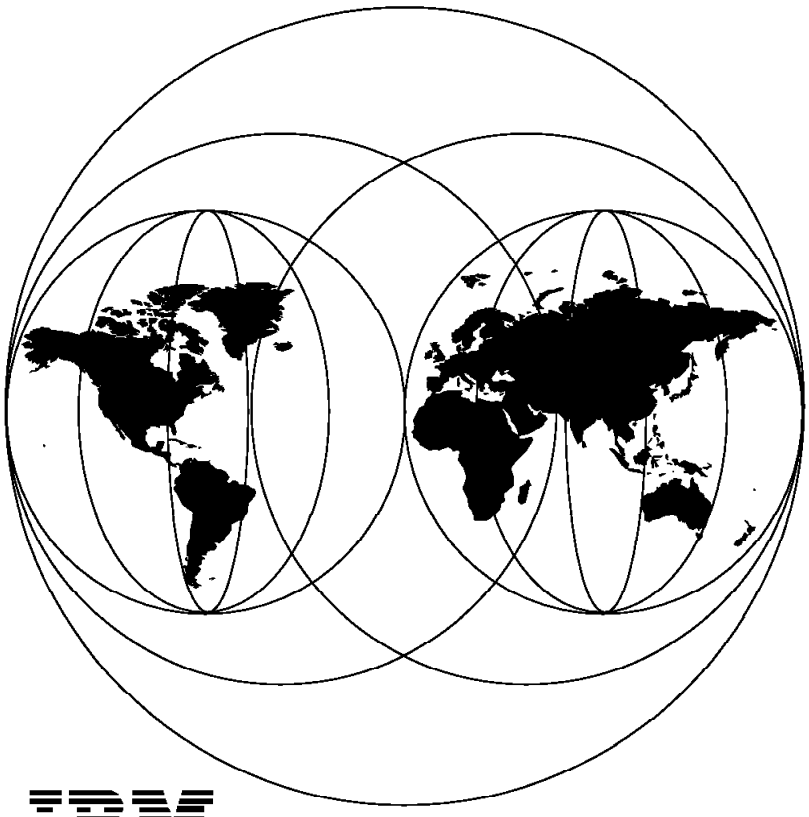
International Technical Support Organization

SG24-2549-00

DB2 - Quick Upper Bound Estimate
An Application Design Methodology

August 1995



IBM

International Technical Support Organization
San Jose Center

International Technical Support Organization

SG24-2549-00

DB2 - Quick Upper Bound Estimate
An Application Design Methodology

August 1995

Take Note!

Before using this information and the product it supports, be sure to read the general information under “Special Notices” on page xiii.

First Edition (August 1995)

This edition applies to Version 3 of DATABASE 2 (DB2) on MVS. Please note that the subject is a design technique that is believed to be fairly release-independent.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSO Technical Bulletin Evaluation Form for reader′s feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. 471 Building 070B
5600 Cottle Road
San Jose, California 95193-0001

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Abstract

This document describes Quick Upper Bound Estimate (QUBE), an application design method in the DB2-MVS environment. QUBE helps application developers make timely and good decisions on performance-related issues. Once the principle of the method is understood, it also can be applied on other DB2 platforms, in particular, the workstation. QUBE can be used throughout a typical application development cycle, thereby providing a common approach for performance topics.

It is assumed that the reader is familiar with application development and – to some extent – the principles and functions of DB2.

(125 pages)

Contents

Figures

Special Notices

This publication is intended to help application developers using DB2 make early and well-founded decisions on performance-related issues by using the Quick Upper Bound Estimate (QUBE) method.

The QUBE formulae for calculating CPU and elapsed time, even when adjusted to a specific environment, must be used only within the context of QUBE. The outcome of any QUBE formula must not be used for any benchmark comparisons or exact prediction of CPU consumption.

The information in this publication is not intended as the specification of any programming interfaces that are provided by DB2 (DATABASE 2). See the PUBLICATIONS section of the IBM Programming Announcement for DATABASE 2 for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM′s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM′s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (VENDOR) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer′s ability to evaluate and integrate them into the customer′s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

CICS
CICS/ESA
CICS/MVS
DATABASE 2
DB2
DB2/2
DB2/6000
DFSMS
DFSMS/MVS
DFSORT
DRDA
ES/3090
ES/4381
ES/9000
ES/9370
IBM
IMS/ESA
MVS/ESA
OS/2
PS/2
QMF
RACF
RMF
VTAM

The following terms are trademarks of other companies:

Windows is a trademark of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

C-bus is a trademark of Corollary, Inc.

Other trademarks are trademarks of their respective companies.

Preface

The building of computer applications is driven by requirements. It is usual to concentrate on the *functional requirements*, because they lead to such concrete elements as screen layouts, online dialogs, and report layouts. *Performance requirements* are also important, but these are often surrounded by a cloud of uncertainties, promises, and highly technical issues, all leading to a point where, in final desperation, one tends to think: *"OK, enough of this, let's start to build, it must be on platform X anyway, so why waste all this time, the system could already have been delivered if not ..., etc., etc."*

So, in effect, performance is often seen as the "behavior" of a system *after* it has been built, something that is more or less unavoidable and a consequence of the specific environment where the system will run. In short, many people feel that performance is *unpredictable*. Is this feeling correct? Are *you* happy with it?

Performance is the *expectation or demand* of the end user (remember, the one who pays the bill). As such, it is as firm a requirement as the functional requirements. All that counts is *"whether the application will satisfy the performance requirements of the end user."* If it will, perhaps even rather easily, you will not have to fall back on your "dirty-tricks" toolbox, and a lot of effort will be saved. No need to deliver a race car when an average family car will do.
If it will not, you will have to look for a design alternative; the extremes are a simple change in the database design or requesting additional hardware.

It is therefore of paramount importance that we can deliver an *early statement on performance* and continue to refine it as we go on with our design. The major cost in applications is data access, which in the present world means the services of a database management system (DBMS). Modern DBMSs also provide many services that traditionally were within the application. These services include data selection, data formatting, and data integrity checking.

In the late 1980s, two system engineers from IBM Finland, Tapio Lahdenmäki and Harri Hakulinen, discovered that may people believed they could neither predict nor influence the performance of the popular relational DBMSs. Being capable system engineers, they knew that this was wrong and recognized that a *simple and attractive method* was needed to make performance predictable, *at the earliest possible stage*. They went on to develop such a method for DB2, IBM's relational database manager on MVS, and called it **Quick Upper Bound Estimate**, or **QUBE**. The method was succesfully introduced in a regular course, which since then has been taught by IBM Education in several countries. This book is intended to be complementary to that course.

Broadly speaking, the following people can use the QUBE method:

1. **Systems analysts**, to make a first assessment of feasibility – whether required response times will be achieved – and inform end users of the expected computer costs
2. **Technical database designers**, to check various alternatives in database design
3. **Technical program designers**, to check and design some common approaches in the application
4. **Programmers**, to check whether various SQL alternatives match the original assumptions of the designer

5. **Performance analysts**, to check what went wrong in a production environment and to report their findings to developers in QUBE terminology to enhance understanding of the problem.

It is outstandingly clear from the above that QUBE can be used by different people, at different stages, almost in the same manner. Therefore, we regard QUBE as a *common method to be used throughout application development*, giving solid ground to otherwise vague and ad hoc decisions.

How This Document Is Organized

The document is organized as follows:

Chapter 1, "The QUBE Approach" introduces the QUBE method, its objectives, prerequisites, terminology, and formulae, and some first examples.

Chapter 2, "Applying QUBE to DB2 Access Paths" demonstrates how QUBE is used for the various access paths in DB2 and explains the access paths themselves.

Chapter 3, "Applying QUBE to Updates" explains how QUBE is used for all forms of update activity.

Chapter 4, "How Accurate Is QUBE?" deals with some situations where QUBE might be (or is known to be) inaccurate. Several alternative approaches are offered for those situations. There are some advanced discussions that you may skip at first reading.

Chapter 5, "Alarm Limits" discusses the actions that can be taken when the first QUBE calculation shows that the performance requirements cannot be satisfied.

Chapter 6, "Implementing the Physical Database Design" demonstrates how to implement the physical database design in a production environment and discusses several decisions to be made, some of which relate to the QUBE approach.

Chapter 7, "Program Coding and Testing" discusses the use of QUBE in the programming stage and considers whether a programmer can also benefit from the QUBE method.

Chapter 8, "QUBE and Accounting Trace Data" shows the various (standard) accounting trace data that can be obtained in a test or production environment and explains how it should be related to the original QUBE estimates. This is an important chapter for developers who are not familiar with accounting data.

Chapter 9, "Closing Topics" covers some DB2 features for which you may want to adjust QUBE, and several other topics, such as relating to QUBE in other DB2 environments.

Appendix A, "The Order-Entry Database" contains specifications of the sample database that is used for most of the examples in this book.

Appendix B, "The QUBE Constants" explains how to adjust the QUBE formula constants to a particular environment.

Appendix C, "Using a Specialist Formula" explains how existing (detailed) formulae can be used to construct QUBE-style formulae.

Appendix D, "Calculating the Buffer Pool Size" shows a statistical approach to calculate the size of a buffer pool.

Appendix E, "The PLAN_TABLE" contains specifications on the use of the DB2 PLAN_TABLE, as it is used in some examples in the book.

Appendix F, "QUBE Worksheet" contains a sample QUBE worksheet, with various common cases and room for you to enter your own installation's values.

Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document.

- *IBM DB2: Administration Guide*, SC26-4888
- *Managing SQL Performance with QUBE*, CF99 (IBM course material)

International Technical Support Organization Publications

- *DB2 V2.3 Nondistributed Performance Topics*, GG24-3823
- *DB2 V3 Performance Topics*, GG24-4284
- *DB2 V2R2 Design Guidelines for High Performance*, GG24-3383
- *Capacity Planning for DB2 Applications*, GG24-3512

A complete list of International Technical Support Organization publications, with a brief description of each, may be found in:

International Technical Support Organization Bibliography of Redbooks, GG24-3070.

To get a catalog of ITSO technical publications (known as "redbooks"), VNET users may type:

```
TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG
```

How to Order ITSO Redbooks

IBM employees in the USA may order ITSO books and CD-ROMs using PUBORDER. Customers in the USA may order by calling 1-800-879-2755 or by faxing 1-800-284-4721. Visa and Master Cards are accepted. Outside the USA, customers should contact their local IBM office.

Customers may order hardcopy ITSO books individually or in customized sets, called GBOFs, which relate to specific functions of interest. IBM employees and customers may also order ITSO books in online format on CD-ROM collections, which contain redbooks on a variety of products.

Acknowledgments

The need for this publication originated within IBM Education circles. The project started as a personal effort of the author and was adopted by the International Technical Support Organization, San Jose Center.

The author of this document is:

Aad Goosen
IBM Netherlands

Thanks to the following people for the invaluable advice and guidance provided in the writing of this document:

Werner van Ipenburg, Rabobank Nederland (Netherlands)

Tapio Lahdenmäki, IBM Finland

Thanks to the following people for their general advice and assistance in the production of this document:

Ravi Kumar, ITSO San Jose
Viviane Anavi-Chaput, IBM France
Maggie Cutler, Editor

Chapter 1. The QUBE Approach

In this chapter we present the following topics:

- Surveying the scene – what is it all about?
- Objectives of the QUBE method – what can you expect?
- Prerequisites – what do you need to get started?
- Definitions and terminology
- Formulae for CPU and elapsed time.

After you have read this chapter you should have a good picture of the QUBE method, its relative ease of use, potential benefits, and limitations.

1.1 Surveying the Scene

The technical design of an application commonly starts with business processes and a logical data model. The logical data model will be translated into a physical database. This database must be capable of serving all processes, and from a functional point of view, it certainly will. What about performance? Online users will have certain expectations, if not demands, about response times. The computer center, with its already overloaded batch schedules, would certainly not appreciate a couple of new monsters running for hours and consuming virtually all resources. Can we do something about this? Or must we just build and hope for the best?

What we would like is a simple check-off approach that any developer could easily use to test if performance requirements are (likely to be) met. It must be a simple method, because most developers are naturally more concerned about function and have neither the time nor desire to become involved in lengthy technical discussions.

If you had such an approach, you could avoid a lot of problems, adjusting things *afterward* is always much more costly than *up front*, that is, before one line of code or CREATE statement has been written.

QUBE is this approach.

1.2 Objectives

QUBE (Quick Upper Bound Estimate) provides a **simple method** that you can use as early as possible in the design phase to establish:

- Whether a requested performance can be achieved with a certain database and transaction design
- Which of several alternatives, in database or transaction design, will provide better performance
- The order of magnitude of computer costs and cost components (giving valuable insight in its own right and a first idea of possible charge-back costs).

The QUBE method is simple and thus it:

- Does not require advanced DB2 knowledge
- Is fairly release-independent
- Can be understood and used by anyone working in application development

- Does not require input that is far beyond what is normally available in the design phase
- Assumes a more or less stable and “reasonable” DB2 production environment

After using QUBE, some situations may remain unresolved, requiring the assistance of a specialist.

By using QUBE, you will get a simplified view of elapsed time, ideally just the sum of CPU and I/O components. This simplified view is enough to make common design decisions and discuss any proposed alternative in a methodical way.

QUBE provides a *mental picture* of CPU and elapsed time in a DB2 environment. This picture – and its terminology – is shared within the project team and can be used throughout the development cycle. Therefore, the use of QUBE is not restricted to the design stage; it also can be used in programming, testing, and monitoring.

The major line of thought for the QUBE formulae is:

1. Elapsed time is a sum of CPU time and I/O time, roughly speaking.
2. CPU time is a function of the number of rows that are processed.
3. I/O time is a function of the number of pages that are read.
4. All of the above timing functions are related to the required application functions and the access strategy. The latter, determined by DB2, can be described in a simplified manner so that any developer can make an easy early estimate. The estimate must be safe.

The output of QUBE is a thoroughly tested physical database and transaction design that will support all known requirements from a performance point of view, and a good picture of the computer costs, per application function. These are very tangible results.

The QUBE formulae are designed to give a conservative, hence “upper bound” estimate.[1] The idea is that you should encounter as few disappointments as possible when your application goes into production; in fact, you should get many pleasant surprises.

For those who expect a fairly accurate estimate we have the following remarks:

- QUBE is not designed for such accuracy, *nor is such accuracy needed in the early stages.*
  What is needed though, is to catch utterly impossible database and program designs, before even one statement is written.
- QUBE can be tailored to your specific environment (for example, hardware, DASD, MVS); this tailoring is a specialist′s job. What you must realize is that the tailoring may contain inaccuracies of its own. In such cases, QUBE will still have full value in terms of a relative estimate.

It must be well understood that QUBE is not:

- A method of calculating expected performance. Although QUBE produces many numbers, expressed in real-life units (milliseconds), they must not be

[1] Not to be confused with “worst case.”

used in their own right. They merely act as triggers, answering the question, Are we still on the right track?
- A capacity planning method. For this, the numbers must be more exact, which they are not.
- A tool. QUBE provides a mental picture, a common framework. The QUBE formulae can of course be put into a ″tool,″ for example, a simple spreadsheet, but QUBE itself is still a method.
- A substitute for DB2 Estimator. QUBE and DB2 Estimator are believed to go very well together, although not necessarily used by the same people or at the same time. The IBM DB2 Estimator is discussed in Chapter 9, “Closing Topics” on page 99.

1.3 Prerequisites

To use the QUBE method you need:

- A little DB2 knowledge
- A physical database design
- Estimates of table sizes
- Estimates of size of results (of the application processes)
- Required response (elapsed) time.

1.3.1 DB2 Knowledge

To understand this document, you are expected to be at least aware of:

- The principles of relational databases
- The principle of *cost-based* optimizing, in particular:
  - DB2 must estimate the number of returned rows, in respect to the number of table rows. This is called *filtering*. The common association is the `WHERE` clause of a `SELECT`, `UPDATE`, or `DELETE` statement.
  - DB2 must estimate the number of pages that are accessed. If there is ″reasonable″ filtering, *clustering* becomes important; it tells DB2 on how many different pages the rows will be found.
  - If there are various access paths, DB2 estimates the costs for those and chooses the path with the lowest (expected) cost.[2]
- The physical level, to the extent that tables and indexes (or rows and index-entries) are stored in *pages*. The page is the smallest unit of I/O operations and an important unit inside DB2. It is (of course) transparent to the application program.
- The functionality of the SQL language.

To understand and apply the QUBE method, you should have reasonable knowledge of different access paths. It will be seen, however, that only a few people require expertise on the full range of possibilities – current and future.

1.3.2 Physical Database Design

By using QUBE, you get a tested physical design. To start with QUBE, you also need a physical design, which is only a first version, called *Version 0*. The first version is always a mechanical transition from the logical data model. Each entity is mapped to a table with a primary key and attributes. Furthermore, you start with indexes on every primary and foreign key; if you have identified *candidate keys* as well, you must design indexes on those too.

[2] See the *IBM DB2: Administration Guide* for details on the alternatives considered.

The logical model does not supply the *clustering order*, that is, the physical order in which DB2 stores the table rows – or at least tries to. For our estimates, clustering is a very important factor, so you need to start with an assumption. The simplest method is to assume that clustering is on the primary key. If, however, the table has one or more foreign keys and your experience suggests that one of those is best, go ahead; if you are wrong, you will soon find out.

1.3.3 Estimates of Table Sizes

You must have some idea of table sizes – information that you can obtain from your users. You will need to know table sizes for other purposes, such as DASD capacity planning.

Try to obtain realistic but not overly pessimistic sizes. Of course, many tables have a highly varying number of rows. For those tables, taking 80% of the peak volume should generally yield a good design.

Once you have obtained the number of rows, QUBE may want an estimate of number of pages and number of leaf-pages (in the indexes). You can calculate the number of pages with the formula given in the DB2 manuals. You will soon see, however, that QUBE does not require that precision. A far easier method is to make a rough assumption, for example, 10 or 20 rows per page, and divide the number of rows by 10 or 20.

Likewise, for the number of leaf pages you can choose between the complex formulae or the simple assumption of something like 200 or 400 entries per leaf page. Choose a single number that reasonably fits your particular environment.

1.3.4 Estimates of Size of Results

This may at times be rather difficult, and you will need the help of the users again. Here are some examples that shows the idea:

- *"Show all orders belonging to a given customer."*
  How many are there? Given that at this point you already know the number of orders (say 100,000) and the number of customers (say 2,000), you may conclude that there will be on average 50 orders per customer. But this assumes that all customers actually have orders, and much as you would like this, it need not be true.
  So the average will be higher; the exact figure depends on business rules such as *"how long do we keep a customer in our system, after the last order-date"*? If you have an existing system, you can make a reasonable estimate; otherwise, it will be something of a guess. Any guess is better than no guess, and furthermore, your guess will be documented and can later be compared with the production database.
- *"Search for a customer with a given name."*
  How many different names do you have?
- *"Delete all orders with a status of 'finished' and an order-date of more than three months ago."*
  This is a month-end process. Suppose that you have 1,000 orders per (working) day; after four months (90 days) of insertions you will have 90,000 orders. If, on average, an order takes two months to become "finished," you will have 45,000 of those in the table, but only one-half of them qualify for the predicate " ... more than three months ago." So you have 22,500 orders.

This conclusion may be arrived at more easily: if the process that you are looking at is the *only* process that deletes rows (once a month), it will delete just as many rows as are inserted per month, or 22,500.

It is a good idea to keep notes of MIN/AVG/MAX occurrences, as designers often do. The QUBE method as we present it here uses the ″average″ case. If there is a very strict performance requirement that must be met ″in all circumstances,″ you must use the MAX numbers.

1.3.5 Required Response (Elapsed) Time

The best way to deal with required response time is to categorize each transaction, query, or batch program as ″small,″ ″medium,″ or ″large″ and assign ranges (alarm limits) to each category (see Figure 1). Note that the highest range is called ″suspect″; normally you do not design for this category, but it may occur during the early calculations. Also note that ″response time″ here means ″response time in DB2.″ The network component is not included.

Type	Size	CPU (ms)	Response (s)
Transaction	Small	< 50	< 0.5
	Medium	50-100	0.5 - 1
	Large	100 - 500	1 - 5
	Suspect	> 500	> 5
Query	Small	< 50	< 1
	Medium	< 200	< 3
	Large	< 1000	< 10
	Suspect	> 1000	> 10
Batch	Small	< 500	< 30
	Medium	< 10000	< 600
	Large	< 50000	< 4 hr
	Suspect	> 50000	> 4 hr

Figure 1. Alarm Limits

These numbers become the alarm limits for the outcome of the QUBE formulae. You should establish the limits before you start to use QUBE; otherwise it becomes too tempting to ″change the requirements after the estimate.″ Of course we do not suggest that *you* would ever do that.

The alarm limits must be established per application and could vary significantly among applications.

For the batch category you could also experiment with the following sizes (in addition to and/or instead of small, medium, and large) and establish the limits accordingly:

- Concurrent with online
- Daily
- Weekly
- Monthly
- Yearly.

1.4 Definitions and Terminology

In order for QUBE to present a simplified view of the various cost components, the terms must be clearly defined.

1.4.1 Touches

Scanning indexes and table rows incur a CPU costs. QUBE assumes that there is a very simple relation between CPU cost and the number of entries processed. This is expressed with the idea of a ″touch.″

Touch

Every time DB2 reads an index entry or a data row, that read is counted as one touch (**T**).

You can also think of our use of the word *reads* as ″processes,″ ″visits,″ ″tests,″ or ... ″touches,″ the most suitable word of all. The touch is directly related to the CPU component of the response time. It is derived from the number of index entries and rows that DB2 must visit in order to obtain the result.

1.4.2 Synchronous Reads

If a single page must be read, DB2 will do that for you. This takes some time, and therefore you must know the number of times that this kind of I/O occurs. QUBE, like DB2 itself, calls this read a ″synchronous read.″

Synchronous read

Every time DB2 reads a single index leaf-page or read a single data page, that read is counted as one synchronous read (**SR**).

This time, *read* is a true read, that is, a physical I/O operation to retrieve a page. The I/O is scheduled on behalf of the application, as soon as it needs the page (but not earlier), hence ″synchronous.″ The number of SRs is a little more difficult to establish than the number of touches, because to do so requires some DB2 knowledge. We will demonstrate, however, that this is in fact very little knowledge.

1.4.3 Sorting

Touches and synchronous reads deal with the basic cost of accessing data. Another (possible) major cost factor is the sorting of data.

Sorting

Every time a process needs sorting we count the number of sort operations (**NS**), the overhead cost, and the number of sorted rows (**NSR**).

The most common reason for sorting is the `ORDER BY` clause of a `SELECT` statement. What you should know for QUBE is that DB2 can avoid a physical sort if a *suitable index* is available (you may in fact design an index for this very reason). So, if you have such an index, you must not count sorting costs.

In the design phase watch out for sorts if the following SQL clauses seem to be needed: `ORDER BY, GROUP BY, DISTINCT, UNION`.[3] The last, `UNION`, always requires a sort; the others may use an index. To be sure, there are other circumstances where a sort is used by DB2, but none of immediate interest.

1.4.4 QUBE Assumptions

In order for QUBE to be simple, we need to make some assumptions:

- The access path as chosen by DB2 is ideal. Thus you can concentrate on what is *logically sufficient* to obtain a certain result. It is DB2's duty, and sometimes that of the programmer as well, to be as intelligent as you are.
- Root and nonleaf index pages are already available in the DB2 buffers (no SR required). Thus the depth of an index plays no part.[4]
- Other pages will have to be read in, but *only once* within the same process (no reread).
- The tables and indexes are well (re)organized. In particular, proper *clustering* is assumed.

