqueries are executed."

A.: "Unlike our practical example, the operator OR is between the subqueries here. It therefore makes sense only to execute the second subquery for those rows for which the first subquery does not provide any result."

```
SQL> select count(*) from t1 where not exists (select /*+ no_unnest index(t3) */ * from t3 where
t3.a=t1.c);

  COUNT(*)
----------

       560
```

P.: "Another small optimization!"

A.: "Now let's examine the optimization filter 'IS NOT NULL.' To do this we change the function DECODE, so that it returns null values (Fig. A.6)."

```
select count(*) from t1 where decode(t1.b,1,-1,2,-2) = (select /*+
no_unnest index(t2) */ max(decode(t2.b,1,-1,2,-2)) from t2 where
t2.a=t1.a) or t1.c = (select /*+ no_unnest index(t3) */ max(t3.b) from
t3 where t3.a=t1.c)

Plan hash value: 1147310957
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		1	00:00:00.20
1	SORT AGGREGATE		1	1	1	00:00:00.20
* 2	FILTER		1		452	00:00:00.20
3	TABLE ACCESS FULL	T1	1	1000	1000	00:00:00.01
4	SORT AGGREGATE		1000	1	1000	00:00:00.18
5	TABLE ACCESS BY INDEX ROWID	T3	1000	33	14670	00:00:00.13
* 6	INDEX RANGE SCAN	I_T3	1000	33	14670	00:00:00.05
7	SORT AGGREGATE		24	1	24	00:00:00.01
8	TABLE ACCESS BY INDEX ROWID	T2	24	25	600	00:00:00.01
* 9	INDEX RANGE SCAN	I_T2	24	25	600	00:00:00.01

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(("T1"."C"= OR DECODE("T1"."B",1,(-1),2,(-2))=))
   6 - access("T3"."A"=:B1)
   9 - access("T2"."A"=:B1)
```

Fig. A.6 Optimization FILTER ‘IS NOT NULL’ for subquery

P.: “With the best will in the world, I don’t understand where the figure 24 comes from.”

A.: “That’s very easy. Subquery caching is deactivated. Then Oracle only has to use the optimization filter ‘IS NOT NULL.’ This means that the subquery from the table T2 is executed for all rows for which the subquery from table T3 does not return a result and for which the function decode(t1.b,1,-1,2,-2) is ‘not null’:”

```
SQL> select count(*) from t1 where not exists (select /*+ no_unnest index(t3) */ * from t3 where
t3.a=t1.c) and decode(t1.b,1,-1,2,-2) is not null;

  COUNT(*)
----------
        24
```

P.: “I should have guessed that myself. I notice that although the filter ‘IS NOT NULL’ is used, it does not appear in the predicates. This makes the analysis more difficult.”

A.: “That’s right. The important thing for us is that Oracle always uses the optimization filter ‘IS NOT NULL’ for subqueries even if the relevant tables have no optimizer statistics (as in our test case). Now let’s consider the optimization filter ‘IS NOT NULL’ for joins. To do this, we force Oracle to transform the subqueries into inline views. In the test case, we use outlines so that Oracle always generates the same execution plan (in this way, it is easier to compare the test results). As the SQL text with the outlines is very big, only the relevant execution plan is presented here (Fig. A.7).”

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		1	00:00:00.17
1	SORT AGGREGATE		1	1	1	00:00:00.17
2	NESTED LOOPS		1	1	8	00:00:00.17
3	NESTED LOOPS		1	6	440	00:00:00.02
4	VIEW	VW_SQ_2	1	30	30	00:00:00.01
5	HASH GROUP BY		1	30	30	00:00:00.01
6	TABLE ACCESS FULL	T3	1	1000	1000	00:00:00.01
* 7	TABLE ACCESS FULL	T1	30	1	440	00:00:00.01
* 8	VIEW PUSHED PREDICATE	VW_SQ_1	440	1	8	00:00:00.15
* 9	FILTER		440		440	00:00:00.15
10	SORT AGGREGATE		440	1	440	00:00:00.14
11	TABLE ACCESS BY INDEX ROWID	T2	440	25	11000	00:00:00.11
* 12	INDEX RANGE SCAN	I_T2	440	25	11000	00:00:00.04

```
Predicate Information (identified by operation id):
---------------------------------------------------

   7 - filter(("T1"."C"="MAX(T3.B)" AND "ITEM_2"="T1"."C"))
   8 - filter("MAX(DECODE(T2.B,1,-1,2,-2))"=DECODE("T1"."B",1,(-1),2,(-2)))
   9 - filter(COUNT(*)>0)
  12 - access("T2"."A"="T1"."A")
```

Fig. A.7 Optimization FILTER 'IS NOT NULL' for joins is not used

P.: "I don't see any sign of this optimization."