In Chapter 4, "How Accurate Is QUBE?" we discuss under which circumstances these assumptions are wrong (optimistic or pessimistic) and need to be adjusted.

1.4.5 Examples

So far we have discussed T, SR, NS, and NSR. These numbers will become input to QUBE formulae, but first we look at some examples that show the QUBE idea. Please refer to our sample database in Appendix A, "The Order-Entry Database," which we use throughout the book.

Note: The examples that follow assume a basic knowledge of index structure, the clustering principle, index scans, and table scans. Complete coverage of access paths is provided in Chapter 2, "Applying QUBE to DB2 Access Paths."

"Retrieve customer data of a given customer (CUSTNO)"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_1	1	1	CUST	1	1

Figure 2. Unique Row Selection

Figure 2 shows a simple access via the unique index, yielding two touches and two SRs. The size of the table is not important.

"Retrieve customer data of CUSTNOs between 3100 and 3200; the expected number of rows is 15."					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_1	15	1	CUST	15	1

Figure 3. Multiple Rows, Clustered Index

[3] In this phase we are not writing SQL statements and should therefore not think in terms of SQL clauses; for the present topic, however, they are the shortest way to express the idea.

[4] Please refer to 2.1, "Index Structure" on page 15 for an explanation of the terminology.

As simple as the example (Figure 3) is, some points require clarification:

- **Why 15 index touches?** Because the structure of the index allows DB2 to find the first qualifying entry quickly and then proceed until the first entry higher than 3200 is found (a *matching index scan*).[5]
- **Why 1 index SR?** Because the 15 entries can safely be assumed to be on one leaf-page. This is not a matter of advanced statistics, but merely being practical.
- **Why 1 table SR?** Because the table is clustered on CUSTNO, so all rows are found on the same page. Or again, they may be on two pages, but does it really matter? The next chapter, where we discuss the DB2 I/O types, will make this clear. In the meantime, it *is* important to note that, if the table were clustered on another column, we would have to count 15 table SRs. Consider Figure 4.

"Retrieve customers with a given ZIPCODE;
the average number of hits is 30."

INDEX	T	SR	TABLE	T	SR
XCU_ZIP	30	1	CUST	30	30

Figure 4. Multiple Rows, Nonclustered Index

Here is the effect of access through a nonclustered index: QUBE assumes that the rows are all on a different page, so we write down 30 table SRs. This is clearly worst case and may seem too pessimistic in some cases, but we strongly suggest that you stick to it. QUBE is meant to be simple. The only case where you should enter a smaller number of SRs is when the table contains fewer pages than "expected hits." In the example, if the CUST table had 20 pages, we would write 20 SRs to satisfy the QUBE assumption *"no reread within the same process."*

"Retrieve customers in a given city;
the average number of hits is 50."

INDEX	T	SR	TABLE	T	SR
			CUST	2000	1

Figure 5. Multiple Rows, Table Scan

If there is no index DB2 must scan the entire table, which is 2,000 rows.[6] In QUBE, this counts as 2,000 touches.

But why only one SR, you may ask? This has to do with *sequential prefetch*, and we kindly ask you to remain patient until the next chapter.

[5] We do not count the sixteenth entry, although indeed DB2 must touch it.

[6] Throughout this book we use the term *table scan* instead of *table space scan*, which is used in the DB2 manuals.

1.4.6 Local Response Time

Figure 6 illustrates the concept of *local response time* (LRT).

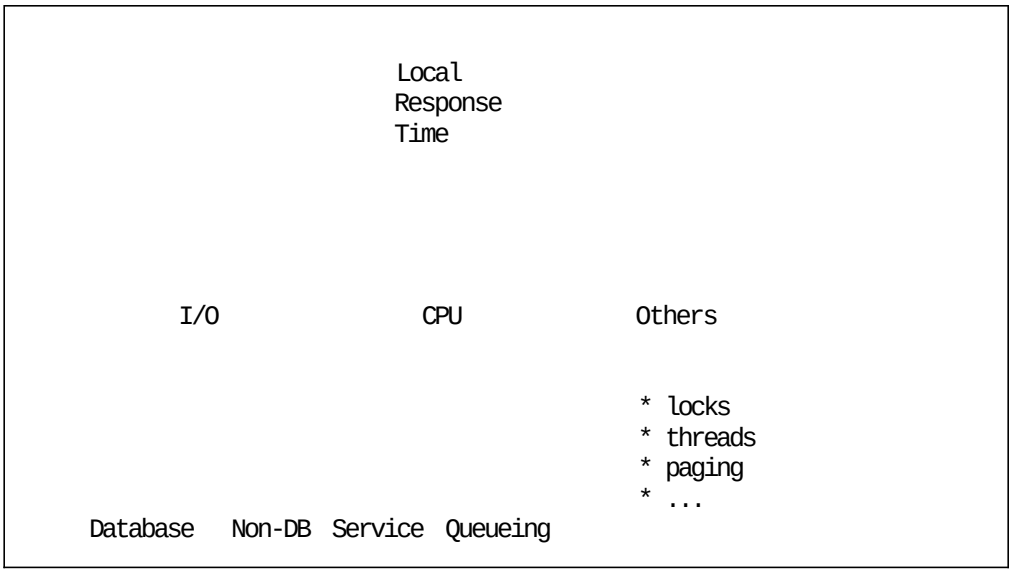


Figure 6. Local Response Time

LRT is basically the sum of I/O and CPU time. These can be calculated with the QUBE formulae. Of course, there is nothing new about calculating I/O and CPU; it has been done for many years. The charm of the QUBE method is that it provides a simple, yet fairly exact prescription for *how* you must calculate them (in a DB2 environment).

As we describe in Chapter 8, “QUBE and Accounting Trace Data” on page 91, the trace facility of DB2 provides similar numbers from the production environment. This gives you excellent feedback on your design phase, thus facilitating your understanding of response time problems.

Figure 6 shows the following components:

- **I/O**
  Database I/O is every DB2 I/O operation to your data, read or write, but only if that affects response time.
  Nondatabase I/O is DB2 logging.
- **CPU**
  The *service time* is the CPU needed for the actual process. The *queueing time* is the waiting that might occur if your process does not get access to a CPU (an MVS decision).
- **Others**
  I/O and CPU are fully covered in QUBE. For completeness, there is always the ″others″ category. After presenting the QUBE formulae we go into more detail about this category, which plays no part in the QUBE approach.

1.4.7 Alarm Limits

The QUBE alarm limits are established per application function, as discussed in 1.3.5, “Required Response (Elapsed) Time” on page 5. With the outcome of the QUBE formulae you can test whether your alarm limit is exceeded. Basically, the process is very simple. Assume that the alarm limit is 1 s:

- LRT = 4 s
  The alarm limit is exceeded, and you must do something. (It is unlikely that LRT will really be 4 sec (in production), but that does not matter).
- LRT = 1.5 s
  Again, you must do something. Because you are closer to the alarm limit, the kind of action that you take may well differ from the first case. (See also Chapter 5, “Alarm Limits.”)
- LRT = 0.8 s
  You are satisfied. QUBE has served its purpose. You might as well forget the outcome of 0.8 s; it will not be used further.

1.4.8 The QUBE Cycle

The overall approach of QUBE is this:

1. Take a single application function.
2. Perform the QUBE calculation.
3. Check the alarm limit:
   a. If not exceeded, take the next function (back to step 1).
   b. Otherwise, make some adjustment and go back to step 2.

In this manner, you gradually refine your database design and establish general programming techniques for certain situations.

The order in which you go through your application functions may vary. Consider a number of choices:

- First the most business-critical functions. These functions may have a significant impact on clustering and degree of normalization. It may be best to establish these things first.
- First the read-only functions. These functions often lead to new indexes, so you will have them in your design before you come to the data-updating functions. If you are new to QUBE, the read-only functions are usually easier to calculate than the updating functions.
- First the major batch updates. These updates often benefit from a specific clustering. High table volumes may have a predominant effect on your choice of clustering order, and you might as well establish the clustering right away.

There is no hard argument for which application functions you consider first; it could be a matter of personal taste, and it could be different per application. As long as you systematically go through all of your functions, the end result should always be the same.

1.5 The QUBE Formulae

Figure 7 shows the QUBE formula for CPU:

$$\textbf{CPU} = M \times (A + (T \times 0.2) + (NS \times 1) + (NSR \times 0.06))$$

Figure 7. QUBE Formula for CPU (ms)

M	machine dependent factor
A	application factor
T	number of touches
NS	number of sort operations
NSR	number of sorted rows

Figure 8 shows the QUBE formula for *local response time*.

$$\textbf{LRT} = CPU + (CPU \times Q) + (SR \times D) + L$$

Figure 8. QUBE Formula for Local Response Time (ms)

CPU	CPU cost according to the first formula
Q	CPU queueing factor
D	DASD elapsed time per I/O (one 4K page)
SR	number of synchronous I/O operations
L	logging costs, 0 if read-only

The constants M, A, Q, D and L should first be determined by a specialist and then used to simplify the formulae (see Figure 9).[7] [8]

$$\textbf{CPU} = 3 + (T \times 0.2) + NS + (NSR \times 0.06)$$

$$\textbf{LRT (CICS)} = (CPU \times 1.5) + (SR \times 20) + 50$$

$$\textbf{LRT (QMF*)} = (CPU \times 3) + (SR \times 20)$$

Figure 9. QUBE Formulae for Designers

The simplification is important from a practical and psychological point of view. Designers just need the formulae, not background details.

[7] Based on a model 9021-711, without the Sort Assist feature.

[8] From hereon, for simplicity we refer to the transaction environment as "CICS," which also means IMS. Note that the CICS formula contains a logging factor of 50 ms.

In Figure 10 and Figure 11 we put the formulae into practice.

"Retrieve customer data of a given customer (CUSTNO)"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_1	1	1	CUST	1	1
CPU = 3 + (2 × 0.2) = 3.4 ms					
LRT(CICS) = (3.4 × 1.5) + (2 × 20) = 45.1 ms					

Figure 10. Single Row Selection, CPU and LRT

"Retrieve all orders of a given customer, sequenced by order date; the average number of hits is 100."					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XOR_2	100	1	ORDER	100	100
NS = 1 NSR = 100					
CPU = 3 + (200 × 0.2) + 1 + (100 × 0.06) = 50 ms					
LRT(QMF) = (50 × 3) + (101 × 20) = 2.17 s					

Figure 11. Multiple Rows with a Sort, CPU and LRT

Note: In these examples we show all fractions, so that you can follow the calculation. In subsequent examples we often use rounded numbers, as you would in real life.

Important warning

All formulae and their outcome must be understood in the right context, which is the QUBE approach. Although the formulae contain machine- and environment-dependent constants, they are – generally – conservative estimates of what might happen in a production environment. QUBE numbers cannot be used for capacity planning. They serve only to test the QUBE alarm limit, thereby either assuring developers that they are on the right track or warning them that they should consider an alternative.

1.5.1 The Formula Constants

We now briefly explain the various formula constants and some related topics. A more detailed discussion can be found in Appendix B, “The QUBE Constants” on page 107.

Machine factor (M)
This factor relates to your hardware environment, specifically the CPU model and the number of processors.

Application factor (A)
This factor relates to certain CPU costs outside DB2. For the time being, consider them as overhead costs and count them as 3 ms per COMMIT; most examples assume one COMMIT.

Queueing factor (Q)
It is sad but true: you are not the only user of the mainframe's CPU resource; you must share it with others. Therefore, depending on *priorities*, *concurrent workload,* and *available processors*, you experience CPU contention. This is what we call the queueing factor.

For the CICS or IMS environment we assume a high priority and therefore a low factor (0.5). In contrast, TSO (for example, QMF) or batch could have a factor of 2, 3, or even higher in peak hours; the examples assume 2.
Please note the definition of *Q* in Figure 8 on page 11: it is what you add to the calculated CPU. In the simplified version, it becomes $CPU \times (1 + Q)$, and hence the examples work with 1.5 or 3, depending on the environment.

DASD I/O elapsed time (D)
We are dealing here with an I/O to retrieve a single page. On many current devices, an estimate of 20 ms is reasonable and safe in the spirit of ″upper bound″. (By the way, the main reason why we use 20 ms in this book is ease of multiplication.)
Your DASD specialist can tell you whether it is significantly more or less (if, for instance, *caching* is used). Your DB2 specialist can tell you whether you can disregard these I/Os entirely (because the buffer pools are large enough).

Logging factor (L)
Any operation that modifies the database is logged. The design of DB2 is such that:

- At commit time the application has to wait for the log write I/Os to complete.
- For a long transaction (batch program), most log I/Os are done asynchronously, while the transaction continues.

If, in addition, the log data sets are placed on fast devices, you will understand why in QUBE the *L* factor is never very important; to make it visible in a trivial transaction, we assume 50 ms in the examples. It should be counted per commit, but only if there was update activity within the commit scope.

Locking factor
A locking condition can certainly affect the response time negatively, but it would be most unusual to ″design″ it to happen. Therefore, it is disregarded in QUBE. In Chapter 9, “Closing Topics,” we discuss locking in more detail.

Thread costs
Before any work can be done a *thread* must be made (or made available). A thread consists of structures in memory that enable your application to communicate with DB2; it is transparent to the application programmer.

The thread creation cost is disregarded. The cost becomes relevant only for extremely trivial transactions, but thanks to features in the CICS-DB2 and IMS-DB2 interfaces, it can be kept low. Furthermore, in such a transaction with one commit, you can safely assume that the thread cost is already included in the *application factor (A)*.

Program load
QUBE does not count program PLAN, PACKAGE, or DBD loads from DASD. In the online environment this is a reasonable assumption. In batch, these loads occur quite often but can be disregarded in relation to the (much higher) overall costs.

Network time
For local online systems, the transfer of screen data is certainly part of the end-user response time, but it is beyond the scope of QUBE. Actually, network time is neither within the definition of our LRT nor in the DB2 accounting data. So if you would include it, you would encounter difficulties in comparing your estimates with DB2 accounting reports.

For remote systems, when you access your data through *distributed database protocols*, such as DRDA, the network elapsed time *is* within the DB2 accounting interval. Estimating those times, however, is still a network specialist's job and therefore not considered in this book.

No doubt, your network specialist also has a QUBE-style formula to estimate the network factor.

Chapter 2. Applying QUBE to DB2 Access Paths

In Chapter 1 we looked at some first examples. We now provide a complete overview of the various access paths, mixed with examples and further refinements for QUBE.

We start with a brief explanation of the index mechanism.

2.1 Index Structure

Figure 12 shows the basic structure of an index.

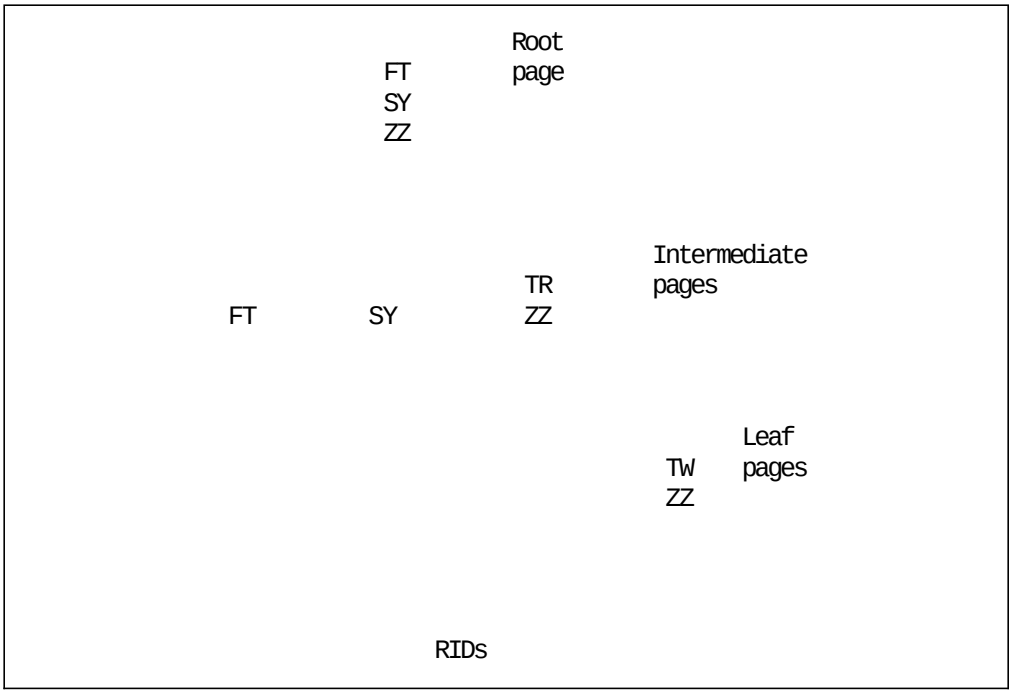


Figure 12. The DB2 Index Structure

The following elements are shown:

Root page The root page contains the cutoff points for lower levels in the index. More precisely, it contains the *highest value* found in a lower-level index page, together with a pointer to that page. In the figure, keys with a value not higher than FT are somewhere in the left-most path, the next range (not higher than SY) in the middle path, and finally the highest range in the right-most path.

Intermediate pages Intermediate pages are similar to the root-page. The root page and intermediate pages together are also called the *nonleaf pages*.

Leaf pages Leaf pages contain all data values and pointers (row identifiers; RIDs) to the data itself. For a unique index, there is one RID per data value; for a nonunique index, there will be more. This is called an *RID chain*.
Leaf pages are logically chained by means of forward pointers.

RIDs An RID is a 4-byte entry pointing to a data page and a reference within that page. From an index point of view, the RIDs point to random data pages. You can however, define one index as *clustering*, which means that DB2 will try to maintain rows in sequence of the index column(s). If that is achieved, successive RIDs point to successive data pages (loosely speaking).

Search path Figure 12 shows the search path for COLX = 'TW'.

In a very small table there is only one index page, which is a leaf page. As more rows get inserted, a second leaf page is required. At this point, a root page must be created. After many inserts (creating new leaf pages) the root page can also become full and will be split into two intermediate pages, and a new root page will be created. At this point we have the three levels as shown in Figure 12.[9]
The number of levels depends on the length of the index key and the number of rows. In practice you often see two- or three-level indexes; if more, the number of rows is usually expressed in millions.

It helps to have the image of an index shown in Figure 13.

ORDERNO	ITEMNO	RID (page)
1000	A001	1
1000	A010	3
1000	B008	7
1100	A002	1
1100	A010	4
1300	A003	2
....	....	.

Figure 13. A Two-Column Unique Index

The key pairs (ORDERNO,ITEMNO) are always in ascending order. As explained above, the page component of the RIDs is best thought to be random.

2.2 DB2 I/Os

All read I/Os in DB2 can be classified as single-page or multiple-page I/Os. Any I/O operation is carried out only if the required data is not already in the *buffer pool*. This is DB2 storage, shared by all users. Buffer pools are discussed in more detail in Chapter 6, “Implementing the Physical Database Design” on page 59.

2.2.1 Single Page I/O

If you look at Figure 12 on page 15 and consider the search for COLX = ¢TW¢, you can imagine that DB2 needs four pages: a root page, an intermediate page, a leaf page, and a data page (not shown). These pages are in principle read with single-page I/Os. This could mean four I/Os, or in QUBE terminology, four SRs. In QUBE, however, for any index access, we disregard all I/O to the nonleaf pages for two reasons:

[9] DB2 Version 4 works slightly differently if you have *type 2* indexes but does not affect QUBE.

- It makes life simple, because the number of index levels becomes unimportant.
- It is also what reasonably can be expected (or demanded) in a production environment. DB2 tries to keep frequently accessed pages (usually a root page or intermediate page) in the buffer pool. To be sure, this is not guaranteed, but it is reasonable.

2.2.2 Multiple Page I/O

Take a table scan, where DB2 reads the entire table, because either all rows are needed or there is no suitable index. If all pages had to be read one by one, the entire process would not be very fast. Fortunately, it is much better than that.
In evaluating the SQL statement, DB2 can see that it will access the data sequentially. Therefore, it generates code that at run time will launch asynchronous read tasks, almost immediately. This is called *sequential prefetch*, or just *prefetch*.

In prefetch, only the first page will be read synchronously. Hence, QUBE counts 1 SR only. If you look back at page 4 you can now understand why we suggested that the number of pages could be roughly estimated. The number of pages makes little difference in QUBE.

As the program continues to fetch rows, the prefetch tasks will continue to be scheduled, so prefetch overlaps with the CPU processing of the program itself (the time it takes to fetch and process a row). Typically, this is always achieved. Exceptions may occur when the program actually fetches very few rows. The most extreme case is something like this:

```
SELECT SUM(AMOUNT) FROM ORDER
```

where, in the absence of an index, DB2 carries out a table scan with prefetch. An accounting report of the program might now show ″Other I/O waits″, indicating that prefetch is not fast enough, compared to the very rapid CPU process.[10]

Prefetch can work on both index and data:

- In an index scan, the leaf pages will be prefetched when it is estimated that eight or more adjacent leaf pages will be needed.
- In a table scan, or data access by means of a clustered index, the data pages will be prefetched when more than one page is involved. You can now understand why, in the discussion following Figure 3 on page 7, it did not matter too much whether the 15 rows were on one page or not. If DB2 thinks that they are on two (or more) pages, it will already use prefetch, and our assumption of 1 SR is valid.

We can summarize QUBE in the design phase as follows: for the above explained conditions, count only 1 SR. Assume that prefetch is always fast enough to ″keep ahead.″

[10] It is for this reason that DB2 Version 3 supports parallel I/O streams; the details are beyond the scope of this book.

2.3 Standard Access Paths

The sections that follow present the basic elements of all access paths in DB2. A good understanding of these standard access paths is fundamental for the use of QUBE.

2.3.1 Table Scan

Figure 14 shows a table scan, the most simple access path of all. DB2 processes the rows sequentially, yielding as many touches as there are rows in the table, but only one synchronous read (because of *sequential prefetch*).

"Print all orders"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
			ORDER	100K	1
CPU = 20 s	LRT (batch) = 60 s				

Figure 14. Table Scan (All Rows)

Figure 15 shows a table scan with few rows. Apparently there is no index on ORDER_DATE.

"Retrieve order data of a given ORDER_DATE" *(300 hits expected)*					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
			ORDER	100K	1
CPU = 20 s	LRT (CICS) = 30 s				

Figure 15. Table Scan (Few Rows)

Something should strike you when you compare Figure 14 with Figure 15: the number of retrieved rows (hits) does not affect CPU at all. And that cannot be right. In QUBE, the outcome for a table scan can be *very pessimistic* if few rows are fetched. Because there is another such case, we postpone the solution for this until the end of this chapter.

2.3.2 Matching Index Scan

In a matching index scan (Figure 16) the search begins at the root page and proceeds until the leaf page is reached. Logically, the search needs to access as many pages as there are levels. As discussed before, in QUBE we assume that the nonleaf pages are already in a DB2 buffer pool.

"Retrieve customer data of a given customer (CUSTNO)"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_1	1	1	CUST	1	1
CPU = 3.4 ms	LRT (CICS) = 45 ms				

Figure 16. Matching Index Scan (One Row)

2.3.3 Nonmatching Index Scan

Figure 17 shows one of the most common reasons for a nonmatching index scan, a condition on the second column of a multicolumn index. If you think about it, you will understand why a search by means of the root is not possible.

"Retrieve order data of a given ORDER_DATE"
(index on CUSTNO,ORDER_DATE, 50 hits expected)

INDEX	T	SR	TABLE	T	SR
XOR_COD	100K	1	ORDER	50	50
CPU = 20 s	LRT (CICS) = 31 s				

Figure 17. Nonmatching Index Scan

Instead, DB2 must access all leaf pages. In itself, this can be an efficient process, because of *sequential prefetch*,[11] but in QUBE you have to count as many touches as there are rows (100K). As with some table scans, this may be *very pessimistic.*

2.3.4 Index-only Scan

An index-only scan (Figure 18) is very efficient because the data pages do not have to be accessed. In QUBE this cost reduction is clearly seen.

"Retrieve ITEMNO's of a given ORDERNO, 4 on average"

INDEX	T	SR	TABLE	T	SR
XOI_1	4	1			
CPU = 4 ms	LRT (CICS) = 26 ms				

Figure 18. Index-only Scan

You can always design for index-only, if you keep some points in mind:

- Indexes can have up to 64 columns, but the total length of the index key cannot exceed 254 bytes.
- For VARCHAR data, DB2 still must access the data pages because the actual length of a particular field is not stored in the index. In fact, because VARCHAR keys are padded with blanks to the maximum, the index may use more DASD than you thought at first.
- There is the usual conflict with updating activity. This confict is clearly seen in the QUBE approach for insert, update, and delete transactions.

[11] Moreover, if the index is properly reorganized, these leaf pages are physically adjacent.

2.3.5 Clustered Index Scan

If DB2 finds 50 index entries (RIDs) for the condition

```
WHERE NAME = ¢Smith¢
```

it is obviously beneficial if the RIDs point to the same data page (or almost). If this is the case, the index is considered *clustered*.[12] The effect is shown in Figure 19.

"Retrieve customers with a given NAME, 50 hits expected"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_NAME	50	1	CUST	50	1
CPU = 23 ms	LRT (CICS) = 75 ms				

Figure 19. Clustered Index Scan

Clustering is a major consideration, for both the DB2 optimizer and you as technical designer. The effect is also most clearly seen in QUBE, when you compare Figure 19 with the example in Figure 20.

2.3.6 Nonclustered Index Scan

"Retrieve customers with a given ZIPCODE, 50 hits expected"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_ZIP	50	1	CUST	50	50
CPU = 23 ms	LRT (CICS) = 1.05 s				

Figure 20. Nonclustered Index Scan

The example in Figure 20 has the same number of hits as in Figure 19, and we see the same amount of CPU time but a dramatic increase in elapsed time, because of the 50 SRs on the table. Note that with the actual sizes in our sample database, 2000 customers on 200 pages, you would be very unlucky indeed to get 50 synchronous reads. At this stage, however, we are not looking at those subtleties, although, in passing, we observe that if our *alarm limit* would happen to be LRT < 1 s, it would be somewhat harsh to consider this limit to be exceeded.

It is essential that you understand what happens here: DB2 reads an index entry and goes to a data page; then it reads the next index entry and goes to another data page, indeed likely to be another, either higher or lower (in page number). And so on. Data pages are accessed purely randomly, and each page is likely to be revisited many times.

Cases like that in Figure 20 have led DB2 designers to come up with a neat solution, *list prefetch*, which works like this: At the beginning of the query DB2 first retrieves *all qualifying RIDs* and sorts them in (data) page order. Then DB2 starts building suitable lists of pages to be read and passes them to VSAM (one I/O operation per list). Thus list prefetch works pretty much like sequential prefetch, but, in practice these lists are much smaller; 5 would be a typical number (32 otherwise).

[12] This is the usual terminology. In fact, it is better to say that the *table is clustered* on the columns of the index. Any index can be *clustered*, but only one is the *clustering index*, the index that DB2 respects for INSERTs and REORGs.

List prefetch is a neat technique, but what does it mean for QUBE? Keeping in mind that QUBE should require *as little DB2 knowledge as possible*, we decided to ignore list prefetch altogether. So Figure 20 remains as it is, 1 SR per touch. This may seem a little too pessimistic, but:

- There is no easy rule to determine whether DB2 will actually choose list prefetch. So it is better to treat list prefetch as a pleasant surprise at run time.
- If we were to consider list prefetch, we might also have to count costs for the *RID sort*. This is not in the QUBE spirit of keeping things simple.
- As the number of pages increases, the lists may become smaller, and list prefetch thus becomes less effective.
- If you hit an alarm limit, the *QUBE specialist* can always consider the effect of list prefetch and give you a new LRT estimate. It might also happen, by the way, that the QUBE specialist spots a case where the *"*pleasant surprise*"* is not pleasant at all.
- QUBE was meant to be conservative.

2.4 Complex Access Paths

Under this heading we present a number of access paths that are more complicated than the standard access paths discussed in 2.3, “Standard Access Paths.” For the most part, however, any complex access path can be decomposed into simple access paths.

2.4.1 Multiple Index Scan

Figure 21 shows a mulitiple index scan for two separate indexes on CUSTNO and ORDER_DATE.

"Retrieve orders with a given CUSTNO and ORDER_DATE"
(separate indexes on these columns)

INDEX	T	SR	TABLE	T	SR
XOR_2	50	1			
XOR_OD	300	1			
			ORDER	2	2
CPU = 73 ms		LRT (CICS) = 190 ms			

Figure 21. Multiple Index Scan (AND-ing)

In the early years of DB2 the optimizer would just pick out one of the two indexes (say XOR_2) and then go to the data rows (50 in the example) to test the other condition. In a worst case condition, there could be 50 SRs.
Accessing another index first is just 1 SR to retrieve a second set of RIDs, which can be matched with the first set; ideally, a much smaller set of RIDs remains (2 in the example).

The *multiple index scan* also works for OR conditions (see Figure 22).

"Retrieve orders with a given CUSTNO or ORDER_DATE"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XOR_2	50	1			
XOR_OD	300	1			
			ORDER	320	320
CPU = 137 ms LRT (CICS) = 6.6 s					

Figure 22. Multiple Index Scan (OR-ing)

The number 320 is an arbitrary value. Note that:

- For index AND-ing, one index would be chosen anyway, but multiple index scan may avoid accessing a large (random) number of pages.
- For index OR-ing, mulitiple index scan is needed to avoid a table scan. In the example you may think that a table scan would not be so bad at all, because you would have 1 SR instead of 320, but you get a much higher number of touches, and 320 SRs is pessimistic anyway.

Multiple index scans internally always use list prefetch. If you review our argument on page 21 to ignore list prefetch, you can see that the first argument (″unpredictability of list prefetch″) does not apply. The other arguments *do* apply, however, so we still propose to ignore it and – as a consequence – write 320 SRs. We note that this is pessimistic, especially with an eye on the total number of pages (5,000). You could also say that if DB2 itself would really think that 320 SRs would be needed, it would surely choose the table scan instead.

2.4.2 Multitable Access Paths

For those who are familiar with SQL it is tempting to categorize multitable access paths as ″joins,″ ″subqueries,″ or ″unions.″ We will not do that, however, because we are still designing and not programming. The examples will show that when more than one table is involved, the cost for any multitable access path is mostly an addition of the costs for two or more simple access paths. You must try to picture which data and (simple) access path is *logically sufficient* to solve the problem. In QUBE-thinking, you must assume that DB2 has a suitable physical access method; this is discussed further in Chapter 7, “Program Coding and Testing.”

There is nothing special in the example in Figure 23; after DB2 has read the row from ORDER, the CUSTNO is known and can be used in an index scan to the CUST table.

"Retrieve order and customer data of a given ORDERNO"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XOR_1	1	1	ORDER	1	1
XCU_1	1	1	CUST	1	1
CPU = 4 ms LRT (CICS) = 86 ms					

Figure 23. Single Result Row, Two Tables

With the given CUSTNO you can obtain the data though indexes (see Figure 24). Note that the shown sequence seems the most logical, but in fact it is a matter of personal taste and first accessing ORDER makes no difference. There is only 1 SR on the ORDER table, because it is clustered on CUSTNO.

"Retrieve order and customer data of a given CUSTNO" *(50 orders per customer)*					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_1	1	1	CUST	1	1
XOR_2	50	1	ORDER	50	1
CPU = 23 ms LRT (CICS) = 115 ms					

Figure 24. Multiple Result Rows, Two Tables

In Figure 25 the XOR_2 index is logically sufficient to test whether a customer has any orders QUBE assumes that DB2 is as intelligent as we, and that programmers know their business.

"Retrieve customers with a given ZIPCODE, *who have no orders, in CUSTNAME order"* *(50 customers per zipcode, final result 20 rows)*					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_ZIP	50	1	CUST	50	50
XOR_2	50	50			
Sort = 1 + (20 × 0.06)					
CPU = 35 ms LRT (CICS) = 2 s					

Figure 25. Test for Existence

For the example in Figure 26, a little technical knowledge might be needed: if an ordered result is built from merging two tables, as is the case here, a final sort step is always needed.

"Show current and historical order data of a given CUSTNO, *in date sequence; the ORDERH table is clustered on DATE"* *(20 current orders, 100 historical orders)*					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XOR_2	20	1	ORDER	20	1
XOH_2	100	1	ORDERH	100	100
Sort = 1 + (120 × 0.06)					
CPU = 59 ms LRT (CICS) = 2.1 s					

Figure 26. Two Nonrelated Tables

2.5 Very Pessimistic Cases

We have seen two important cases where the QUBE method is far too pessimistic: the table scan and the nonmatching index scan, if few rows are retrieved. Why are they very pessimistic (VP)?[13]

If you look again at Figure 15 on page 18 and Figure 17 on page 19 and mentally apply the QUBE formulae, you will quickly see that the major component is the number of touches. And therein lies the explanation: the basic cost for a touch in QUBE (0.2 ms) is based on a ″qualifying touch,″ that is, a touch on a row that will be fetched. Thus the basic cost includes costs to:

- Select the row
- Extract the columns
- Convert from internal to external format
- Pass the columns through internal DB2 layers
- Pass the data to the application (fetch cost).

For a nonqualifying row this is far too much. In fact, DB2 has been designed to disqualify a row as early as possible, so we must in some way reflect that in the QUBE method. What we are looking for is a formula that makes a difference between *qualifying* and *nonqualifying* touches. Such formulae can be found in various publications and are used by specialists. Because the formulae are for specialists, they contain many parameters that must be understood correctly. They also are probably pretty release-dependent and therefore difficult to maintain.

So now it seems that in VP cases we must choose between using a specialist formula or accepting the original outcome, flagging it as VP, and then hope for the best. Neither choice is very attractive, so we have developed an alternative formula (see Appendix C, “Using a Specialist Formula” on page 113 for details). Figure 27 shows the formula we developed.

$$\textbf{CPU} = M \times (A + (T \times TC) + NS + (NSR \times 0.06))\ \text{ms}$$

Figure 27. New QUBE Formula for CPU (adjusted)

The new element *TC* (touch cost) is:

0.2	for normal cases
0.01	for VP table scans and nonmatching index scans.

The VP number must be used only when you are dealing with very few fetches in a nontrivial table.

To apply the new formula to our original cases, first we derive the designer version, still for a model 9021-711 (see Figure 28).

$$\textbf{CPU} = 3 + (T \times 0.01) + NS + (NSR \times 0.06)$$

Figure 28. CPU Formula for Designers, VP Case

[13] The QUBE course developers conceived the phrase ″very pessimistic,″ or VP.

Figure 29 and Figure 30 show significant differences in the outcome of the revised formulae. But, do you believe them all? Look at the LRT for the table scan: 1.5 seconds for retrieving 300 rows out of a 100,000, all of which have to be read from DASD. Sounds pretty good, but in fact it is too good to be true.

"Retrieve order data of a given ORDER_DATE" (index on CUSTNO,ORDER_DATE, 50 hits expected)					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XOR_COD	100K	1	ORDER	50	50
CPU = 20 s	LRT (CICS) = 31 s	(original)			
CPU = 1 s	LRT (CICS) = 2.5 s	(revised)			

Figure 29. Very Pessimistic Nonmatching Index Scan

"Retrieve order data of a given ORDER_DATE" (300 hits expected)					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
			ORDER	100K	1
CPU = 20 s	LRT (CICS) = 30 s	(original)			
CPU = 1 s	LRT (CICS) = 1.5 s	(revised)			

Figure 30. Very Pessimistic Table Scan

The query has become so fast CPU-wise, that we must look again at the I/O component. Suppose that we have 20 rows per page, so 5,000 pages. Even with prefetch, the physical I/O requires a certain time per page. A rough estimate on current devices is 2 ms per page. You are not the only user of DASD, however, so you cannot assume exclusive device-ownership during the entire execution, and it is thus safer to use 3 ms. So the I/O elapsed time can never be lower than 3 * 5,000 = 15 s. And then of course, the LRT also cannot be lower than 15 s.[14]

And so it turns out that the original VP outcome was not terribly pessimistic after all; that is, the CPU was too high, by a factor of 20, but the LRT by a factor of "only" 2.

The same reasoning applies to the nonmatching index scan. But, there are far fewer leaf pages than data pages, so we have to check it: say there are 200 entries per leaf page: 500 pages at 3 ms per page is 1.5 s. This is less than the revised LRT (2.5 s), so there is no adjustment.

Are we now finished with VP? Well, not quite. There are two more points:

- If a substantial number of pages are already in the buffers, prefetch is not required. So above remarks apply to a lesser extent, or not at all. Monitoring this condition can be done with DB2 Accounting reports or with the -DISPLAY BUFFERPOOL command. The relevant items to look at are SEQUENTIAL PREFETCH REQUESTS, PREFETCH I/Os, and PAGES READ.
- To calculate the elapsed time for prefetch, we had to know the number of pages. We assumed 5,000 pages (20 rows per page), which led to an elapsed time of 15 s; if we had had 100 rows per page, the elasped time would have

[14] Disregarding the possibility of parallel I/O and hardware caching.

been something like 3 s. So the number of rows per page becomes rather significant, which it was not before.

Armed with this knowledge we can summarize the admittedly complex theory of VP:

- Table scans and nonmatching index scans, if they retrieve relatively very few rows, are classified as very pessimistic (VP). The word *very* applies to the CPU component in particular.
- In a critical situation, usually an online transaction, a VP case may immediately act as a trigger to look for a design alternative, such as a (better) index.
- You can use the revised QUBE formula to get a more accurate CPU estimate (about 20 times less).
- With the revised formula, you must check the elapsed time for prefetch operations. To do so, you must have a reasonable estimate of the number of pages and multiply that by 3 ms. The outcome will show you whether the query has become I/O bound and, consequently, what the LRT will be.

2.5.1 Nonclustered Index (Revisited)

Let us take an earlier example of a nonclustered index scan – that shown in Figure 31. To the QUBE worksheet we add a solution with a table scan.

"Retrieve customers with a given ZIPCODE, 50 hits expected"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_ZIP	50	1	CUST	50	50
CPU = 23 ms	LRT (CICS) = 1 s				
			CUST	2000	1
CPU = 403 ms	LRT (CICS) = 625 ms	("VP")			
CPU = 23 ms	LRT (CICS) = 600 ms	(revised)			

Figure 31. Index Scan or Table Scan?

For a nonclustered index scan, the point of the example was that in QUBE you must count as many SRs as there are Ts (50). But why did we actually assume an access path though the index? Perhaps on the reasoning that "50 out of 2,000 is a small percentage and therefore an index must be better." But, is it better? And will DB2 really pick the index?

Please look at the QUBE estimates for a table scan: the first is "standard" QUBE, and yields 625 ms thus better than the index scan. The difference lies in the number of SRs, not CPU. And then, to some extent the table scan qualifies as VP, so we could redo the calculation with our revised formula. Now everything is lower still,[15] and we should not be surprised if the DB2 optimizer chooses the table scan.

We cannot test that however, because *we are still only designing*. So what must we do with cases like this? Some observations can be made:

[15] LRT = 600 ms after adjusting for prefetch I/O delays.

- Retrieving rows (in some quantity) by means of a nonclustered index is *never really attractive*. The drawback is the high number of (estimated or actual) synchronous reads.
- DB2 will therefore often decide in favor of a table scan, even when the *filtering* is seemingly good. For instance, in our example only 2.5% of the rows qualify, yet a table scan is likely.
- Both for clustered and nonclustered access, there is a break-even point where the elapsed times for an index scan and a table scan are the same. The break-even point can be expressed as a filtering factor, or as *n* rows. If the number of rows to be retrieved is smaller than *n*, you have an index scan; otherwise a table scan. This point can be calculated for the QUBE approach, and it can be established empirically for DB2 itself (the BIND decision). We leave it as an exercise for you to actually calculate it for QUBE.

  Note: You could do such a calculation for both CPU and LRT. Probably the CPU calculation is best. It is a little dangerous, however, to interpret the outcome as ″Aha, now I know what DB2 is doing,″ because:

  - QUBE is not meant to be accurate
  - There is no exact documentation as to whether DB2 optimizes on CPU or on elapsed time, nor is there documentation on the internal weighing factors for CPU and I/O.
- If you consider adding the index (on ZIPCODE) for this process alone, do so only if the filtering is very good (less than 1% or 2%). Always stay on the alert whether the index is actually used (EXPLAIN).
- You may establish fixed limits (number of pages), above which you do not allow table scans in the transaction environment (rule). If, while designing, you have such a table, plan for nonclustered index access.
  If the cost of such an access slightly exceeds your particular alarm limits, and changing the clustering order is not an option, here are some comforting thoughts:

  - Perhaps there will be more than one qualifying row per page, so fewer SRs.
  - Perhaps, or sometimes rather likely, there will be some *buffer pool hits*, so again fewer SRs.
  - Perhaps there will be a refinement in program design (see Chapter 5, “Alarm Limits”). For our example the idea is that you would not necessarily retrieve 50 rows if you cannot display them on one screen in the first place.

2.6 Summary

For QUBE, the main conclusions and refinements are:

- In any index scan, forget about the nonleaf pages.
- Index-only is a design option for critical read transactions.
- Clustering is a major consideration. If you change clustering, you will have to review a lot of earlier QUBE estimates.
- Ignore list prefetch; accept a little pessimism on SRs.

- Nonmatching index scans and table scans may yield a very pessimistic estimate, if the number of qualifying rows is low. There is an adjusted formula for CPU for these cases.
- Accessing multiple tables is just adding the costs for several single table accesses. Do not even think about internal join methods.

Appendix F, “QUBE Worksheet” on page 121 contains a worksheet that lists the most important access paths together with their Ts, SRs, CPU, and LRT numbers.

Chapter 3. Applying QUBE to Updates

QUBE can be applied to any form of update activity.[16] It uses the same input as for DB2 access paths, touches and synchronous reads, and the formulae for CPU and LRT also remain unchanged.

Some care must be exercised in establishing the Ts and SRs because an update consists of several elements. We need to consider the basic update elements listed in Figure 32.

Type	Operation	Touch
FIND	The ″home″ page for INSERT	1
	in DELETE or UPDATE	n
RI	Check one referential constraint	1
MODIFY	INSERT a table row	1
	Add an index row	1
	DELETE a table row	1
	Delete an index row	1
	UPDATE a table row	1

Figure 32. Basic Update Elements

Furthermore, we need to define some new assumptions:

- No long *RID chains* (see also page 15). In particular when DB2 deletes an index row, a long RID chain has a negative effect on several internal DB2 processes. This does not apply if your index is of TYPE 2, an index type provided by DB2 Version 4.
- Enough free space. It would be difficult to QUBE if you would have to consider extra index or table reads to find free space. Whether enough free space will be available in production is a matter of balance among:
  - Free space parameters
  - REORG frequency
  - Change frequency
  - Change pattern (perhaps a DELETE is compensated for by an INSERT in the same page).

  To achieve this balance some experience is required. In that sense QUBE is helpful, because with its firm assumption of ″enough free space,″ at least you can understand more easily why certain performance problems in production occur.
- Each table has at least one index. Thus there is always a clustering index for insert purposes.
- There are always indexes on primary and foreign keys, regardless of whether the *referential constraints* are enforced by DB2 or by the applications.

[16] The lower-case word *update* always means any SQL operation that modifies data (INSERT, DELETE, or UPDATE, and even most DDL and DCL statements).

In all operations an index is often *logically* accessed more than once; as usual, you must take care not to count more than one SR (no page reread).

3.1 Insert

An insert consists of:

1. **FIND:** the clustering index is used to find the ″ideal page″ (or *home* page). This requires one touch and one SR.
2. **RI:** per foreign key you must count an index-only access to the parent table. Again this requires one touch and one SR.
3. **MODIFY:** To add the data row you count one touch and one SR. Next, for each index on the table, you add one touch, and one SR unless it was counted earlier.

Please observe that you do not include a check for uniqueness; any UNIQUE index is touched anyway in MODIFY.
Let us look at some examples. Figure 33 shows a minimuim single row insert. There are no RI implications (R) and no additional indexes, so this can be seen as a minimuim insert.

″Insert a new customer″					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_1 (*F*)	1	1			
XCU_1 (*M*)	1		CUST	1	1
CPU = 3.6 ms LRT (CICS) = 95 ms Notation: (*F*) is for FIND, (*M*) is for MODIFY.					

Figure 33. Insert Single Row (Minimum)

Note that in this chapter we assume 50 ms for logging elapsed time. Do not get upset by this high number; your installation′s number may well be as low as 5 ms.

Warning

For any minimum case, keep the following in mind:

- It is an upper bound estimate.
- Do **not** use the outcome for reasoning such as: ″one insert is 100 ms, one thousand inserts are 100 s.″
  Quite apart from counting too much application CPU and log I/O, the inserts more often than not are concentrated on certain pages, so the number of SRs is considerably lower.

Figure 34 shows a single row insert with one check for referential integrity.

"Insert a new order"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XOR_2 (*F*)	1	1			
XCU_1 (*R*)	1	1			
XOR_1 (*M*)	1	1	ORDER	1	1
XOR_2 (*M*)	1				
CPU = 4 ms	LRT (CICS) = 136 ms				

Figure 34. Insert Single Row (with RI)

The explanation for Figure 34 is:

1. Find the "ideal" page by scanning the clustering index of the ORDER table.
2. Check the existence of the foreign key, CUSTNO, which is an index-only access of the primary key, in the CUST table.
3. Modify the ORDER table and its primary index.
4. Modify the other index (on CUSTNO).

Figure 35 shows a multiple row insert with two checks for referential integrity.

"Insert 10 orderlines for one order"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XOI_2 (*F*)	10	10			
XOR_1 (*R*)	10	1			
XIT_1 (*R*)	10	10			
XOI_1 (*M*)	10	10	ORDITEM	10	10
XOI_2 (*M*)	10				
CPU = 15 ms	LRT (CICS) = 0.9 s				

Figure 35. Insert Multiple Rows (with RI)

The explanation for Figure 35 is:

1. Find the "ideal" page by scanning the clustering index of the ORDITEM table. Because this is the index on ITEMNO, and we have 10 different ITEMS, it is 10 SRs.
2. Check the existence of the ORDERNO in the ORDER table; it is 10 times the same ORDERNO of course, but we automatically count it as 10 touches.
3. Check the existence of the 10 ITEMNOs in the ITEM table.
4. Modify the ORDITEM table and its indexes. The clustering on ITEMNO again means 10 SRs on the ORDER table.

3.2 Delete

A delete consists of:

1. **FIND:** the cost of finding the rows is determined in the same way as for read operations.
2. **RI:** per foreign key you must count an index only access to the dependent table. In that dependent table, you must count delete costs (if CASCADE is in effect) or update costs (if SET NULL is in effect).

3. **MODIFY:** To delete the data row you count one touch. Next, for each index on the table, you add a touch, and one SR unless it was counted earlier.

Figure 36 shows a single row delete with RI RESTRICT.

"Delete a customer"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_1 (*F*)	1	1			
XOR_2 (*R*)	1	1			
XCU_1 (*M*)	1		CUST	1	1
CPU = 4 ms	LRT (CICS) = 116 ms				

Figure 36. Delete Single Row (with RI RESTRICT)

Figure 37 shows what would change if the delete rule were SET NULL.