A.: "Exactly. This happens because Oracle initially assesses whether this optimization is worthwhile. It does this by using the column statistics NUM_NULLS. According to [3], the optimization filter 'IS NOT NULL' is used for nested loop join when the proportion of rows with null values in the relevant column is more than 5 %. That, by the way, is the answer to the question you asked previously."

P.: "Which column do you mean?"

A.: "In our case, that is the function DECODE(T1.B,1,(-1),2,(-2)). Let's create the extended statistics for this function:

```
SQL> col ext new_value ext
SQL> select dbms_stats.create_extended_stats(null,'T1','(decode(t1.b,1,-1,2,-2))') ext from dual;

EXT
----------------------------------------------------------------
SYS_STU_GXO4ZZKOWJIU3MQ2G$$69D

SQL> exec dbms_stats.gather_table_stats(user,'T1', method_opt=>'for columns "&ext" size 254',
no_invalidate=>false)

PL/SQL procedure successfully completed.
```

If we execute our SQL statement again, we see that the optimization is having an effect (Fig. A.8)."

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		1	00:00:00.02
1	SORT AGGREGATE		1	1	1	00:00:00.02
2	NESTED LOOPS		1	1	8	00:00:00.02
3	NESTED LOOPS		1	1	16	00:00:00.02
4	VIEW	VW_SQ_2	1	30	30	00:00:00.01
5	HASH GROUP BY		1	30	30	00:00:00.01
6	TABLE ACCESS FULL	T3	1	1000	1000	00:00:00.01
* 7	TABLE ACCESS FULL	T1	30	1	16	00:00:00.01
* 8	VIEW PUSHED PREDICATE	VW_SQ_1	16	1	8	00:00:00.01
* 9	FILTER		16		16	00:00:00.01
10	SORT AGGREGATE		16	1	16	00:00:00.01
11	TABLE ACCESS BY INDEX ROWID	T2	16	25	400	00:00:00.01
* 12	INDEX RANGE SCAN	I_T2	16	25	400	00:00:00.01

```
Predicate Information (identified by operation id):
---------------------------------------------------

   7 - filter((DECODE("B",1,(-1),2,(-2)) IS NOT NULL AND "T1"."C"="MAX(T3.B)" AND
"ITEM_2"="T1"."C"))
   8 - filter(DECODE("B",1,(-1),2,(-2))="MAX(DECODE(T2.B,1,-1,2,-2))")
   9 - filter(COUNT(*)>0)

  12 - access("T2"."A"="T1"."A")
```

Fig. A.8 Optimization FILTER ‘IS NOT NULL’ for joins is used with extended optimizer statistics

P.: “I notice that Oracle generates the predicate ‘DECODE(B,1,(-1),2,(-2)) IS NOT NULL’ for the filter in step 7. So in this way one can recognize the optimization. It’s a pity that this is not the case for subqueries. Would these extended statistics also have solved the performance problem after the Oracle migration?”

A.: “Yes, they would have helped.”

Literature

1. Nossov L (2014) Performance Tuning für Oracle-Datenbanken. Methoden aus der Praxis für die Praxis. Springer Vieweg, Berlin
2. Lewis J (2006) Cost-based Oracle fundamentals. Apress
3. Anokhin A. Unique Oracle stories. Filter IS NOT NULL. https://alexanderanokhin.wordpress.com/2013/11/16/filter-is-not-null/. Accessed 18 Jan 2015
4. Meade K (2014) Oracle SQL performance tuning and optimization. It's all about the Cardinalities. Edition: self edition

© Springer-Verlag Berlin Heidelberg 2016

L. Nossov et al., *Formal SQL Tuning for Oracle Databases*,
DOI 10.1007/978-3-662-50417-8