"Delete a customer; one customer has 50 orders on average"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_1 (*F*)	1	1			
XOR_2 (*R*)	50	1			
XCU_1 (*M*)	1		CUST	1	1
XOR_2 (*M*)	100	1	ORDER	50	1
CPU = 44 ms	LRT (CICS) = 215 ms				

Figure 37. Delete Single Row (with RI SET NULL)

The difficult things here are the operations on the ORDER table:

- To check where the orders are, you must count 50 Ts and 1 SR on the XOR_2 index; the internal result is 50 RIDs. (Yes, those 50 Ts with their standard *touch cost* are too pessimistic, but never mind).
- Because the ORDER table is clustered on CUSTNO, the 50 RIDs point to the same data page. Or, at least at consecutive pages, and we rely on DB2 to use prefetch; hence, always only one SR.
- The XOR_2 index has to be modified. That is always one index delete and one index insert, per entry, which as 100 Ts. Also, the index page that contains the NULL entries still must be read (1 SR).

Note: In the order-entry database SET NULL is not very realistic. In a different setting though, the example is still worthwhile, with perhaps two remarks:

- In practice, the entry to be deleted often has far fewer dependent rows (say 10) than the average case (50); you must judge whether to adjust for this.
- It is important to define the exact meaning of sentences such as *"customer has 50 orders on average."* If you have 2,000 customers and 100,000 orders, the statement is mathematically correct. If, however, 50% of the customers do not have an order at all, it would be better to say *"*1,000 *customers have orders (100 on average), and 1,000 customers have no orders."* If you apply this idea in our example, it would mean that it is rather likely that you will delete a customer who has no orders. But, in another program, if you would have to display existing orders you would reckon on 100 orders per customer.

3.2.1 The Context of Delete

Now that we have finished with the two simple DELETE examples, we have a new line of thought: Suppose that before the DELETE, the row has just been read. Therefore in practice some of the 5 SRs might well not occur, so the LRT drops. Now what? Must we consider this in QUBE? And to what extent?
Until now all of our QUBE examples have been very simple processes, decomposed almost to the statement level. But now it seems that we might have to look at the context. For our example there are several possibilities:

- The delete is issued ″out of the blue″: *delete custno 43521*. No previous read, so clearly the example as shown in Figure 36.
- The delete comes from a batch program, while processing rows from the CUST table. Technically, there are two forms of such a delete:
  1. Cursor controlled (`.. WHERE CURRENT OF ..`). Clearly there are no FIND costs now, nor an SR for the CUST table. Whether there is an SR for the XCUST index depends on the access path of the cursor.
  2. Direct delete, within a cursor loop. The usual reason for this type of delete is that the statement contains `ORDER BY`, preventing a cursor-controlled delete. It might now become a matter of the access path: it can still be ″direct fetch via an index,″ so in fact it is similar to Figure 36. But it might also be some form of scan, a physical sort and fetches from the workfile. In that case, if the result set is large, the *time between the last access to a page and the delete* could be considerable, and therefore the pages may no longer be available (in the buffers). Following the QUBE axiom ″no reread,″ we would discard this possibility (no extra SRs).
- The delete is issued from a browse list in an online system (perhaps with a ″d″ line command, or by clicking on Delete). Similar to physical sorting, there is some time between the delete and the initial read, although that time now is mostly user think-time and hence is unpredictable. Can we still follow the no reread axiom? It is a matter of whether you consider the browse transaction and the delete transaction as being the same transaction or not. Not technically of course, but logically. We would suggest looking at the transactions in isolation, so to count the SR. If the SR does not occur at run time, fine; only if this extra SR causes our alarm limit to be reached would we agree to make a side note, ″SR unlikely,″ and continue.

We conclude that there are cases where there is no context (″out of the blue″) and cases where there is a clear context (usually batch). No problem for either. The inevitable in-between cases are often online situations with user think-time; we suggest that you take the safe approach (count extra SRs).

3.3 Update

An update consists of:

1. **FIND:** the cost of finding the rows is determined in the same way as for read operations.
2. **RI:** per changed foreign key you must count an index-only access to the parent table (as for inserts).
3. **MODIFY:** To update the data row you count one touch. Next, for each index that is changed, you add two touches, and two SRs unless they were counted earlier.

In Figure 38, showing a simple update, it is a matter of style whether to write one or two lines in the QUBE worksheet; we prefer two, FIND and MODIFY.

"Update a customer's phone-number"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_1 (*F*)	1	1			
			CUST	1	1
CPU = 3.4 ms	LRT (CICS) = 95 ms				

Figure 38. Update Single Row (Minimum)

The first line in Figure 39 is to find the index entries with the old ZIPCODE. Because the table it is not clustered on ZIPCODE, you must count 20 SRs.
The last line reflects the index updates. Just follow the rule of counting 2 touches per single update. The index page with the new ZIPCODE also has to be read.

"A certain ZIPCODE has become obsolete and must be replaced" *(20 occurrences)*					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_ZIP (*F*)	20	1			
			CUST	20	20
XCU_ZIP (*M*)	40	1			
CPU = 199 ms	LRT (CICS) = 0.8 s				

Figure 39. Update Multiple Rows, Including Index

3.4 Affordable Indexes

Having discussed the three forms of update, it has become clear that the number of indexes per table is important. For INSERT or DELETE the number of indexes always affects the LRT; for UPDATE, only if the index column is changed.

Indexes are usually added because of some SELECT process, so there is the distinct possibility that you will have to redo your INSERT, UPDATE, and DELETE QUBE calculations over and over again. To overcome this problem, we suggest that you fill in the QUBE worksheets of the basic update operations with imaginary indexes, as in Figure 40.

"Insert a new customer"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_1 (*F*)	1	1			
XCU_1 (*M*)	1		CUST	1	1
XCU_a (*M*)	1	1			
XCU_b (*M*)	1	1			
XCU_c (*M*)	1	1			
CPU = 4.2 ms	LRT (CICS) = 156 ms				

Figure 40. Insert Single Row, with Extra Indexes

If the outcome does not exceed your alarm limit, you know that the number of *affordable indexes* for the CUST table is 4. As long as you stay within that number, you can freely design new indexes (replacing XCU_a, and so forth).

The number of affordable indexes varies per table for two reasons: for some tables the update processes are more critical than for others, and for, say a two-column table, it usually makes no sense to have four indexes.

Chapter 4. How Accurate Is QUBE?

Some pessimism is inherent in the QUBE philosophy. We want pleasant surprises in test and production, not sad disappointments. We have already seen the VP cases, for which we had a solution (divide CPU by 20), and list prefetch (ignore it). There is another pessimistic case, discussed below, and a number of cases where QUBE might be too optimistic. Not all of these cases are immediately important, particularly if you are new to QUBE.

4.1 Buffer Pool Hits

Experienced DB2 designers might say that QUBE is too pessimistic when it comes to buffer pool hits. For the less experienced designers we might have to clarify this first: a *buffer pool hit* occurs when a requested page is already in the DB2 buffer pool, a system-memory area that is shared by all users. The page has been read earlier, either by ourselves or by another program, and is politely waiting for reuse. So the estimated SR, or perhaps even an entire prefetch I/O, will not occur.

Is QUBE then too pessimistic? Not really, because with a fresh design, you can not really tell. First, there are no known buffer pool sizes. The design *itself* in due course will establish the needs for certain buffer pool sizes.
Only in the following cases would you take the buffer pool size into account:

- You are adding a function to an existing system, so you can use buffer pool measurements from production.
- During your design it has become clear that certain very important functions can be delivered only if SRs are virtually absent *and* your DB2 SYSADM has confirmed that this absence can be achieved. From this moment on, and only for certain tables and/or indexes, you can eliminate SRs in QUBE.

Second, QUBE must be kept simple, so incorporating the concept of ″probability on a buffer pool hit″ is not in the right spirit. Reasoning like:

> ″If a table has 10,000 data pages (accessed randomly) and goes into a dedicated buffer pool of 3,000 pages, the SR is needed in 70% of the requests. So why not multiply the SR time (20 ms) by 0.7?″

may be superficially attractive to some of you, but we would not recommend it. At least, not for the first year or so; first master the QUBE kernel.

4.2 Page Rereads

One of QUBE′s axioms is: ″No page re-read within the same transaction.″ For short transactions this is a very safe assumption. Because QUBE also can be used for lengthy batch processes, the axiom may sometimes be too optimistic. Take, for example, the pseudo-code in Figure 41.

```
OPEN C1  (large table T1)
FETCH C1
do while (more rows)
   SELECT unique row from table T2
   SELECT unique row from table T3
   process
   FETCH C1
endwhile
CLOSE C1
```

Figure 41. Sample Batch Pseudo-code

The normal QUBE worksheet might look like Figure 42.

INDEX	T	SR	TABLE	T	SR
			T1	100K	1
X2A	100K	50	T2	100K	1000
X3A	100K	20	T3	100K	500

Figure 42. Sample Batch QUBE

Note that the SRs on the ″lookup″ tables, T2 and T3, are limited by the number of pages.
In QUBE, you normally never count more SRs than there are pages. In a transaction environment this is very realistic, but here? If this process uses one and the same buffer pool, you may after all worry whether the 1570 pages of the lookup tables will fit in the buffer pool and will remain calmly there for the life of the process. At worst, you might have 200K of SRs on the indexes and a like amount on the tables. Figure 43 compares the respective LRTs.

best case LRT (s) : $(3 \times 100) + (1571 \times 0.02) = 331$

worst case LRT (s) : $(3 \times 100) + (400K \times 0.02) = 8300$

Figure 43. Best against Worst Case (Batch)

So it is something between just under 6 minutes and well over 2 hours. Quite a range, and you are right in asking for more precision.
We can make some observations:

- The worst case can occur only if the buffer pool is small, either in an absolute meaning (100 pages) or in a relative meaning (100 *available* pages), which happens when there is another demanding program running concurrently. That is not the natural assumption with which to start.
- Without using any statistical analysis, it looks ″obvious″ that to estimate 100K of SRs on the X3A index is not realistic. The index has only 50 leaf pages and is accessed once per loop iteration, so it will frequently reaccess a leaf page again, that is, before it is discarded.
  The same applies to the 500 data pages of T3, but of course less forcefully.

It *is* possible to use some statistical approach for this.[17] Here we explain a pitfall that you are likely to overlook at first.
Consider the lookup tables of the example in Figure 42:

- X2A, 50 leaf pages
- T2, 1000 pages
- X3A, 20 leaf pages
- T3, 500 pages.

Assume that we have a dedicated buffer pool of 2,000 pages. All of the above, including nonleaf pages, will easily fit, and all pages will be read only once; you have 1570 SRs.
Now suppose that the buffer pool gets smaller and smaller; at a certain point DB2:

- Issues a GETPAGE request
- Finds that the page is not in the buffer pool, and that there is no free (buffer pool) page available
- Needs to find a least recently used (LRU) page to make room for the new page.

Which page is in theory *LRU*? It is a page of the *largest* page set, in other words, the T2 table.
To illustrate what now happens, assume that the 1,000 pages of T2 are randomly used, each page exactly once per 1,000 loop iterations. Furthermore, the random pattern repeats itself per 1,000 iterations. Figure 44 shows the result.

Loop Number	T2 Page Number
1	88
2	705
..	..
999	156
1000	4
1001	88
1002	705

Figure 44. Random Page Access

Now assume that at iteration 1000 DB2 needs to find a buffer pool page for data page 4, because all buffer pool pages are already occupied. DB2 now inspects the *LRU chain*: no doubt, the LRU page is data page 88, which was used long ago (loop# 1); it will be discarded.
At the next iteration (1001), unfortunately, this very page is needed again. The process is repeated, and DB2 discards page 705, not knowing that iteration 1002 will need it again.

The point of this (theoretical) example is that:

- With a sufficient buffer pool size of X, you have 1,000 SRs;
- With a new buffer pool size of X - 1, you suddenly get 100,000 SRs;

[17] The IBM-provided course on QUBE demonstrates one.

So there is not a gradual transition from ″best″ to ″worst″ case; it is a very sharp line. This happens *per page set*, so again in theory, in our example the number of SRs is *one* of:

- **1,570**, with a sufficiently large bufferpool
- **100,570**, with a smaller buffer pool, causing 100K for T2
- **200,070**, again smaller, so 100K for T3
- **300,020**, again smaller, so 100K for X2A
- **400,000**, worst case, even X3A pages do not stay in the buffers.

In practice, the access patterns are of course not so ″perfectly pseudo-random,″ but the principle remains: transitions from good to bad happen suddenly and cannot be controlled or predicted easily. The conclusion is that this situation must be avoided, by adopting one of the methods directly below.

- If the rows of one of the lookup tables can be *processed in sequence instead of random*, you can get a big improvement. To achieve this table T1 has to be processed in a certain sequence (the pseudo-code assumed none). If this were by means of a DB2 sort, we would have to add (100K * 0.06 ms) = 6 CPU seconds; that is pretty cheap. This solution has a pretty side effect, discussed in 4.3, “Sequential Detection.”
- Separate buffer pools circumvent the problem altogether, or perhaps you can temporarily enlarge the buffer pool. Both options are very reasonable, from DB2 Version 3 onward.

4.3 Sequential Detection

If in the previous example we manage to process the rows of the large table in key sequence of the largest lookup table, the QUBE worksheet (Figure 42) is valid; certainly for the T2 estimates.
But it is even better than that. If you look at the pseudo-code, you will agree that DB2 could not see at first that the rows of table T2 are accessed sequentially. Rather the contrary, they are accessed by means of the unique index, one row at a time. At run time, however, DB2 has a mechanism called *sequential detection*. This mechanism monitors the access pattern per statement, and if it detects sequential access, it uses sequential prefetch.[18]
The internals of sequential detection are of no concern to us. If, for simplicity, we assume that it takes 10 pages to detect sequential access, our QUBE worksheet looks like Figure 45. (The number of SRs on X3A and T3 is subject to earlier observation.)

INDEX	T	SR	TABLE	T	SR
			T1	100K	1
X2A	100K	10	T2	100K	10
X3A	100K	n	T3	100K	m

Figure 45. Sample Batch QUBE with Sequential Detection

The only thing to consider now is whether to include sequential detention in QUBE, and how. Designers must assume that ″an ideal access path″ exists

[18] And turns it off when the access pattern changes to ″random.″

(QUBE axiom). Furthermore, they are supposed to know about prefetch: *"If DB2 knows that data is accessed sequentially, it uses prefetch, and we count one SR."* For our example, designers can do just that. In other words, they do not have to know of the sequential detection mechanism, nor of the 10 (or other number) pages that are needed anyway.

4.4 Other Factors

QUBE is meant to assist with the design of many "normal" applications. It more or less assumes "normal" row sizes, and "normal" columns. Therefore, it should not surprise you that the outcome is less reliable (optimistic) if you retrieve rows with, say, 500 columns.

Even then, QUBE still has full value if you only want to compare two different designs. If you need a better estimate on real response time, you may either derive an adjusted formula, as we have demonstrated for the VP cases, or do a little benchmarking and derive a formula (or factor) from actual measurements.

Chapter 5. Alarm Limits

When your calculation shows that the alarm limit is exceeded you must take some action. This chapter concentrates on the *kind of action* that you can take and the *impact* that the action has elsewhere.

The actions can be divided into three groups:

- Changing the physical database, with impact on other processes
- Changing the program logic, likely with no further impact
- Other actions.

We discuss these actions below. The order wherein you consider several actions depends on many factors such as:

- Are you designing a completely new system? Then you are in the middle of database design anyway, and looking there first feels natural.
- Are you adding something to an existing system? Then changing the clustering order is a very remote possibility (in practice).
- Is the problem with a small function? Then you might go back to the end users and try to get them to agree with a higher cost.

But these are only guidelines. It all comes down to *judgment and experience*, which you will gain with each project for which you use the QUBE method in a disciplined manner.

When you have a problem, your attention should always go first to the major cost component. Is it CPU? Then you must reduce the number of touches or eliminate sorting costs. Is it I/O? Then you must lower the number of synchronous reads.

Note: This chapter does not pretend to provide a complete overview of possible actions. The ingenuity of application developers is considered unlimited and therefore cannot be described.

5.1 Changing the Physical Database

One of the main foundations of the QUBE method is to be able to quickly assess what a change in the physical database means for *this* process and what impact it has on *other* processes. The main issues are:

- The clustering order
- Index design
- Denormalization.

5.1.1 Change the Clustering Order

Changing the clustering order is easy on paper, but less so in practice. The effects in QUBE are clear and dramatic, as Figure 42 shows.

"Read customers with a given ZIPCODE" (50 occurrences)					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_ZIP	50	1	CUST	50	50
CPU = 23 ms	LRT (CICS) = 1 s (nonclustered)				
XCU_ZIP	50	1	CUST	50	1
CPU = 23 ms	LRT (CICS) = 75 ms (clustered)				

Figure 46. Nonclustered and Clustered Access

Very convincing indeed. The problem, however, is that changing the clustering order is a solution for nearly every situation where a *range of rows* is needed. But if you apply the solution in all such cases, you will have to review all other QUBE calculations, over and over again.
To establish the best clustering order, QUBE is certainly very helpful, because it pinpoints so clearly the difference in SRs. But somewhere along the line you have to select your clustering order and then stick to it. In 5.2.3, "Clustering (Again)" we return to the issue of clustering order.

5.1.2 Add an Index

In the early stages of design, when you have only indexes on primary and foreign keys, adding an index will often be a natural choice. There are three reasons to add an index for better performance:

- Filtering
- Ordering
- Index-only.

5.1.2.1 Index for Filtering

Of course, you must first establish that a table scan will not do:

1. QUBE for a table scan.
2. QUBE for a VP table scan.
3. Is a table scan allowed anyway? Think of the transaction environment.
4. QUBE for an index scan.

In practice, the third point will cause experienced designers to think of an index scan right away. You may in fact set alarm limits per table, a number of pages beyond which you will not allow a table scan in the transaction environment, unless specifically approved by the DB2 system administrator.

5.1.2.2 Index for Ordering

For DB2, the perfect case is that shown in Figure 47.

"Print all orders, in CUSTNO sequence"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XOR_2	100K	1	ORDER	100K	1
CPU = 40 s	LRT (batch) = 120 s				

Figure 47. Ordering via a Clustered Index

The index is read with prefetch, and the table too, because the Catalog tells DB2 that the table is clustered on CUSTNO. It is important to understand that DB2 can deliver the rows without initial overhead. The process is *fetch driven*,

meaning that the meter only runs as long as the program continues to fetch the next row.
Note by the way that *clustered* is not the same as *clustering*. By definition there can be only one clustering index, for instance on CUSTNO, but there can be another index – or even more than one – that is clustered. As an example, if a new CUSTNO is somehow based on CUSTNAME, an index on CUSTNAME could be highly clustered, and DB2 would know it too.

Now suppose that you also have the process shown in Figure 48.

"Print all orders, in ORDER_DATE sequence"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
			ORDER	100K	1
NS=1 NSR=100K					
CPU = 26 s LRT (batch) = 78 s					

Figure 48. Ordering via a DB2 Sort

Note the extra line in the QUBE worksheet, describing the sort process. This time there is a very distinct initial overhead, namely the entire sort. In contrast to fetch driven, a programmer would call this OPEN CURSOR driven.

Can you make this cheaper with an index on ORDER_DATE? Let us look at Figure 49.

"Print all orders, in ORDER_DATE sequence"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XOR_OD	100K	1	ORDER	100K	5000
CPU = 40 s LRT (batch) = 160 s					

Figure 49. Sort via a Nonclustered Index

The number of touches is doubled, and there are many SRs, so obviously DB2 will reject this access path. When do you actually want such an index for ordering? In these cases:

- If it is combined with filtering. If, in the example in Figure 49, we would only need a small range of dates (a week), the index would be used anyway, for filtering.
- If you do not intend to fetch all rows. This is the classical online browse situation, where you can only display some 20 lines on a screen. The access path of the example in Figure 48 is now devastating for performance, because of the initial overhead.

  So we create the XOR_OD index for a single screen (see Figure 50).

INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XOR_OD	20	1	ORDER	20	20
CPU = 11 ms LRT (CICS) = 437 ms					

Figure 50. Ordering via a Nonclustered Index (One Screen)

  The only thing left now is to convince DB2 that this is indeed the best access path. The coding should be:

```
SELECT OR.*
  FROM ORDER OR
 WHERE ...
 ORDER BY OR.ORDER_DATE
 OPTIMIZE FOR 1 ROW
```

Without that last clause, DB2 would still choose the access path of Figure 48, because it does not know how many rows we intend to fetch.

Note: The experienced technical designer (or programmer) knows that in most of these situations a *unique ordering* is required. In the example, it is most likely that the ordering is actually by ORDER_DATE, ORDERNO. If so, an index with both columns should exist.

- As a last possibility, if you have an extremely large answer set, there may not be enough space in the DB2 work files for sorting. These work files are shared among all concurrent users of DB2 (needing some sort), so the limit could be reached earlier than expected.

  The use of an index with an `OPTIMIZE FOR 1 ROW` clause can circumvent this. The QUBE worksheet is that of Figure 49, so it is a costly solution, unless you have the data pages already in the buffers (no SRs). An alternative might be to use an external sort.

5.1.2.3 Index for Index-Only

To repeat and extend our observation on adding an index for index-only:

- Indexes can have up to 64 columns, provided that the length of the index key does not exceed 254 bytes.
- Any VARCHAR field in an index deserves attention:
  - It is expanded to maximum length. In a multicolumn index, you will quickly arrive at the 254-byte limit.
  - It does not work for index-only anyway, because the length field is stored in the data page.
- In the following example, index-only might also not happen:

```
/* index X1 on (C1,C2,C3)  */

SELECT ..
  FROM ..
 WHERE C1 = 12
   AND C3 > 100
```

Assuming an index scan, it is a matching-index scan with MATCHCOLS=1. The predicate `C3 > 100` is tested *after* retrieving the index page. This is called *index screening*.
If DB2 decides on list prefetch, however, it does *not* perform index screening as well; in other words, it goes to the data page. To overcome this problem, an `OPTIMIZE FOR 1 ROW` clause can be added.

5.1.2.4 Index drawbacks

In all cases, if you add an index, you must consider the following:

- Extra costs for insert and delete, 1 T and 1 SR
- Extra costs for updates on the indexed column, 2 Ts and 2 SRs
- How many ″duplicates″ will there be? Many duplicates affect inserts and deletes (see *RID chains* on page 15).

A common solution is to add an extra column to the index. If no natural choice presents itself – a column that will also help *filtering* – either add the primary key or make a note that you will reconsider the issue of duplicates (or *cardinality*) at the very end of your design.

5.1.3 Drop an Index

When studying update programs you may hit an alarm limit just because of the number of indexes. However, if you remove one or more indexes, some other process will suffer. How should you proceed then? Here are some alternatives:

- You may try to combine indexes. If, for instance, you have separate indexes on CUSTNO and ORDER_DATE (in the ORDER table), you might look at a single index on CUSTNO,ORDER_DATE. You must still review previous calculations, because what used to be a matching index scan on ORDER_DATE now becomes a nonmatching index scan; but perhaps you can afford that.
- Increase buffer pools to eliminate SRs (typically, on indexes).
- Temporarily drop one or more indexes. This is a solution only for a stand-alone batch environment, but it can be a good one. Suppose you have a table with four indexes, as in Figure 51. (We assume some arbitrary number of entries per leaf page and data page, and 100 commits.)

"Add 100,000 rows. No RI Checks"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
X1	100K	1K			
X1	100K		T1	100K	10K
X2	100K	1K			
X3	100K	2K			
X4	100K	3K			
CPU = 120 s LRT (batch) = 705 s					

Figure 51. Batch Inserts, Four Indexes

If you drop all indexes except X1, you get the case shown in Figure 52.

"Add 100,000 rows. No RI Checks"					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
X1	100K	1K			
X1	100K		T1	100K	10K
CPU = 60 s LRT (batch) = 405 s					

Figure 52. Batch Inserts, One Index

A clear difference. The comparison is not honest, however. To the first case we must add cost to:

– format new index pages, possibly requesting new extents. Although the QUBE axiom for INSERT is *"enough free space is available,"* it is not always realistic to apply this to mass inserts in batch.

– REORG four indexes afterward. The usual action after mass inserts.[19]

And to the second case we must add cost to:

– DROP three indexes
– REORG one index afterward
– CREATE three indexes afterward
– REBIND plans and packages. Even if done automatically, it costs.

Dropping an index (or more) is worthwhile considering only in a heavy batch process. Each case must be looked after by a specialist. If your installation's rules do not allow this kind of solution – it is operationally complex and therefore risky – you must manage the indexes differently, for instance:

- REORG in advance, with enough freespace.
- Provide sufficient buffer pools, if only temporarily.

5.1.4 Change Ascending Index to Descending

You may have an index that is designed only to provide filtering. It is (by default) designed as an ascending index. You may subsequently have some process that requires descending ordering or perhaps retrieving the maximum value of the index column. You can consider to redesign the index as descending. It is only fair to say that this seems more a theoretical than a practical possibility, and we have not encountered an opportunity for this idea in practice.

5.1.5 Change Order of Index Columns

This action applies only to multicolumn indexes, obviously. It can be an existing index on a composite primary or foreign key, or any other multicolumn index.
As an example, in our sample database we have the index XOI_1 (ORDERNO,ITEMNO), a primary key. Reversing the two columns in the index does not in any way affect the principle of the primary key, but it helps any process whose filtering or ordering is based on ITEMNO. The other side of the coin is of course that similar processes on ORDERNO will now become less efficient.

You must therefore have a complete picture of all your processes, before you can establish the best order of the index columns. It is of no use to "flip-flop" back and forth per process.

In a query environment, if your multicolumn index is on nonkey columns – primary or foreign – and you are uncertain about the column order, you could also consider having two separate single-column indexes and rely on DB2's *multiple index scan* mechanism. This will often be a good choice for large tables in a query environment. The exact processes, their frequency, and required response times are not always well known anyway; what *is* known, however, are the columns that will be used in selecting the data, in whatever combinations. Multiple index scans are well suited for this.

[19] It does not matter whether the indexes are reorganized separately or as part of a REORG TABLESPACE.

5.1.6 Denormalize Your Tables

There are a large number of possibilities here. We examine the more likely possibilities, but not without remarking first that it is a serious mistake to think that *denormalizing is always needed in a relational database*. There may have been some (performance) truth in this in the early 1980s, but not now.

5.1.6.1 Case 1

Without even showing LRT, Figure 53 makes it obvious that the accesses to the CUST table are a major cost factor.

"Retrieve a range of order data (based on ORDER_DATE), together with customer's PHONENO." (20 occurrences)

INDEX	T	SR	TABLE	T	SR
XOR_OD	20	1	ORDER	20	20
XCU_1	20	20	CUST	20	20

Figure 53. A Join of Two Tables (Normalized Tables)

If this is a critical transaction, or the requirement to get the phone number is frequent, you may consider adding the phone number to the ORDER table. It would yield the QUBE worksheet of Figure 54.

"Retrieve a range of order data (based on ORDER_DATE), together with customer's PHONENO." (20 occurrences)

INDEX	T	SR	TABLE	T	SR
XOR_OD	20	1	ORDER	20	20

Figure 54. Denormalized Tables

This is a violation of third normal form (*no functional dependency between attributes*). Here are the impacts:

1. Changing the phone number becomes considerably more costly. In this case it is unlikely to be a real problem, *but it must be looked after*.
2. Adding a new order now requires access to the customer table to get the PHONENO. This seems to imply that you incur additional cost with the denormalized ORDER table (for INSERTs).
   However, if you think about the entire process *"*add a new order*"*, you will agree that the customer table must be read at some time during the process anyway. Therefore, the aforementioned additional cost would also occur without denormalization.
3. The rows become longer, so you need more pages. This *might* cause a difference in QUBE: if at some place you have limited the number of SRs to the number of pages, you will have to adjust this.
   More pages in the table also has effects outside QUBE. DASD estimates, table space parameters, and buffer pool estimates must be reconsidered.
4. Adding a column to the ORDER table does not mean that it is taken away from the CUST table, so you must worry about *consistency of the data*. Not because you distrust your programmers, but just because.
   So you must periodically check this consistency. It is not the cost of the check itself that should worry you (index-only access, once a week), but the overall project costs of having an extra program, extra scheduling, and the like.

If you review the points above you might still say to yourself, *"Well, that's not much impact, so let's do it."* However:

- There are always other solutions. In our case, the number of SRs to the CUST index and table (pessimistic as they are) can probably be reduced significantly by some buffer pool change.
- If you go for denormalization, the next thing users want included is the customer's name (or something else). Now what? Yet another column in the ORDER table? Or back to fully normalized tables?

Conclusion for Case 1:
Consider this form of denormalization only reluctantly. While the gain is obvious, it is only *one gain* against *many drawbacks*. Here is a good candidate for denormalization:

```
Version 0:
  Table EMP  (EMPNO, DEPTNO, ..)
  Table DEPT (DEPTNO, DEPTNAME, ..)

Proposed Version 1:
  Table EMP  (EMPNO, DEPTNO, DEPTNAME, ..)
  Table DEPT (DEPTNO, DEPTNAME, ..)
```

Suppose DEPTNO is just a key, with few people aware of its existence. That means that DEPTNAME is – in the end-user world – the key and possibly UNIQUE as well. The consequence is that DEPTNAME is probably needed in many processes, not only in the case at hand, which means that you will always need a join between the tables EMP and DEPT. By denormalizing you can eliminate some of these joins. In a query environment, it also makes the coding easier. the case at hand.

5.1.6.2 Case 2

A violation of second normal form (*no functional dependencies between attributes and part of the primary key*).

An example would be to add ITEM_NAME to the ORDITEM table, instead of retrieving it from the ITEM table. Technically this is not different from Case 1.
For both cases you must realize that sometimes it is not denormalization at all. For instance, the attribute ITEM_PRICE may be included in two tables: in the ITEM table it is "current price," and in the ORDITEM table it is "ordered price." Two entirely different attributes, which are best given a different name to avoid confusion.

Conclusion for Case 2:
Be reluctant to use this form of denormalization. Same arguments as for Case 1.

5.1.6.3 Case 3

A *vertical split*. The idea of splitting a table is to put less frequently used, perhaps large, columns in a separate table. The primary key is included, of course.
The original table contains the "main" columns and now occupies far less space. The gain should come from this. The number of pages becomes so much less that I/O costs will go down significantly, even to the point where the entire table fits in a buffer pool. In QUBE, if we have SRs, they will decrease; if we have only prefetches, having all pages in the buffers does not help much in the way of LRT, but also for prefetch it is obviously good if there is no I/O at all. This will give

better overall performance and throughput, although not visible in standard QUBE (apart from the initial SR).

On the negative side we note:

- Inserts and deletes will cost at least 2 Ts and 2 SRs extra.
- Likewise for updates that need the two tables.
- Some multicolumn indexes might become impossible.
- If there is a true one-to-one relationship between the two tables, you need extra safety checks to determine whether this is achieved as indicated in Figure 55. It cannot be enforced with DB2′s Referential Integrity:

```
CREATE TABLE T1 (PK1 INTEGER NOT NULL, ..
       PRIMARY KEY (PK1) ) ;
CREATE TABLE T2 (PK2 INTEGER NOT NULL, ..
       PRIMARY KEY (PK2) ) ;
ALTER TABLE T1 ADD FOREIGN KEY (PK1)
      REFERENCES T2 ;
ALTER TABLE T2 ADD FOREIGN KEY (PK2)
      REFERENCES T1 ;
```

Figure 55. Split a Table Vertically

So far, no problem. But, *how do you insert a row in either table?*

Splitting a table may be an advantage for other reasons: for example, you may give it a different clustering order, different table space options, or a different REORG frequency.

Conclusion for Case 3:
Splitting a table vertically can have merits. Try to identify some long fields that are used seldomly and put them away in a different table, likely also a different table space. Note a difference from the first two cases: the number of pages of your critical tables is now reduced instead of increased.

5.1.6.4 Case 4

The idea of storing data that can also be derived from other data usually gets different reactions. The theorists dismiss it as ″redundant.″ The practical-minded see it as an easy way out of a performance issue. Figure 56 is a typical example.

″Show customer name and telephone of a given CUSTNO, and the total number of orders (50 on average)″

INDEX	T	SR	TABLE	T	SR
XCU_1	1	1	CUST	1	1
XOR_2	50	1			

Figure 56. Derived Data

It is of course correct to design an index-only access of the ORDER table, but it is still 50 touches and 1 SR. So why not add an attribute NO_OF_ORDERS to the CUST table? The arguments against it are similar as in previous cases. In this particular case it helps enormously in QUBE′s CPU and LRT, but you must not forget that the index-only access was actually VP because only one row is fetched (SELECT COUNT(*) ..).

It seems that this kind of solution sometimes comes from people who start off like this:

```
SELECT CUSTNAME,CUSTPHONE,COUNT(*)
  FROM CUST CU,ORDER OR
 WHERE CU.CUSTNO = OR.CUSTNO
   AND CU.CUSTNO = :custno
 GROUP BY CUSTNAME,CUSTPHONE
```

Having done so, they are first quite proud of the solution (*one statement sir!*),[20] but then they start to worry about performance (not necessarily a problem) and finally realize that it simply is not correct (if the customer has no orders). This is where they ask for the separate counter in the CUST table.

This case is a typical case of keeping a count of the number of occurrences in a dependent level; it might also become a typical case of locking problems and – again – inconsistencies.

The ″right″ solution (in Version 3) is:

```
SELECT CUSTNAME,CUSTPHONE,COUNT(*)
  FROM CUST CU,ORDER OR
 WHERE CU.CUSTNO = OR.CUSTNO
   AND CU.CUSTNO = :custno
 GROUP BY CUSTNAME,CUSTPHONE
UNION ALL
SELECT CUSTNAME,CUSTPHONE,0
  FROM CUST CU
 WHERE CU.CUSTNO = :custno
   AND NOT EXISTS
      (SELECT 1
         FROM ORDER OR
        WHERE OR.CUSTNO = CU.CUSTNO)
```

Or, in Version 4 with its *outer join* support:

```
SELECT CUSTNAME,CUSTPHONE,VALUE(COUNTER,0) AS NO_OF_ORDERS
  FROM CUST CU
       LEFT OUTER JOIN
      (SELECT CUSTNO,COUNT(*) AS COUNTER
         FROM ORDER OR
        GROUP BY CUSTNO) AS TEMP
       ON CU.CUSTNO = TEMP.CUSTNO
 WHERE CU.CUSTNO = :custno
```

If you are used to thinking ″relationally″, you will have little difficulty in understanding this query.[21] However, at present we venture to say that most programmers would code:

```
SELECT CUSTNAME,CUSTPHONE
  FROM CUST CU
 WHERE CU.CUSTNO = :custno

SELECT COUNT(*)
  FROM ORDER OR
 WHERE OR.CUSTNO = :custno
```

[20] The keen SQL-ist will jump at the opportunity to point out that the GROUP BY can be avoided by coding MIN(CUSTNAME), MIN(CUSTPHONE).

[21] Also, it is likely that you will think of other formulations, not necessarily yielding the same access path or performance.

Conclusion for Case 4:
It is not so much a performance problem, but a convenience problem. In a transaction environment, with everything precoded, you can use the two statements shown above and get your result.

In the query environment, however, you may need the result in one result table, and hence the complex statement with UNION or outer join. If this is beyond the capacity of your users, or beyond the willingness of your database administrator to build a number of views, you have more reason for the separate ″counter″ field. And if the query environment is on more or less frozen data (history), the consistency problem is small.

5.1.7 Create a Larger Buffer Pool

This is an easy way out. In Chapter 6 we discuss buffer pools in detail. Some conclusions are:

- If designers ask for a larger buffer pool, they must show QUBE worksheets, proving that the requested performance cannot be obtained otherwise.
- Dedicated buffer pools, with guaranteed in-memory processing, can be achieved only if there is good size control over the data. The easiest tables are small tables that are infrequently changed, such as code-reference tables.

5.2 Changing the Program Logic

Changing the database is an option, but it may also be possible to change the program logic. If there are many programs that perform similar tasks, designing and testing out a *skeleton program* can be of great benefit.

5.2.1 Read Partial Answer Sets

Figure 57 shows a familiar example of reading a partial answer set.

″Retrieve customers with a given ZIPCODE, order by NAME″ (50 hits expected)					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_ZIPNAM	50	1	CUST	50	50
CPU = 23 ms LRT = 1 s					

Figure 57. Retrieve 50 Rows

The case says ″Retrieve ....″ There are two possibilities:

- A batch program, retrieve = print. The QUBE worksheet is correct.
- An online program, retrieve = display. The QUBE worksheet now assumes that we first fetch all rows and then start to display them, for example, 20 at a time.

If we have the latter situation, and if our alarm limit is LRT < 0.5 s, we could rewrite the QUBE worksheet and program logic to retrieve one screen (Figure 58).

"Display customers with a given ZIPCODE, order by NAME" (one screen of 20 rows)					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_ZIPNAM	20	1	CUST	20	20
CPU = 11 ms LRT = 437 ms					

Figure 58. Retrieve One Screen

The program now retrieves no more than it can display. Please note that the index on ZIPCODE,NAME is essential; if only on ZIPCODE, DB2 would need a complete sort of the 50 rows.

Experienced designers and programmers will recognize that the example is inaccurate, in the sense that two rows with the same NAME will be returned in a random order. This makes the *screen repositioning* of the last example awkward. The known solution is that the ordering should be by something unique, like NAME,CUSTNO; the index should be extended accordingly.
The repositioning algorithm then becomes:

1. Define host variables for all columns of the `ORDER BY` clause (`:last_name` and `:last_custno`).
2. Extend the `WHERE` clause:

```
... WHERE  ZIPCODE = :zipcode
     AND ((    NAME   = :last_name
           AND CUSTNO > :last_custno)
            OR NAME   > :last_name)
```

   The important thing here is the overall construct:

```
basic predicate-1 AND (compound predicate-2)
```

   It must never be with `OR`, because you then get a table scan. In this example, please note that the major filtering was (and remains) on ZIPCODE. In a case where we would not have the first predicate, and an index XCU_NAMCUST, the logic should be:

```
... WHERE  NAME  >= :last_name
  AND NOT (    NAME    = :last_name
           AND CUSTNO <= :last_custno)
```

   (if you read AND NOT as BUT NOT, you will understand)

3. Initialize the host variables correctly and, after filling a screen, assign them to the values of the last displayed row.

5.2.2 Presort Input

We already had an example of this, see the discussion after Figure 41 on page 38. Figure 59 is the basic example.

"Update the CUST table, with sequential input (500 records)."					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_1	500	50	CUST	500	200

Figure 59. Batch Program, First Approach

We assume no updating of indexed columns.

If the input were sorted on CUSTNO (a small requirement), the QUBE worksheet becomes that shown in Figure 60.

INDEX	T	SR	TABLE	T	SR
XCU_1	500	1	CUST	500	1

Figure 60. Batch Program, Second Approach

To explain Figure 60, we add the pseudo-code:

```
READ seq.input
do while NOT EOF
   UPDATE CUST
      SET ...
    WHERE CUSTNO = :custno
   READ seq.input
endwhile
```

(Slightly different if we need to read the row first, but not affecting QUBE).

Obviously, DB2 cannot see this as a multirow process (on the contrary). Still, as already explained in 4.3, “Sequential Detection” on page 40, there is a run-time ″catcher″ for this; we also concluded that we would not bother about the 10 SRs (or so) to start it off.

5.2.3 Clustering (Again)

The last example (Figure 60) is a typical batch environment that benefits from processing rows in clustering order. In such scenarios the driving key is often the primary key for the table, suggesting that the corresponding index should be defined as clustered. But what about the other environments?

5.2.3.1 Online transactions

A lot of transaction workload is based on unique keys, where clustering is not an issue. The rest is retrieval – perhaps update too – of (small) ranges, such as:

- All customers for a given ZIPCODE
- All orders for a given CUSTNO.

The beneficial clustering now is always on a nonunique key, quite often a foreign key.

5.2.3.2 Queries

Because of their ad hoc nature, it is not possible to analyze all queries in advance. Nevertheless, you should try to answer the following questions:

1. In which order is the data frequently wanted?
2. On which columns are ranges frequently selected? One would expect typical foreign key fields here, such as CUSTNO (in the ORDER table) or ITEM (in the ORDITEM table).
   If ranges on primary keys are frequently needed, then in our opinion that is an indication (but not proof) of badly designed keys.
   If, for instance, †CUSTNO > 8999† is meaningful, there is probably a hidden meaning within the primary key (″foreign customers have CUSTNOs of 9000 and higher″). Such key design is contrary to theory, but any data processing professional can tell you that it still occurs frequently. A good example in the DB2 area itself is:

```
/* show me the names of your tables (not Views) */

SELECT NAME
  FROM SYSIBM.SYSTABLES
 WHERE CREATOR = USER
   AND TYPE = ¢T¢

/* BUT NOT, because of some clever naming convention: */

SELECT NAME
  FROM SYSIBM.SYSTABLES
 WHERE CREATOR = USER
   AND NAME LIKE ¢T%¢
```

Do not let yourself be carried away by the "performance argument" that the latter SQL statement runs faster; it does indeed, but the former statement gives the *correct answer*, which is what counts.

Because by definition there is only one clustering order, and because updating and reading processes always conflict with each other (as far as performance is concerned), there will always be trade-offs. The large, updating batch programs tend to benefit from clustering on the primary key, if they receive keyed input. Transactions and queries tend to benefit from range retrievals, often combined with ordering on the same column(s); these are often foreign key columns.

The right way to handle the issue is to identify and prioritize all of your processes and use QUBE for the performance analysis.

5.3 Other Actions

All of the issues discussed above have been technical. There are also nontechnical approches for dealing with your alarm limits. Some of these approaches are very practical, as discussed below.

5.3.1 Back to Your Users

A clear option that you have, in the case of an exceeded alarm limit, is to go back to your users and rediscuss the performance requirement. For instance, it is not at all uncommon that the first stated requirement for an infrequent process, such as end-of-week processing, is not a real requirement, but just a guess.

Therefore, even if your users stick to their requirement, you can explain to them that, to meet the requirement, you have to perform a number of tricks, with such consequences as:

- Increased project costs
- Increased project risks
- Increased computer costs for other transactions
- Increased charge-back costs from production.

Also, you could review the estimates on the size of the results. These translate directly into *touches*, and therefore CPU. It is possible that someone has given you a very high number, just to be on the safe side. QUBE is in and of itself on the safe side, and its input should be realistic (not a theoretical worst case).

5.3.2 Accept the Outcome

Accepting the QUBE outcome to be ″somewhat above″ the alarm limit may be a possibility. Let us look at Figure 61 as an example.

″Retrieve customers with a given ZIPCODE, 50 hits expected″					
INDEX	**T**	**SR**	**TABLE**	**T**	**SR**
XCU_ZIP	50	1	CUST	50	50

Figure 61. Retrieve 50 Customers

Suppose that the eventual outcome (either CPU or LRT) exceeds your alarm limit by only a small margin. Wouldn't you then be tempted to say, *″Well, let's leave this for the moment″*? It depends. You should make the following distinction:

- CPU too high? The CPU cost, as estimated by QUBE, should in reality be lower, but not ″infinitely″ lower. After all, it is a function of the number of rows needed to obtain the result.
  If this is a frequently executed transaction, the alarm limit should be taken seriously because it affects the number of transactions per time unit.
  Otherwise, you could build a realistic prototype environment. If that confirms a high CPU cost, you can still reconsider.
- LRT too high? A major component is now the number of SRs. Contrary to CPU, these *might* be reduced to zero.
  In our example, with only 200 pages, the estimate of 50 SRs indicates that:
  - All hits are on different pages. Possible, but unlikely.
  - None of the needed pages is in the buffer pool. Possible, but (very) unlikely, because the natural assumption is that the customer is an important entity in our application and will be referred to by many transactions. Hence, the pages of the CUST table and its indexes tend to stay in the buffer pool.

  In other words, 50 SRs *does* seem pessimistic, although we have no further quantification. It is possible to accept this and hope for the best; a more formal approach would be to state something like *″no more than 30 SRs are acceptable″* (30 being a number that yields an acceptable LRT). This statement would give direction to the size and type of buffer pool for this particular table, and to the monitoring strategy in production.

Finally, if the limit is exceeded by a large margin, you certainly cannot accept the outcome. What you can do is:[22]

- Report to project leader, as a risk
- Report to programmer, as a challenge
- Report to user, as a problem
- Report to specialist (yourself, maybe, to add to your prestige as a specialist).

You should not have *too* many of these problem cases, obviously, but it is better to detect them early, instead of embarking on a totally impossible project.

[22] Observation made by Werner van Ipenburg.

Chapter 6. Implementing the Physical Database Design

From the preceding chapters we know how QUBE is used in the technical design phase. It is now useful to review the output of that phase as it is relates to QUBE. The primary output is a physical database design, in terms of tables, columns, and indexes. Each attribute is put into a specific table and gets a specific data type, length, and NULL attribute. Indexes are defined on primary and foreign keys, and on any other columns if you have discovered their need (through QUBE).

The physical design must still be *implemented*, in at least the test and the production environment. It is then that you look at:

- Table to table space mapping
- All table space and index space options
- Buffer pool mapping, and buffer pool sizes.

Note: This chapter is not directly about QUBE, and you can skip it at first. We include it because in a normal project cycle the real physical database implementation is (of course) an activity of the database administrator, and the starting point is a physical model, thoroughly tested with the QUBE method. QUBE may have identified several cases where the number of SRs is a bit high, leading to a side note that adequate buffer pools are required. The subject of buffer pools is an important one in this chapter, and hence there is some relation with the SRs in QUBE.

6.1 How Is a Table Used?

Physical implementation decisions require insight into the use of each table (see Figure 62).

Table : ORDER (100,000 rows, 5,000 pages)
XOR_1 : ORDERNO (500 leaf pages)
XOR_2 : CUSTNO (500 leaf pages)

Process	SELECT	INSERT	UPDATE	DELETE	Freq.	Index
Online						
New order		X			High	All
Cancel order				X	Low	All
Batch						
Print orders	X				Day	
Confirmation	X				Day	XOR_2
Cleanup			X	X	Month	All
Query						
Order inquiry	X				Med	XOR_1
Cust. inquiry	X				Med	XOR_2
Gen analysis	X				Low	Any

Figure 62. Synopsis of Table Usage

You also need information on the INSERT and DELETE rates (Figure 63).

```
INSERTS/day   :  1,000
INSERTS/month : 22,500
DELETES/day   :      5 (cancellations)
DELETES/month : 22,500 (clean-up)
```

Figure 63. Table Update Rates

6.2 Mapping Tables to Table spaces

The number of tables per table space is an important issue and by no means clear-cut. Will we have one table or more tables per table space? The changing features of the various DB2 releases have caused sways in either direction. Nevertheless, use the following remarks, and you should get some approving nods:

- *"It depends"*
  The ever-correct answer, usually coming from a specialist. It will be followed by a one-hour lecture, ending with: *"*.. so you see, there are a lot of options; it is up to *you* to decide which is best ...."
- *"Put large tables in a separate table space"*
  Carefully avoiding a definition of "large," of course; never commit yourself.

- *"If more than one table in a table space, it should be a segmented table space"*
  Few people will disagree with this, but the *If* still leaves you uncertain about *whether* to do it.
- *"Tables in the same table space should have more or less the same update rates, preferably small"*
  In other words, the prime candidates are read-only tables.
- *"There is a point where the number of data sets per DB2 subsystem becomes too high to be practical; you must then either define a second subsystem or combine tables in table spaces."*
  This remark is intended for those who have a firm rule "one table per table space" and are quite happy too but do not (yet) have thousands of tables.

For present purposes – mainly our discussion of buffer pools – we assume a one-to-one mapping. Therefore we can speak freely of "putting a table in a different buffer pool," and, more importantly, if we do so, no other table is involved.

6.3 Table space Options

Based on information such as that in Figure 62 we have to make decisions on the various CREATE TABLESPACE options. It is not our intention to go into great detail here. In conjunction with the example we can say:

- **PRIQTY:** 5,000 pages is 20M, similar reasoning for indexes.
- **SECQTY:** it is not read-only, so specify a small amount.[23]
- **PCTFREE:** new orders will have increasing ORDERNOs, but they will be scattered throughout the table because the clustering is on CUSTNO. You have to look at your REORG schedules:
  - **REORG once a month.** The volume (100,000 rows on 5,000 pages) is reached just before cleanup. This process deletes 22,500 rows (23%). If PCTFREE is 23%, we will keep 5,000 pages after the REORG. Together with FREEPAGE this should be enough (at least for now, during design).
  - **REORG twice a month.** If you have just derived the number that fits a monthly REORG, you can now simply divide by 2; PCTFREE 11.
  - **Not REORG at all**, on the theory that the deleted rows will be occupied by new rows. If so, PCTFREE does not matter at all. In practice we would still choose something explicitly, like PCTFREE 10; you never know when you will need it.

  Note that if the INSERTs were at the end of the table (clustering index on ORDERNO), REORG after cleanup would definitely be needed and PCTFREE can be set to 0.[24]

  So, while discussing PCTFREE, we have come across clustering once more. This emphasizes how important clustering is, and how important QUBE is in establishing the right clustering in advance. If you have to change clustering after a year of production, you will have a lot of stuff to review!

[23] If you want allocation units of CYLs, your SECQTY must at least be one cylinder; otherwise it will become TRK.

[24] Disregarding VARCHAR expansions.

- **FREEPAGE:** this is useful if you have either very long rows or extremely varying VARCHARs; if you have neither, you could consider "0". However, you have random inserts and want to maintain a reasonable level of clustering, so a few free pages per segment would be nice. Specify a value of, say, 20.
- **LOCKSIZE:** an online updated table, so PAGE or ROW.[25] You may make a side note that programs that are running stand-alone, may (or must) use the LOCK TABLE statement.
- **CLOSE:** this is too release- and environment-dependent to discuss here. We suggest YES, but check your current manuals.
- **SEGSIZE:** based on the table size, 64. The most simple rule for SEGSIZE is to set it to a multiple of 4, just larger than the number of pages in a table, and not larger than 64.
  Talking about SEGSIZE implies that we do not go for *partitioning*. In general, the larger the table, the more likely it is that you want partitioning. If you come to that conclusion at this stage, you must look at the designed clustering, in our example on CUSTNO. For partitioning, you must be able to design ranges of CUSTNOs that will be put into each partition. Because the INSERTs are random with respect to CUSTNO, you have no further problems.
  If, however, clustering would be on ORDER_DATE, you must realize that the *partition ranges are fixed* after your CREATE INDEX statement. If you design one partition per month, the index specifications would start like this:

  ```
  .. PART 1 VALUES(¢1995-01-31¢)
     PART 2 VALUES(¢1995-02-28¢)
     etc.
  ```

  You could go on like this until partition 64, so covering more than five years. When that period has expired, however, you must drop and re-create all your definitions.
  Conclusion: For very large tables, where partitioning simply is needed for manageability, the choice of the clustering index must be made even more carefully.
- **COMPRESS:** frequently used table, so NO. See also Chapter 9, "Closing Topics" on page 99.
- **STOGROUP**: this must fit into your general ideas of spreading data over volumes. A common idea is to put data and related indexes on different volumes. Perhaps your installation manages this via SMS (System Managed Storage), not via DB2 STOGROUPs.
- **private table space?** yes, but if you have a strong preference for combining tables, the ORDITEM table is a candidate: its updates are likely to be at the same time as on ORDER, so the backup and reorganization schedules are the same.

[25] Row locking provided in Version 4. The use of ANY is possible, but the supposed advantage, *lock escalation*, is not always as good as it looks.

6.4 Buffer Pool Overview

The buffer pools in DB2 consist of 4K-slots in memory. After reading from DASD, the data and index pages go into these slots and remain there until the *buffer manager* decides to use their slots for other pages. The idea is that data gets a chance to be reused, thus *minimizing I/O*.

The decision to free a buffer pool slot (″steal a page″) is based on an LRU principle. The pages are administered in a logical usage chain; every time a page is touched, it becomes ″top-of-list″ (or, *most recently used*).[26] So if a page is touched with a certain frequency, it will never become LRU; all requests for that page will be logical instead of physical. This phenomenon is a major reason why database management systems can deliver very good performance. DB2 in fact has two LRU-chains, one for randomly and one for sequentially accessed pages. Thus there is some protection from large table scans stealing all of your random pages.[27]

Database management systems also deliver good write performance because of buffer management. DB2 maintains logical chains of pages to be written (because they have been updated) and waits as long as reasonable before writing them. Thus UPDATES PER PAGE and PAGES PER WRITE I/O, two key figures in DB2 monitoring, can reach high values. Writing out chains of pages may be considered normal in a batch environment, but sometimes it can be achieved in a transaction environment as well. Moreover, the page writes are performed asynchronously and therefore do not affect response time.

DB2 Version 3 has many enhancements in buffer pool management, specifically:

- Defining up to 50 buffer pools[28]
- Altering the sizes without stopping DB2
- Controlling some internal thresholds, previously fixed
- Increaseing the sum of all buffer pools to 9.6GB (1.6GB before), by using *hiperpools* in expanded storage.

It is quite possible that you are not using any of these enhancements, and if you are still happy, please do not change but by all means read on.

In a mature DB2 environment, the exploitation of the Version 3 enhancements presents exciting challenges but requires discipline. To be specific, you should have:

1. Proper design techniques (among them, QUBE)
2. Good ″size control″ over the data (knowing how large a table is and will eventually become)
3. Well-established monitoring environment, such as regular reports from DB2 trace data (Statistics and Accounting), *and people who understand those reports*
4. Well-defined REORG frequencies
5. Good change-management control over the applications
6. Good communication among people, such as the MVS systems programmer, the DB2 system administrator, and the developers.

---

[26] This does not happen always; sometimes DB2 knows that a page is (likely to be) not needed again. The buffer pool slot therefore becomes immediately available.

[27] Another protection mechanism is the **VPSEQT** parameter, which is alterable.

[28] We do not discuss 32K buffers (another 10).

Few installations have managed to achieve all of the above requirements . We will settle for four out of six and the willingness to look at the other two.
For the rest of this chapter you must picture a Version 3 system with more than three buffer pools (the previous limit).

6.4.1 MVS Paging

If you have no or little *expanded storage* (ES), an increase in the buffer pool will cause MVS to write buffer pool pages to the *page data set*. If the buffer pool pages would be needed again, which is likely, MVS must *page in* them again. This is something that should be avoided. You might as well have a smaller buffer pool and let DB2 do the I/O than an MVS paging operation with its I/O and CPU overhead. So this is a situation that the system programmer should monitor.

If you have expanded storage, MVS will page buffer pool pages to ES, which is certainly more efficient than paging them to the page data set (no I/O), but still not as efficient as hiperpool buffer pools.

6.4.2 Hiperpool Buffer Pools

It is possible to have a part of a buffer pool in *expanded storage* and managed by DB2 as a *hiperpool*. Then if you need to increase your buffer pools from 5,000 to 40,000 pages, you can define, say, 35,000 pages in hiperpools. This should please the MVS systems programmer, who otherwise might worry about the MVS overhead of managing *virtual storage*, as would happen if you define a normal buffer pool of 40,000 pages.

For this option to work you need specific hardware; check your DB2 manuals. In addition, there is an excellent chapter on this topic in *DB2 V3 Performance Topics*.

6.4.3 Hit Ratio

If you want to monitor your buffer pools with just one number, the *hit ratio* is the number to use. The formula for buffer pool hit ratio is:

$$\text{Hit Ratio} = \frac{GETPAGES - PAGESREAD}{GETPAGES} \times 100\%$$

Figure 64. Buffer Pool Hit Ratio

In this formula GETPAGES represents *logical I/O*, and PAGES READ, the number of pages *physically* read. The input for the formula comes from a number of counters, for example, from a `-DISPLAY BUFFERPOOL` command:

- GETPAGES (R)
- GETPAGES (S)
- SYNC READ I/O (R)
- SYNC READ I/O (S)
- SEQUENTIAL PREFETCH PAGES READ
- LIST PREFETCH PAGES READ
- DYNAMIC PREFETCH PAGES READ

The outcome of the formula is such that 90% would mean that 90% of the requests for a page were satisfied from the buffers. This is a global hit ratio.

You could also determine a random hit ratio; in the formula you would then include only the GETPAGES (R) and SYNC READ I/O (R) numbers.

You should monitor the hit ratio on a regular basis. Per buffer pool you should establish a target hit ratio. It is best to have at least two different target hit ratios, one for daytime and the other for nighttime. It is possible to monitor hit ratios *systemwide*, with Statistic Reports, or *per application*, with Accounting Reports.

Note: To avoid confusion, in all cases, *always per buffer pool*.

If you observe that the hit ratio slowly drops (from day to day) and threatens to fall below the target, you must do one or more of the following:

- REORG more often
- Spread the workload differently
- Detect and remove unwanted batch-type programs in peak hours
- Reserve more pages for random I/O (set the VPSEQT parameter to a lower value)
- Increase the buffer pool
- Establish more separate buffer pools, perhaps to isolate a problem application
- Review whether the target was reasonable.

The last bullet may seem odd, but it is of course possible that a target was set too ambitiously, and that with a (much) lower target all application requirements would still be met.

6.5 Mapping Spaces to Buffer Pools

The word *spaces* in the heading indicates that we are looking at both table spaces and index spaces. If you consider your present environment too small to bother with this mapping at all, you are probably right. However, both the number of applications using DB2 and the amount of data managed by DB2 will inevitably grow, and you then can use the buffer pool tuning experience you acquired at a time when, strictly speaking, you did not need it.

It is our purpose to end with some practical guidelines for designing, monitoring and adjusting the buffer pool parameters. This topic is only a side topic in this book; you may want look at other publications, such as *IBM DB2: Administration Guide* or *DB2 V3 Performance Topics*.

6.5.1 Buffer Pool Criteria

To make decisions about which buffer pool to use and how many, you can look at several criteria:

- Type of access (random or sequential)
- Size of data (small, medium, or large)
- Variability of data (constant or fluctuating)
- Update rates
- Read rates.

It is a good idea to look at each criterion separately, which we do below. If you try to form all combinations, with the idea of assigning one buffer pool per combination, you will soon find out that it just is not practical. We deal with that in 6.5.2, “Buffer Pools: A Practical Approach” on page 69.

6.5.1.1 Access Types

There are only two types of access to consider: random and sequential.

Random

If your data is accessed randomly, the access path is through one or more indexes. For a unique (or almost unique) SELECT the number of index pages needed is equal to the number of index levels, but we do not want more than one physical read operation; the size of the buffer pool must guarantee that.

Sequential

For sequential access our first thought is a table scan as access path. More often than not however, some order is required, and this order may be achieved through an index; all of its leaf pages will be needed, and this is also a sequential process.

The number of read operations is not of immediate concern, because the I/Os are asynchronous.

Most data has both access types, but not necessarily at the same time. It should be possible to classify each table into one of these groups:

- Strictly random
- Strictly sequential
- Random and sequential, concurrently
- Random and sequential, but never concurrently.

6.5.1.2 Size of the Data

We look at small and large. Of course an endless range of in-between classifications is possible, but it serves no purpose to deal with those specifically; ″small″ and ″large″ are relative terms anyway.

It is necessary to consider indexes and data separately, and we temporarily assume that there is one buffer pool per space.

A small index

You would try to fit a small index entirely into the buffer pool.

A large index

To give you an idea of what a large index means in terms of pages we present an example based on 10,000,000 entries, and six indexes defined with PCTFREE 0 and SUBPAGES 1 (Figure 65).

Index Name	Key Length	Duplicates	Entries per Leaf Page	Entries per Nonleaf Page	Leaf pages	Pages on 2nd Level	Pages on 3rd Level	Pages on 4th Level
X1	8	1	337	368	29674	81	1	-
X2	8	2	368	368	27174	74	1	-
X3	8	10	750	368	13334	37	1	-
X4	30	1	119	123	84034	683	6	1
X5	30	2	184	123	54348	442	4	1
X6	30	10	530	123	18868	154	2	1

Figure 65. Index Sizes

(For the calculation method refer to the *IBM DB2: Administration Guide*)

Some observations on this example:

- Millions of entries require a three- or four-level index.
- More duplicates mean fewer pages, especially with longer keys.
- The number of nonleaf pages is always less than 1% of the number of leaf pages.

For all of these large indexes we want to define a buffer pool size that will support our axiom: *don't count SRs for nonleaf pages.*

At design time we would like to have an easy rule-of-thumb for this. Later, in production, we can always monitor the actual situation and (gradually) increase the buffer pool if needed. In Appendix D, "Calculating the Buffer Pool Size" on page 117 we develop such a rule. The outcome is:

RULE 1

If the buffer pool size is 5 times the number of nonleaf pages, you can safely count on only one physical I/O (for the leaf page).

This rule assumes that accesses are purely randomly spread throughout the index. If this is not the case, you must adjust the rule:

RULE 1A

If the buffer pool size is 5 times the number of ″hot″ nonleaf pages, you can safely count on only one physical I/O (for the leaf page). The hot nonleaf pages are those that are needed in the vast majority of cases.

By ″vast majority,″ you could think of 80%. Of course, these are all attempts for a rule-of-thumb. You could also start with a very small buffer pool and gradually increase it if your hit ratio is too low.

If you want a simple rule based on the number of leaf pages, you can try:

RULE 2

If the buffer pool size is 5% of the number of leaf pages, you can safely count on only one physical I/O (for the leaf page). If the index key length is 15 or less, use **1%** instead.

A small table
You would try to fit a small table entirely into the buffer pool.

A large table
If the pages are accessed randomly, a (very) small buffer pool will do. Suppose you have 50,000 data pages and a 500-page buffer pool. The chance that a page will be reused (a buffer pool hit) is small anyway, and the number of SRs will not be much affected if you lower the buffer pool size to, say, 100 pages. Taken by itself, this is right, but not practical. If a (dedicated) buffer pool is really that small, any sequential access will be less efficient because the *prefetch quantity* is related to the buffer pool size. To get the maximum (32) the buffer pool must be at least 1000 pages. Sequential accesses cannot be excluded; think also of utilities such as COPY and REORG.

6.5.1.3 Variability of Data

All data behaves exactly the same: it is created, updated and deleted. What makes it different is the *timing* of these actions, or if you want, the length of the *life cycle*.

In our case we are interested in the behavior of table volumes. Some typical cases:

- A daily collection table of incoming orders. It begins at zero, grows, is processed during the night, and cleaned.
- Similar in a month or year cycle.
- Work in progress data. The difference here is that cleanup only cleans part of the data (″finished″). So there is always a minimum and a maximum amount of data. The frequency of cleanup can be anything from a day to a year.

The interesting cases are those with a well-defined life cycle. What you must do now is compare this life cycle with your willingness to alter the buffer pool sizes (and other parameters perhaps). If you would consider changing those once a month, you can manage data with a yearly cycle more accurately; you can gradually increase the buffer pool size, demanding more resources only when needed. If, however, your policy is to keep the buffer pools rather fixed, your options are:

1. Design for the maximum (or close to it). If you use this technique for too many tables, and if their yearly cycles are at the same time (for example, December), you would have a terrible waste of memory at the beginning of the year.
   You might also have a problem with end users who, spoiled by superperformance in the first quarter, begin to complain about the slowdown later on.
2. Design for ″average.″ You may now get a ″cramped″ buffer pool at the end of the year, with response times going up and perhaps exceeding alarm limits.

6.5.1.4 Update and Read Rates

Any buffer pool can contain pages of heavily updated page sets, together with read-only page sets. Some of the advanced tuning parameters that you might use may affect your choice of which tables to put together.

The parameters are:

- **Deferred Write Queue Threshold (DWQT)**
  If the number of updated but unwritten pages exceeds this threshold, DB2 starts write I/Os. DWQTH is expressed as a percentage of the entire buffer pool; the default is 50%.
- **Vertical Deferred Write Queue Threshold (VDWQT)**
  If the number of updated but unwritten pages of a single page set exceeds this threshold, DB2 starts write I/Os for this page set. VDWQT is expressed as a percentage of the entire buffer pool; the default is 10%.
- **Sequential Steal Threshold (VPSEQT)**
  Sequential prefetch operations can occupy up to the VPSEQT percentage of the buffer pool; the default is 80%.

You can dynamically alter all parameters with the -ALTER BUFFERPOOL command. Here are some ideas:

- Increase DWQT and VDWQT. This would tell DB2 not to get too ″nervous″ too quickly, and to let page-updates accumulate in the buffer pool. The idea is

that more updates per page get captured, or that more adjacent pages get updated (enhancing the effect of the eventual I/O). For a buffer pool that is dedicated to sort workfiles (DSNDB07), the number of write I/Os might drop significantly, because after a sort is completed, there is no reason at all to write the pages. Their buffer pool slots can immediately be made available for the next process (this of course could never occur with ″normal″ user data).

- Decrease VDWQT. With this parameter DB2 can accumulate lists of updated pages per page set, to make the write I/O more efficient (more pages per VSAM I/O). Now consider a large table space with moderate random update activity. You may get lists of updated pages, but it is likely that the pages are physically distant and the I/Os thus inefficient. Also, it may take a while before the number of pages per data set reaches VDWQT. In that case DB2 will find other reasons to start writing anyway.[29] If so, you might as well let DB2 write the pages almost immediately after update, by lowering VDWQT. This spreads out I/O activity more evenly across the timeline.
- Increase VPSEQT. If you want to make your entire buffer pool available for large prefetch operations, you could set this parameter to 100%. You could do this for sort workfiles, and for user data if you are running utilities. In the latter case, you can reset VPSEQT to 80% after the utilities have finished.
- Decrease VPSEQT. If your buffer pool is designed for random I/Os, you could be worried about sequential processes (typically table scans and nonmatching index scans). Such processes have by default access to 80% of the buffer pool, so your random pages will have less time to stay in the buffer pool. This affects the hit ratio negatively.

  What would happen to sequential I/Os if you decrease VPSEQT? They would still work but would only get access to a small part of the buffer pool. In the ultimate case (VPSEQT 0), a prefetch operation will become a number of synchronous I/Os, for one page at a time and will have a very noticeable effect on response time.

6.5.2 Buffer Pools: A Practical Approach

Having discussed a number of topics in isolation, we must now define some practical guidelines for buffer pools. We propose a number of ″flavors,″ each suited for a certain type of data.

Buffer Pool A - small important indexes

Purpose: guaranteed in-memory processing
Size: the sum of the individual index sizes, plus some reserve
Justification: Designers must justify their request for this buffer pool by showing QUBE calculations that prove that the required performance cannot be obtained otherwise.
Monitor and Control: frequent REORG and RUNSTATS, SQL queries like:

```
SELECT SUM(NLEAF)
  FROM SYSIBM.SYSINDEXES
 WHERE BPOOL = ¢...¢
```

With the Statistic Trace output you can test your objectives: after initial reading, the hit ratio must be 100%. You could even have dummy jobs that

[29] At *checkpoint time*, meaning lots of small I/Os.

are executed right after a DB2 start, just to prime the buffer pool. Your monitored statistics interval should then begin just after these priming jobs.

Buffer Pool B - other indexes

Purpose: keep nonleaf pages in memory
Size: follow Rule 1 (or 1A): 5 * number of (hot) nonleaf pages. There is, however, an important adjustment: you must determine *which indexes are used at the same time*. If, for instance, you have 10 indexes with 20 nonleaf pages each, and 3 of the indexes are used only in some end-of-day process, the buffer pool size can be $5 \times 7 \times 20 = 700$.

You may want a certain minimum, in order to obtain a reasonable *prefetch quantity* (see *IBM DB2: Administration Guide*).
It is also quite possible that you want a certain maximum, either in size or number of indexes, just for administrative reasons (manageability). If you reach that maximum, you assign a new buffer pool and then decide on one of the following strategies:

- Keep indexes of the same application together. This makes it easier to monitor and report *hit ratios* and other numbers.
- Spread indexes randomly over the buffer pools. This should spread out the demand for storage more evenly than the first strategy does.

Justification: none required (natural demand). Actually, the justification works the other way: the system administrator must justify that there is not enough memory (hardware) to satisfy the humble QUBE request of ″no SRs for nonleaf pages.″
Monitor and Control: REORG and RUNSTATS, SQL queries to monitor unexpected growth. The target hit ratio is easily established for random requests on, say, three-level indexes. Two out of three GETPAGES must be found in the buffers, which is 67%. For the hit ratio formula presented in Figure 64 you must use the ″random″ variation. You can do this on both the system level (Statistics) and the application level (Accounting). With the Accounting Reports, you can specifically check whether your estimated number of SRs is not exceeded.
Special settings: you could protect your random pages from being flushed by prefetch scans by setting VPSEQT to a lower value than the default of 80%. If, for instance, you set it to 20% (in a 1,000 page buffer pool) the occasional nonmatching index scan can use no more than 200 pages.

Buffer Pool C - small important tables

This buffer pool is similar to buffer pool A. It seems more convenient to assign a separate buffer pool for this category, but it is certainly not a ″rule″: the small important tables could equally well go together with the small important indexes.

Buffer Pool D - large important tables, random access

Purpose: to enable specific settings for this category and ensure that there is no interference from other tables.
Size: Small, because you expect random requests and are prepared to pay the one SR anyway. Because you also have utilities and batch jobs on these tables, you cannot exclude prefetch. For that, you want efficiency, which means a buffer pool size of 1,000 pages.
Justification: see Purpose.
Monitor and Control: Because the assumption is ″mostly random access,″ you should monitor the number of prefetch requests during peak time.

Special settings: for the batch prefetches, you want to utilize the entire buffer pool and hence set VPSEQT to 100%.

Buffer Pool E - other tables

Purpose: enough for large prefetch operations and to maintain reasonable hit ratios.
Size: as large as you can afford. The maximum size requires expertise from the MVS systems programmer, who knows the overall load on the system, paging rates, and such.
Justification: none required (natural demand).
Monitor and Control: REORG and RUNSTATS, SQL queries to monitor unexpected growth in size.
With the Statistic Trace output you can follow the ratio between Getpage request and Sync. Read I/Os.

Buffer Pool X - system tables

For completeness, with the many buffer pools available, it is a good idea to reserve separate buffer pools for:

- Catalog and Directory (must be BP0)
- Sort-work table spaces.

These buffer pools can then be monitored and tuned separately, which is beyond the scope of this book.
Having these buffer pools is immediately useful if you analyze SQL (through Accounting reports) coming from SPUFI or QMF. These dynamic statements include logical and physical I/O on the Catalog, which you want to see separate from your other I/O.

Chapter 7. Program Coding and Testing

Application programmers deliver functionally correct programs, written in an understandable fashion, and adhering to project standards. This is achieved by:

- Correct reading of the specifications, and translation into statements
- Modular design
- Development of standard algorithms
- Discipline.

Performance issues generally come second. Most programmers, for example, are at best marginally aware of things like:

- Current hardware configuration (CPU, memory, and DASD)
- Operating system concepts such as paging, swapping, CPU load, and priorities
- DB2 concepts such as buffer pools, logging, thread management, and internal sort system
- Physical database design and its (DB2) parameters

because they do not need to know them, and such knowledge would not help them at all in their day-to-day activities.

What then is the role or responsibility of the programmer in the performance issue? In our opinion, a small one. The ever-present ″List of Standards, Tips and Techniques″ should guard against the major errors. A random sample (not only DB2, not only performance-related):

- For array subscripts, use binary declared variables
- Don′t code something inside a loop, if outside will do as well
- Use SYNCPOINTs as soon as possible, also after read
- Use DCLGEN-generated variables
- Always check the result after an executable SQL statement
- If the logic relies on ordering, use `ORDER BY`[30]
- Don′t use `SELECT *`
- Never be clever.

Even where performance is somehow involved, the programmer may not necessarily see it as such. For instance, the argument against using `SELECT *` will usually be a dependency argument, not a performance argument.

7.1 Check the Access Path

So much for the introduction. What about QUBE? The *big advantage* is that the QUBE worksheets of the design stage are available, on paper right on the programmer′s desk. These worksheets will clearly show:

- The tables needed
- The indexes that are expected to be used and how
- Whether sort activity is expected.

Assuming a proper test environment, programmers need only check whether the SQL solution and chosen access path conform to the QUBE-design. If they do,

[30] Every book on RDBMSs should at least once contain this sentence: ″If you want a specific ordering, **tell him.**″ Mission accomplished.

that access-path should also be chosen in the production environment (still to be checked of course, but a different responsibility).

The DB2 EXPLAIN facility checks the access path. We expect most readers to be familiar with the principle: at compile time (BIND) DB2 can produce information about the access path, in the form of one or more rows in the PLAN_TABLE. This table (a regular table) can then be queried. All DML statements are potentially ″explainable.″

Before going on to demonstrate how the check should be done, a brief word on (QUBE) CPU and elapsed time. Frequently, programmers cannot check CPU because they do not have the proper facilities, and for elapsed time the situation is hardly better. Even if they *could* check both, it would be of little value anyway because the test and production environments generally are too different.

7.2 Single Table Access Paths

Below we review most of the examples from Chapter 2, “Applying QUBE to DB2 Access Paths,” showing the corresponding EXPLAIN output. An understanding of the first set, all access paths that involve only one table, is crucial for any programmer.

7.2.1 Table Scan

In Figure 66 you get the first example of how to extend the QUBE worksheet in order to show EXPLAIN output.

″Print all orders″								
INDEX		**T**		**SR**	**TABLE**		**T**	**SR**
					ORDER		100K	1
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
	0	N	0	R	ORDER	NNNN	NNNN	S

Figure 66. Table Scan - EXPLAIN

Note: In Appendix E, “The PLAN_TABLE” on page 119 you can find a VIEW definition that produces similar output, and a short explanation of the meaning of each column.

Not much to tell here. The access-type is *R* (relational scan), almost invariably with PF = S (sequential prefetch).
It is possible, under certain conditions, that parallel I/O is chosen. This is also shown with EXPLAIN. We have not considered parallel I/O in QUBE, so there is no need to discuss it here. Please refer to Chapter 9, “Closing Topics” on page 99 for more on this topic.[31]

[31] Parallel I/O is difficult to demonstrate in the test environment anyway, so it is of little concern to the programmer.

7.2.2 Matching Index Scan

Figure 67 shows the unique selection of one row.

"Retrieve customer data of a given customer (CUSTNO)"								
INDEX	**T**		**SR**		**TABLE**	**T**		**SR**
XCU_1	1		1		CUST	1		1
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
XCU_1	1	N	0	I	CUST	NNNN	NNNN	

Figure 67. Matching Index Scan (One Row) - EXPLAIN

The check now goes like this:

1. **Expected index used?** Yes. If a table scan is chosen, the *primary* check in the test environment is *"Is my table large enough?"*. This applies to all further examples as well. Only when a table really *is* small, such as a currency-code table, you have a small conflict with the QUBE worksheet. The designer has made no error, nor has DB2. The best way to overcome this dilemma is for the database specialist, after the design stage, to deliver a list of tables for which table scans are to be expected and allowed.
2. **Matching columns?** One; this does not prove that there is only one touch, but it *does* prove that the index is used as intended. The proof of one touch is not essential anyway, because that is (functionally) proven by the output of the program.
3. **Sort activity?** No.

A matching index scan can of course retrieve more than one row (Figure 67).

"Retrieve customer data of CUSTNO's above 90000" *(400 hits expected)*								
INDEX	**T**		**SR**		**TABLE**	**T**		**SR**
XCU_1	400		1		CUST	400		1
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
XCU_1	1	N	0	I	CUST	NNNN	NNNN	S

Figure 68. Matching Index Scan (Multiple Rows) - EXPLAIN

It now becomes very clear that EXPLAIN does not show the number of (expected) touches or hits. The only indication is the *prefetch* (PF) column. What can it contain?

- **S** Sequential prefetch is used on either the index leaf pages or the data pages, or both. It should guarantee the QUBE estimate of one SR.
- **L** List Prefetch is used with the RIDs from the index. Thus there is initial overhead for RID fetching and sorting.
- **b** (blank) No prefetch used.

What if EXPLAIN had shown List Prefetch? We decided to ignore List Prefetch in QUBE (see 20), so there is no immediate worry; regard it as an internal technique, apparently thought to be efficient by DB2. It *would* be of great worry if the data had to be retrieved in sequence of CUSTNO, because the EXPLAIN output would now show that you have forgotten an ORDER BY clause. Of lesser worry is the performance argument, namely that *initial overhead* is taken at OPEN

CURSOR time. With the actual estimate of 400 hits it is not dramatic, but if the program's logic is such that not all 400 are (necessarily) retrieved, the OPTIMIZE FOR clause should clearly be considered.

A final (small) point: if PF is L or S , you cannot really see how the index leaf pages are accessed. So you cannot test the QUBE assumption of 1 SR. Do not worry: the possibility that DB2 reads, say, five leaf pages synchronously is offset against the rather likely possibility that some of those five pages are already in the buffer pool (it's an index, after all).

What if EXPLAIN had shown no prefetch at all? Two possibilities:

- DB2 estimated the selected range to be so small that prefetch would make no sense. This in turn indicates that your test data is not representative. If you cannot change that, do not worry. Either the BIND to the production database will give you prefetch again, or otherwise there is still *sequential detection* at run time. In other words, the designer's estimate of 1 SR on the table can never be terribly wrong.
- The table is not as clustered as it should be (according to the Catalog, as always). This indicates an error in the test database that can be easily repaired with REORG and RUNSTATS.

7.2.3 Nonmatching Index Scan

In the next example (Figure 69) we show the nonmatching index scan.

"Retrieve order data of a given ORDER_DATE"
(index on CUSTNO,ORDER_DATE, 50 hits expected)

INDEX	**T**		**SR**		**TABLE**	**T**		**SR**
XOR_COD	100K		1		ORDER	50		50
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
XOR_COD	0	N	0	I	ORDER	NNNN	NNNN	S

Figure 69. Nonmatching Index Scan -EXPLAIN

The information that *matching columns* is 0 means that the entire index is scanned and corresponds to the 100K touches on XOR_COD. As discussed in 2.5.1, "Nonclustered Index (Revisited)" on page 26, it is not easy for the designer to tell in advance whether the index XOR_COD will be used; it depends on the *estimated filtering*. As a consequence, the alternative access path (a table scan) might show up fairly often, certainly in the test environment. This can hardly be avoided but is not serious either. Only in the end, if the index is designed for this program only, should you consider dropping it and reestimating programs that update the ORDER_DATE.

7.2.4 Index-Only Scan

Another basic access path is presented in Figure 70.

"Retrieve ITEMNOs of a given ORDERNO, 4 on average"								
INDEX		**T**		**SR**	**TABLE**		**T**	**SR**
XOI_1		4		1				
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
XOI_1	1	Y	0	I	ORDITEM	NNNN	NNNN	

Figure 70. Index-Only Scan - EXPLAIN

With a simple case as in Figure 70, there should not be a problem. If EXPLAIN shows access of the table as well (XO = N), you have either selected more columns than necessary, or ITEMNO is implemented as a VARCHAR. In the latter case, the QUBE worksheet was wrong from the start. Note that the prefetch column contains a blank. Apparently, DB2 did not think that many leaf pages would be needed. This matches our expectation of four hits.

7.2.5 Clustered and Nonclustered Indexes

All forms of index scans can occur on clustered or nonclustered indexes. What is the effect (see Figure 71)?

"Retrieve orderlines that contain a given ITEM"
(200 hits expected)

INDEX	T	SR	TABLE	T	SR
XOR_2	200	1	ORDITEM	200	1

INDEX	MC	XO	ME	AT	TABLE	SRTN UJOG	SRTC UJOG	PF
XOR_2	1	N	0	I	ORDITEM	NNNN	NNNN	L

Figure 71. Clustered Index Scan - EXPLAIN

Nothing different from earlier examples; it has to be compared with the example in Figure 72.

"Retrieve orderlines of a range of 50 ORDERNO's"
(200 hits expected)

INDEX	T	SR	TABLE	T	SR
XOR_1	200	1	ORDITEM	200	200

INDEX	MC	XO	ME	AT	TABLE	SRTN UJOG	SRTC UJOG	PF
XOR_1	1	N	0	I	ORDITEM	NNNN	NNNN	L

Figure 72. Nonclustered Index Scan - EXPLAIN

If you compare Figures 71 and 72, you can see that EXPLAIN for both is exactly the same. Something can be said on list prefetch:

- **Clustering index scan**: the 200 RIDs from the index should already be ascending. DB2 uses list prefetch because it wants to make sure that there is only one (prefetch) I/O, and that the rows are processed in page sequence.
- **Nonclustering index scan**: the 200 RIDs from the index are random. Naturally, DB2 sorts them to avoid 200 SRs to the data pages. It is likely that the ensuing list (for VSAM) cannot be handled in one I/O because the pages are too distant.[32]

[32] Details not relevant and possibly hardware- or release-dependent.

7.2.6 Multiple Index Scan

The EXPLAIN output becomes lengthier in this example (Figure 73).

"Retrieve orders with a given CUSTNO and ORDER_DATE"
(separate indexes on these columns)

INDEX	T	SR	TABLE	T	SR
XOR_2	50	1			
XOR_OD	300	1			
			ORDER	2	2

INDEX	MC	XO	ME	AT	TABLE	SRTN UJOG	SRTC UJOG	PF
	0	N	0	M	ORDER	NNNN	NNNN	L
XOR_2	1	Y	0	MX	ORDER	NNNN	NNNN	
XOR_OD	1	Y	0	MX	ORDER	NNNN	NNNN	
	0	N	0	MI	ORDER	NNNN	NNNN	

Figure 73. Multiple Index Scan (AND-ing) - EXPLAIN

The four lines of EXPLAIN in Figure 73 are:

1. Start of a multiple index operation, with List Prefetch as the (mandatory) technique
2. First index accessed to retrieve the RIDs (hence, index-only)
3. Second index accessed
4. Building of the final RID list by *intersection* (MI) of the two separate RID lists, and using this final RID list to go to the data pages.

For multiple index scan AND-ing to show in the test environment, you must have enough rows beacuse, if DB2 thinks that by accessing the *most filtering* index the number of RIDs will already be small (for example, 10), it will – rightly – reject the other index and go to the data pages instead.[33]

The order in which the indexes are accessed can be different between the design (QUBE) and real life (EXPLAIN). Is this troublesome? Certainly not right away; the order in which the designer wrote down the indexes in the QUBE worksheet usually has no significance anyway. The order which DB2 chooses the indexes is supposed to be based on *estimated filtering*. In an advanced situation, such as debugging a complex performance problem in production, you may have to get into this, but not at programming and test time.

[33] This is a decision at BIND time; a similar situation may arise at run time, with the consequence that the second index access (as planned) will not occur.

The other case of multiple index scan is presented in Figure 74.

"Retrieve orders with a given CUSTNO or ORDER_DATE" (separate indexes on these columns)								
INDEX		**T**		**SR**	**TABLE**		**T**	**SR**
XOR_2		50		1				
XOR_OD		300		1				
					ORDER		320	320
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
	0	N	0	M	ORDER	NNNN	NNNN	L
XOR_2	1	Y	0	MX	ORDER	NNNN	NNNN	
XOR_OD	1	Y	0	MX	ORDER	NNNN	NNNN	
	0	N	0	MU	ORDER	NNNN	NNNN	

Figure 74. Multiple Index Scan (OR-ing) - EXPLAIN

The difference with Figure 73 is access type *MU* in the last row, indicating *union* of the two RID lists. Here too, you need enough rows in the table, but – in general – not nearly as much as for index AND-ing. What *is* important, is that the conditions are filtering enough; otherwise DB2 will not see much benefit in the method and use a table scan instead.[34]

7.2.7 Sorting

Let us now look at the ordering of rows, or – more generally – any sort process. There are two basic methods, as shown in Figures 75 and 76.

"Print all orders, in sequence of ORDER_DATE"								
INDEX		**T**		**SR**	**TABLE**		**T**	**SR**
					ORDER		100K	1
Sort : NS = 1 NSR = 100K								
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
	0	N	0	R	ORDER	NNNN	NNNN	S
	0	N	3			NNNN	NNYN	

Figure 75. Table Scan with Ordering (Separate Sort) - EXPLAIN

The designer foresaw a sort operation and put an extra row in the QUBE worksheet, so we are not surprised to see an extra row in EXPLAIN in Figure 75. This row has a *method* (ME) of 3.
The sort flags (SRTN) are:

- **U** a sort to make the rows *unique* (remove duplicates)
- **J** a sort needed to assist a *join* operation
- **O** a sort needed for *ordering*
- **G** a sort needed for *grouping* purposes

[34] At run time DB2 might revert to this table scan anyway, if the number of RIDs exceeds a certain threshold. This condition can be monitored through Accounting reports.

There are two such flag sets. For this moment, look only at the right-most set, documented as the sort on the *composite table*.

For the second method it should be obvious that if the rows are retrieved through the index on CUSTNO, they are *automatically sequenced* by CUSTNO. Two further remarks:

- DB2 can now under no circumstances consider List Prefetch on the index.
- For this access path to be chosen, DB2 must still think that the index sort is cheaper than the physical sort. This should in general be true for clustered indexes, which DB2 checks in the Catalog. Later, in production, it is fundamentally important that this clustering is *achieved* too.

"Print all orders, in sequence of CUSTNO"

INDEX	T	SR	TABLE	T	SR
XOR_2	100K	1	ORDER	100K	1

INDEX	MC	XO	ME	AT	TABLE	SRTN UJOG	SRTC UJOG	PF
XOR_2	0	N	0	I	ORDER	NNNN	NNNN	S

Figure 76. Table Scan with Ordering (No Separate Sort) - EXPLAIN

The example in Figure 77 is similar, but with a smaller answer set.

"Show orders of CUSTNO's above 90000, in sequence of CUSTNO"
(400 rows expected)

INDEX	T	SR	TABLE	T	SR
XOR_2	400	1	ORDER	400	1

INDEX	MC	XO	ME	AT	TABLE	SRTN UJOG	SRTC UJOG	PF
XOR_2	1	N	0	I	ORDER	NNNN	NNNN	S

Figure 77. Filtering and Sorting through the Index - EXPLAIN

Please note that the index serves two distinct purposes: to filter and to sort. Also note that programmers *must code* ORDER BY. If they do not (and get away with it), we happily wait for the day when DB2 decides to use list prefetch, or perhaps some form of parallelism.

Continuing the example, it is also possible that EXPLAIN shows something different (Figure 78).

"Show orders of CUSTNO's above 90000, in sequence of CUSTNO" (400 rows expected)								
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
XOR_2	1	N	0	I	ORDER	NNNN	NNNN	L
	0	N	3			NNNN	NNYN	

Figure 78. Separate Sort after Index Filtering - EXPLAIN

This may not please the designer. What went wrong? For some reason, DB2 thought it cheaper to:

1. Obtain the RIDs from the index, through a matching index scan
2. Sort the RIDs in data page order
3. Use list prefetch to get the data
4. Sort the answer set.

This is a valid approach if the index is *nonclustered*. Assuming that the test environment is correctly set up, there are two possibilities:

- The designer has failed to calculate extra sort costs.
- The designer did not want extra sort costs.

In the first case, suggest that the designer read this book, or attend a course on QUBE.
In the second case, check the program logic:

- If *Show* is actually *Display*, and you intend to retrieve only one screen at a time, code:

  ```
  SELECT ...
    FROM ORDER
   WHERE CUSTNO > 90000
   ORDER BY CUSTNO
   OPTIMIZE FOR 1 ROW
  ```

  If the wrong access path still persists, check with your DB2 specialist.

- If *Show* is actually *Print*, check with the designer. There is a possibility (certainly remote in this example) that he or she wanted to avoid a physical sort at all costs because of a systemwide constraint of the DB2 sort system, as implemented by the DB2 system administrator.

The example in Figure 79 eliminates duplicate rows.

"Show the cities where we have customers" (400 cities expected)								
INDEX	**T**		**SR**		**TABLE**	**T**		**SR**
					CUST	2000		1
Sort: NS = 1 NSR = 2000								
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
	0	N	0	R	CUST	NNNN	NNNN	S
	0	N	3			NNNN	YNNN	

Figure 79. Eliminate the Duplicates - EXPLAIN

No doubt, the SQL statement is:

```
SELECT DISTINCT CITY
  FROM CUST
```

The elements in the QUBE worksheet can be found in EXPLAIN.

7.3 Multitable Access Paths

We are now entering the world of joins and subqueries. We remind you that the designer's only interest was to identify which table accesses would be logically sufficient to solve the problem. The more tables get involved, the more SQL solutions exist. It is up to the programmer to code an SQL statement that will satisfy the designer's QUBE worksheet; at the same time, the statement should be understandable.

7.3.1 Joins

There are various join methods in DB2. It is quite understandable that people become confused when they hear about all of these methods. It is our intention to relieve you of this confusion, initially with the following remarks:

1. The more sophisticated DB2 is, the less is the need for you to know the internals.[35] Just trust DB2.
2. Consider only the nested-loop join. It is by far the most common method and generally easy to understand.
3. Designers using QUBE concentrate on the accesses that are logically sufficient to perform the task. In a multitable situation, this is almost automatically what nested-loop join in fact does. So there is a good chance that the QUBE worksheet matches the EXPLAIN output.
4. The vast majority of joins are *equijoins* on primary and/or foreign key columns. Thus in practice there are always indexes to support the join.

In the examples we have more to say. Let us take a look at Figure 80.

[35] Unless you really want to become a *join-specialist*.

"Retrieve order and customer data of a given ORDERNO"								
INDEX		**T**		**SR**	**TABLE**		**T**	**SR**
XOR_1		1		1	ORDER		1	1
XCU_1		1		1	CUST		1	1
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
XOR_1	1	N	0	I	ORDER	NNNN	NNNN	
XCU_1	1	N	1	I	CUST	NNNN	NNNN	

Figure 80. Single Result Row, Two Tables - EXPLAIN

The SQL statement is:

```
SELECT OR.*,CU.*
  FROM ORDER OR,CUST CU
 WHERE OR.CUSTNO  = CU.CUSTNO
   AND OR.ORDERNO = :orderno
```

This is a nested-loop join. What is relevant to know?

- Any join involves two tables. The join method is shown in the row of the second table. A method of *1* indicates a nested-loop.
- The table in the first row is called the outer table. The outer table is scanned once; this can be by a table scan or an index scan.
- The table in the second row is called the inner table. The inner table is scanned as many times as there are qualifying rows in the outer table; therefore, access to the inner table must be efficient, usually an index. DB2 has no objection to a small table scan, however, because the pages are likely to stay in the buffers.[36]
- DB2 always considers both combinations of outer and inner tables. The choice of outer and inner is based on (estimated) cost. The table that contains the fewest rows, after applying the local predicates, usually becomes the outer table. A local predicate is a predicate that involves only the table itself.
  Look at the example: the local predicate is the *"="* for a specific ORDERNO. That reduces the ORDER table to one row, and therefore it is chosen as the outer table.
- Clustering properties on both tables are important. A mismatch between the DB2 catalog information at BIND time and the actual situation at run time can degrade performance dramatically.
- An `ORDER BY` clause may favor a specific choice of outer table.
- A nested-loop join is generally fetch-driven; that is, no initial overhead is taken at `OPEN CURSOR` time. This is true if all sort flags are N and list prefetch is not chosen (and obviously, if a final sort step is not needed for whatever reason).
  It is possible, however, that DB2 cleverly starts sorting the outer table, to achieve a specific sequence of keys, to access the inner table efficiently. In this case, the SRTC-J flag is set to Y (in the PLAN_TABLE row where ME = 1). If needed you can try the `OPTIMIZE FOR 1 ROW` clause to eliminate this particular sort.

[36] This is a point where DB2 looks at the size of the buffer pool.

- Generally, if the designer has written a logical order of table accesses in the QUBE worksheet, the same order can be expected in EXPLAIN. If it is different, do not panic.

Another example in Figure 81, with more rows.

"Retrieve order and customer data of a given CUSTNO"
(50 orders per customer)

INDEX	T	SR	TABLE	T	SR
XCU_1	1	1	CUST	1	1
XOR_2	50	1	ORDER	50	1

INDEX	MC	XO	ME	AT	TABLE	SRTN UJOG	SRTC UJOG	PF
XCU_1	1	N	0	I	CUST	NNNN	NNNN	
XOR_2	1	N	1	I	ORDER	NNNN	NNNN	L

Figure 81. Multiple Result Rows, Two Tables (1) - EXPLAIN

The SQL statement is:

```
SELECT OR.*,CU.*
  FROM ORDER OR,CUST CU
 WHERE OR.CUSTNO = CU.CUSTNO
   AND CU.CUSTNO = :custno
```

Please confirm for yourself that the choice of outer table is obvious. Note once more that DB2 use list prefetch on an index that is already clustered. If you could look at the RID list *before* and *after* the sort, it could well be that both lists are identical.
Now, consider the slight variation in Figure 82.

"Retrieve order and customer data of a given CUSTNO range,
with an ORDER_DATE within the last two months"
(60 customers, 50 orders per customer, of which 2 recent orders)

INDEX	T	SR	TABLE	T	SR
XCU_1	60	1	CUST	60	1
XOR_2	3000	60	ORDER	3000	60

INDEX	MC	XO	ME	AT	TABLE	SRTN UJOG	SRTC UJOG	PF
XCU_1	1	N	0	I	CUST	NNNN	NNNN	
XOR_2	1	N	1	I	ORDER	NNNN	NNNN	

Figure 82. Multiple Result Rows, Two Tables (2) - EXPLAIN

The SQL statement is:

```
SELECT OR.*,CU.*
  FROM ORDER OR,CUST CU
 WHERE OR.CUSTNO = CU.CUSTNO
   AND CU.CUSTNO BETWEEN :cust1 AND :cust2
   AND OR.ORDER_DATE > :orderdate
```

Understand the designer's reasoning:

> "There are two filtering conditions, but since the CUST table is much smaller than the ORDER table, I start by accessing CUST, via the unique index.

That gives me 60 rows. Per CUSTNO I go to the XOR_2 index and find 50 hits (3000 touches on 60 different leaf pages).
With these hits I have to read the ORDER table itself and DB2 will apply the condition on ORDER_DATE. I note that this makes the estimate a bit pessimistic, since of the 3000 touches, relatively few qualify (120).
But, I check CPU and LRT, no problem, so this is it.″

We see clearly that if designers can find a *plausible approach that stays within the alarm limits*, they are satisfied. This is entirely within the philosophy of QUBE. We have never said that designers must consider more than one approach, if the first one is already within limits. They *may* do so, of course.

Note: The keen reader may have observed that the estimate of 60 different leaf pages (XOR_2) is pessimistic, because we are retrieving *ranges* of CUSTNOs. These are on adjacent leaf pages, and at 200 entries per leaf page, you would not expect more than 15 SRs.
The same reasoning applies for the data pages (of ORDER), but the effect is less pronounced because the number of rows per page is only 20, making it 25 SRs. What would likely happen too, on both XOR_2 and OR DER, is *sequential detection*.

In the example in Figure 82, QUBE and EXPLAIN match; could you also get the EXPLAIN of Figure 83?

INDEX	MC	XO	ME	AT	TABLE	SRTN UJOG	SRTC UJOG	PF
XOR_2	1	N	0	I	ORDER	NNNN	NNNN	L
XCU_1	1	N	1	I	CUST	NNNN	NNNN	

Figure 83. Multiple Result Rows, Two Tables (Alternative) - EXPLAIN

Try to follow DB2′s reasoning:

″I use the XOR_2 index to get the 3000 qualifying entries (RIDs). This is efficient, because it′s a matching-index scan. Next, I use list prefetch to access the data pages. Within those pages, I apply the condition on ORDER_DATE, leaving me with 120 CUSTNOs (60 different ones). With those I access the CUST table, through the XCU_1 index of course.″

Well, why not? The point is that the choice of outer and inner table becomes less obvious, both to you and to DB2, if there are local predicates on both tables. A lot depends on *expected filtering* versus *actual filtering*, both as you see it and as DB2 sees it.
If you get the unexpected EXPLAIN (Figure 83), you can have the designer make a new QUBE worksheet. If the designer does not agree with DB2′s reasoning, it becomes a matter of studying the DB2 Catalog and the theory on *filter factors*. Before getting into that, however, it may be better to run the statement in a good test environment and study the DB2 Accounting data.

7.3.2 Correlated Subquery

"Retrieve customers with a given ZIPCODE, who have no orders"
(50 customers per zipcode, final result 5 rows)

INDEX	T	SR	TABLE	T	SR
XCU_ZIP	50	1	CUST	50	50
XOR_2	50	50			

INDEX	MC	XO	ME	AT	TABLE	SRTN UJOG	SRTC UJOG	PF
XCU_ZIP	1	N	0	I	CUST	NNNN	NNNN	
XOR_2	1	Y	0	I	ORDER	NNNN	NNNN	

Figure 84. Correlated Subquery - EXPLAIN

The SQL statement is:

```
SELECT CU.*
  FROM CUST CU
 WHERE ZIPCODE = :zipcode
   AND NOT EXISTS
      (SELECT 1
         FROM ORDER OR
        WHERE OR.CUSTNO = CU.CUSTNO)
```

This time, you cannot see (in our EXPLAIN) whether this statement has been written. The programmer may have coded the subquery in a separate statement (wrongly, but it happens).[37]

7.3.3 Noncorrelated Subquery

In Figure 85 we consider a case where there is a clear mismatch between the QUBE worksheet and the EXPLAIN output.

"Retrieve customer data of those who have placed
at least one order before a given date"
(1000 qualifying orders, there is an index on CUSTNO, ORDER_DATE)

INDEX	T	SR	TABLE	T	SR
			CUST	2000	1
XOR_COD	2000	1			

INDEX	MC	XO	ME	AT	TABLE	SRTN UJOG	SRTC UJOG	PF
	0	N	0	R	CUST	NNNN	NNNN	S
XOR_COD	1	Y	0	I	ORDER	NNNN	NNNN	S
	0	N	3			NNNN	YNYN	

Figure 85. Noncorrelated Subquery - EXPLAIN

[37] It can be seen in the PLAN_TABLE, but you need to include the QUERYNO column in your output.

The SQL statement is:

```
SELECT CU.*
  FROM CUST CU
 WHERE CUSTNO IN
       (SELECT OR.CUSTNO
          FROM ORDER OR
         WHERE ORDER_DATE < :order_date )
```

The QUBE worksheet and EXPLAIN output in Figure 85 do not match. There is an unexpected sort. What is happening?

- **The designer**: ″The CUST table is read sequentially. Per CUSTNO the XOR_COD index is read. This can be done through a matching-index scan. Also, it is only one touch per CUSTNO, because the only test is *whether* a qualifying entry exists (not *how many*).
  Because the CUST table is clustered on CUSTNOs, the access to XOR_COD is in ascending sequence, and DB2 should be able to do some form of prefetch; I count one SR.″
- **The programmer**: ″Let's see. I need customer data of CUSTNOs that exist within the set of CUSTNOs from the ORDER table under a certain condition of ORDER_DATE. That looks like an IN subquery construct.″
- **DB2**: ″By scanning of 2000 XOR_COD entries I get 1000 CUSTNOs. To eliminate duplicates and get them in sequence, I perform an internal sort. The result is a work file of 250, say, different CUSTNOs in ascending sequence.[38]
  Now I access the CUST table, with a table scan. Per row I test whether this CUSTNO is present in the work file. This costs me 125 *ifs* on average.″

  So, what we see in EXPLAIN is in fact reversed: the *second* row is executed *first*. The YNYN sort flags are indicative of a noncorrelated subquery.

Figure 86 shows what DB2 does in QUBE style.

INDEX	T	SR	TABLE	T	SR
XOR_COD	100K	1			
			CUST	2000	1
			WORK	250K	
NS = 1 NSR = 2000					

Figure 86. Noncorrelated Subquery - the DB2 approach

The accesses to WORK are 2000×125 touches. We have not discussed this kind of touch before, but if we assume the normal *touch cost*, it is evident that this is a costly solution. And this is exactly the threat of a noncorrelated subquery: the *repetitive scans of the work file* cost a lot of CPU.

What if the designer *had realized* what the programmer would eventually code? *Should* he have realized it?
According to the QUBE philosophy, *no*. Remember, for a designer, any access path that is logically sufficient is good.

Could the programmer then have written a different statement? Most likely, *yes*:

[38] ″Work file″ means a table in the DSNDB07 database, and therefore possibly on DASD. Small work files stay in the buffers, but to say ″in-memory″ is dangerous, because that phrase has a different meaning in the DB2 manuals.

```
SELECT CU.*
  FROM CUST CU
 WHERE EXISTS
       (SELECT *
          FROM ORDER OR
         WHERE OR.CUSTNO = CU.CUSTNO
           AND ORDER_DATE < :order_date )
```

Figure 87 shows the corresponding EXPLAIN (repeating QUBE for convenience).

INDEX	T		SR		TABLE	T		SR
					CUST	2000		1
XOR_COD	2000		1					
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
	0	N	0	R	CUST	NNNN	NNNN	S
XOR_COD	1	Y	0	I	ORDER	NNNN	NNNN	L

Figure 87. Correlated Subquery as a Solution - EXPLAIN

Now QUBE and EXPLAIN are in harmony again.

For completeness, the programmer could also have written this:

```
SELECT DISTINCT CU.*
  FROM CUST CU, ORDER OR
 WHERE OR.CUSTNO = CU.CUSTNO
   AND ORDER_DATE < :order_date
```

This statement is officially called a *semijoin*, because the second table is not to retrieve data, but to test a condition. As a matter of interest, any "IN subquery" can be rewritten as a semijoin. There are two variations:

- The subquery contains *nonunique values* (our example). You now have to use the DISTINCT keyword in the outer SELECT.
- The subquery contains *unique values*. You can now omit DISTINCT, but it is even better than that. You do not have to bother at all, because DB2 already does the subquery-to-join transformation.

7.3.4 Summary of Subqueries

Some important conclusions emerge:

- Designers must try to think in abstract terms, not SQL statements, of what is needed to solve a particular case.
- The DB2 approach of a *noncorrelated* subquery is:
  - If subquery-to-join transformation is possible, use it.
  - Otherwise, use a work file.

  The subquery-to-join transformation should not worry you. The work file, however, can easily become troublesome. If you want to, you can use a QUBE-style method to show *why* (or to predict *when*) this is the case; see Figure 85. But we do not recommend bothering with such internals, particularly if you are an application designer.
- Programmers must know well how *correlated subqueries* work, often in combination with the EXISTS predicate. DB2 handles these queries very efficiently, and EXPLAIN should match the designers view (which is the QUBE worksheet).

7.3.5 Union

We conclude this chapter with an example of UNION (Figure 88).

"Show current and historical order data of a given CUSTNO;
the ORDERH table is clustered on CUSTNO"
(final ordering on ORDER_DATE)
(20 current orders, 100 historical orders)

INDEX	T	SR	TABLE	T	SR
XOR_2	20	1	ORDER	20	20
XOH_2	100	1	ORDERH	100	1

Sort: NS = 1 NSR = 120

INDEX	MC	XO	ME	AT	TABLE	SRTN UJOG	SRTC UJOG	PF
XOR_2	1	N	0	I	ORDER	NNNN	NNNN	
XOH_2	1	N	0	I	ORDERH	NNNN	NNNN	S
	0	N	3			NNNN	NNYN	

Figure 88. The Use of UNION - EXPLAIN

The SQL statement is:

```
SELECT ORDER.*
  FROM ORDER
 UNION ALL
SELECT ORDERH.*
  FROM ORDERH
 ORDER BY ORDER_DATE
```

Note that if UNION had been written (instead of UNION ALL), the U sort flag would also be Y. In this case though, where a sort was needed anyway, the penalty would have been very small (if at all).

Chapter 8. QUBE and Accounting Trace Data

In this chapter we show how the QUBE numbers can be compared with actual measurements from production. Naturally, we take the production environment, because that is the final test. Measurements from the test environment can be useful to get acquainted with the various reports, but the numbers themselves have only relative value.

8.1 Accounting Reports

DB2 can produce a lot of trace data, which can be transformed into valuable trace information, if you know *where* to look. Otherwise the reports become just a pile of paper with no other purpose than to impress your boss.
We only discuss the accounting trace data. Every installation will know how to gather this data in some way. The bottleneck usually occurs in passing the data (or reports) to the application developers. Therefore, to those: *"See that you get it!"*

A way of passing the data to developers might be DB2 itself. As an example, DB2 Performance Monitor (DB2PM) not only produces a variety of detailed and customized reports, but it also can put accounting data back into DB2 tables. From there it is easily retrievable, in a familiar environment.

The following items can be easily obtained, on the PLAN, PACKAGE, or DBRM level:

- **Times**: Elapsed and CPU. There are several classes:

  CLASS 1 The elapsed time and CPU from the first SQL statement until the end of the accounting interval (loosely, end of program). It includes work in DB2 and in the application.

  CLASS 2 The elapsed time and CPU *within DB2*. CLASS 2 is therefore always lower than CLASS 1. The difference between CLASS 1 and CLASS 2 shows whether a problem is in DB2 or in the application.

  CLASS 3 The elapsed time of the *waits* that occur within DB2. It is always lower than CLASS 2 elapsed time.

- **Counters**: How many occurrences of:
  - Each possible SQL statement
  - Getpages, synchronous I/Os, prefetch requests, and the like, all per buffer pool
  - Lock (and related) requests.

Figure 89 shows what a simple accounting report might look like:

```
PLANNAME: PTY670     #OCCURRENCES: 150

          CLASS 1    CLASS 2    CLASS 3

Elapsed      10          8          5
CPU           4          2        n/a

CLASS 3 Suspensions:     I/O:       4.2
                         Locks:     0.8


                                  BP0    BP1
                                 ----   ----
SELECT   10     GETPAGES        2000      0
OPEN      1     SYNC. IO         150      0
FETCH  1000     ASYNC.IO           2      0
CLOSE     1     UPDATES           80      0
(etc)
```

Figure 89. A Sample Accounting Report

The report shows:

- A PLAN name and the number of occurrences, which means accounting records, which in turn means that all other numbers are averages of 150 executions. Certainly for transactions you would not want to look at a single execution.
- Elapsed time 10 seconds, of which 8 are in DB2. This shows that, if there is a problem, it is somewhere within DB2.
- DB2 time of 8 seconds breaks down into 2 CPU seconds and 5 seconds waiting (4.2 on I/O and 0.8 on locking). The I/O waits are broken down into a number of reasons, not shown here. In the subsequent examples we usually mean synchronous read I/Os (SRs).
- One second is unaccounted for, which is CPU queueing, paging, swapping, or something similar (MVS-related).
- There were 10 ″singleton″-SELECT statements and 1 multirow (cursor) SELECT, issuing 1,000 FETCHes.
- There was buffer activity in BP0. The 2,000 GETPAGES relate to the amount of work (logical I/O). There were 150 synchronous I/Os; if they account for the 4.2 second wait, the average I/O time is 28 ms, which is higher than expected.

  The 80 UPDATES (assuming only SELECT processing) come from sort activity. A common reason is ORDER BY.

8.2 Relate Accounting to QUBE

How does all of this relate to QUBE? In Figure 90 we show the relationship between trace items and QUBE items.

Trace Item	QUBE Item	Remarks
Class 1 elapsed	LRT	Best comparison
Class 1 CPU	CPU	Best comparison
Class 2 elapsed	″LRT″	Usually close to LRT, but it depends on the application type
Class 2 CPU	CPU - x	The number *x* is the application factor in the QUBE formula
CL3 sync I/O waits	SR + L time	The time spent on synchronous I/Os and log writes (per COMMIT); see also related remark below
CL3 async I/O waits	″VP theory″	Waits for prefetch operations
SQL	Related to touches	The executed number of SQL statements and the number of touches are somehow related, because they both express the amount of work.
Getpages	Related to touches	If the number of rows per page is well known, there often is a direct relation.
Sync I/Os	SR	One of the places where *pleasant surprises* occur.
Prefetch I/Os	″SR = 1″	In QUBE you know when you expect prefetch. The number of I/Os is related to the size of the tables, so there is a relation.
Buffer updates	Related to NSR	Only if the DB2 sort system has its own buffer pool. The relation becomes more exact if you know the size of the sorted row, and some internals. Usually too far fetched.

Figure 90. Accounting Data and QUBE Terms

Before continuing, some important remarks:

- Obtaining trace data is not free. For a routine check of overall CPU consumption and elapsed time, CLASS 1 should suffice and incurs very little overhead. You also get the valuable counters.
  The keen specialist wants to have CLASS 2 and CLASS 3. With the advent of Version 3 improvements, the overhead has become much less. You are encouraged to turn on these classes because of the very valuable information.
- The way in which log I/O shows in the accounting data is not straightforward. Please counsult B.5, “Logging Factor (L)” on page 110 for details.
- Separate buffer pools greatly improve understanding of I/O efficiency.
- The prefetch quantity is smaller for smaller buffer pools. Could be important if you try to figure out why so-and-so-many prefetch requests were done.

- If you look at accounting reports from SPUFI or QMF, be aware that the BIND costs are included (CPU, GETPAGES, and possibly synchronous I/O).
- It is very interesting to study accounting reports to compare them with QUBE, but first check with your original assumptions:
  - Is the size of the answer set close to the estimate?
  - Is the number of rows and pages close to the estimate? (This question is particularly important for table scans.)
- You cannot always explain *all* numbers on *every* Accounting Report. If the numbers ″don′t add up,″ you may have to look at a lot of technical information in DB2 and DB2PM manuals, only to find out that it is mostly a matter of definition.

Let us now look again at familiar examples. The QUBE worksheets, already extended with EXPLAIN information, become larger once more.

8.2.1 Matching Index Scan

The values of the few trace items shown in Figure 91 – and in all other examples – are of course purely fictitious. They serve only to help you understand the relationship between trace items and QUBE items. You must think of these values as being measured averages over a reasonable number of executions.

"Retrieve customer data of a given customer (CUSTNO)"								
INDEX	**T**		**SR**		**TABLE**	**T**		**SR**
XCU_1	1		1		CUST	1		1
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
XCU_1	1	N	0	I	CUST	NNNN	NNNN	
Getpage: 3	Two index pages and one data page							
Sync I/O: 2	One index page and one data page. Could well confirm your assumption that the nonleaf pages were in the buffers. Pity you didn't get a *buffer pool hit*.							
SELECT: 1	Under circumstances, perhaps also an OPEN/FETCH/CLOSE cycle.							

Figure 91. Matching Index Scan (One Row) - Accounting

Please note that we do *not* show QUBE CPU and/or LRT, or the corresponding accounting items. The reason is that we either would have to make it up, with the objection that we could make QUBE look as accurate as would please us, or go into a real execution environment, described in full detail, yet still with all possible disclaimers. It would be an interesting exercise, however.

The example in Figure 92 retrieves more rows.

"Retrieve customer data of a CUSTNOs above 90000" *(400 hits expected)*								
INDEX	**T**		**SR**		**TABLE**	**T**		**SR**
XCU_1	400		1		CUST	400		1
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
XCU_1	1	N	0	I	CUST	NNNN	NNNN	S
Getpage: 45	Perhaps 5 index pages and 40 data pages. Separate buffer pools would help, because GETPAGE is shown per buffer pool.							
Sync I/O: 1	A (small) pleasant surprise							
Prefetch: 2	With the usual *prefetch quantity* of 32 pages, this makes sense.							
FETCH: 440	This is important. It should be in the order of magnitude of your *400 hits expected*, otherwise you can't even start to look at CPU and LRT (actual versus QUBE). Note that 440 usually means that 439 rows were found.							

Figure 92. Matching Index Scan (Multiple Rows) - Accounting

8.2.2 Nonmatching Index Scan

The following example (Figure 93) shows a nonmatching index scan.

"Retrieve order data of a given ORDER_DATE" *(index on CUSTNO,ORDER_DATE, 50 hits expected)*								
INDEX	**T**		**SR**		**TABLE**	**T**		**SR**
XOR_COD	100K		1		ORDER	50		50
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
XOR_COD	0	N	0	I	ORDER	NNNN	NNNN	S
Getpage: 560	Mostly index pages							
Sync I/O: 10	A pleasant surprise. You had either buffer pool hits or more than one hit per page.							
Prefetch: 18	All prefetch I/O on the index. At least that is the assumption, because according to QUBE the data pages themselves are read randomly. Some prefetches could have occurred on the data itself, triggered by sequential detection. This event can be seen in normal accounting data.							
FETCH: 51	Now, that was a good design estimate!							

Figure 93. Nonmatching Index Scan - Accounting

If you want to see whether the 100K touches were right, you have to look at the Catalog (after RUNSTATS, please) or do a `SELECT COUNT(*) ..` query.

8.2.3 Clustered and Nonclustered Index Scan

The next example (Figure 94), a seemingly innocent example of a – perhaps slightly – disorganized table, deserves some attention.

"Retrieve customers with a given NAME, 50 hits expected" (clustering changed to NAME)								
INDEX	**T**		**SR**		**TABLE**	**T**		**SR**
XCU_NAME	50		1		CUST	50		1
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
XCU_NAME	1	N	0	I	CUST	NNNN	NNNN	S
Getpage: 18	Too high (3 index pages, 15 data pages). Fifty clustered rows should be on about 5 pages. There must have been some insert or update activity, such that rows could not be placed on their "ideal" page. A revision of the REORG strategy is in order.							
Sync I/O: 3	Not disturbing; you are not yet punished for the disorganized table.							
Prefetch: 2	Normal							
FETCH: 55	According to expectation							

Figure 94. Clustered Index Scan - Accounting

The prefetches are issued because BIND *thought* that the 50 names would be on a few adjacent pages. The RIDs from the index, however, point to some 15 data pages, possibly in random order. In this case we were just lucky to get few SRs: either the prefetched pages were needed anyway, or the requested pages were already in the buffer pool.
In the same scenario, however, it is possible that:

- The prefetch tasks retrieve pages that are hardly needed. This means unnecessary work. In addition, when a GETPAGE request is issued internally, the buffer manager still has to read in the page (because it was not in the prefetch range).
- The prefetched pages are needed but accessed in such random order that one or more have been discarded ("stolen") before being used.

In both cases you get more unexpected SRs.

QUBE promises "pleasant surprises" at run time; let's see (Figure 95).

"Retrieve customers with a given ZIPCODE" (50 hits expected)								
INDEX	**T**		**SR**		**TABLE**	**T**		**SR**
XCU_ZIP	50		1		CUST	50		50
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
XCU_ZIP	1	N	0	I	CUST	NNNN	NNNN	L
Getpage: 42	Perhaps 2 index and 40 data pages							
Sync I/O: 2	A big pleasant surprise (many hits)							
Prefetch: 2	Normal							
FETCH: 51	Perfect							

Figure 95. Nonclustered Index Scan - Accounting

Remember that QUBE simply assumes 50 SRs on the table, only checking whether the table has at least 50 pages (it has 200). From the 40 GETPAGES you can deduce that there was more than one hit per page. With only 200 pages in all, this looks entirely reasonable; you could substantiate this with statistics.
Very few GETPAGES would have meant that there is some clustering on ZIPCODE after all. You might want to analyze whether this is by accident or not.

8.2.4 DB2 Sort

Figure 96 shows an expected (designed) sort, so no worry. As an aside, in your query environent it may well be that too much sorting is one of the main reasons for the high CPU consumption in DB2. If you have such a situation, it may represent a rewarding area for investigation and resolution.

"Print all customers in NAME order"								
INDEX	**T**		**SR**		**TABLE**	**T**		**SR**
					CUST	2000		1
NS = 1 NSR = 2000								
INDEX	**MC**	**XO**	**ME**	**AT**	**TABLE**	**SRTN UJOG**	**SRTC UJOG**	**PF**
	0	N	0	R	CUST	NNNN	NNNN	S
	0	N	3			NNNN	NNYN	
Getpage: 320	Assuming 200 data pages, the other 120 must be sort pages. The fewer columns are selected (no SELECT *, please), the fewer sort pages are required.							
Sync I/O: 1	Normal							
Prefetch: 12	These include prefetch on the data and on the sort pages. If you need the distinction you need separate buffer pools or a performance trace (on I/O events).							
Updates: 120	For sorts, the buffer-update counter is incremented once per sort page. For SQL-related updates, it is once per row.							
FETCH: 2102	Normal							

Figure 96. Table Scan With Sort - Accounting

Chapter 9. Closing Topics

In this chapter we look at several QUBE-related topics about which you already may have wondered.

9.1 Release Dependency

We trust the DB2 designers to come up with performance improvements with virtually every release. Many of these improvements will not immediately affect the technical designer. For instance, there could be some global improvement that all programs run 10% faster. Because QUBE is not designed for utmost accuracy, we do not mind. If we want, we can change a constant in a formula, but there is no hurry.

In particular a good DB2 specialist must resist the temptation to review all QUBE issues with the advent of a new release. Stay calm, wait until everything is settled down and there is a clear view of where and how a specific improvement affects the *application designer*.

9.2 DB2 Features

We discuss several features that are available (or announced) in DB2. Must you adjust QUBE for them? By this time, it is evident that we would answer ″no,″ but it is also evident that those who have worked themselves through all previous chapters might answer ″yes, but only by a specialist.″

Compression
Compression of data was possible before, but here we look at the COMPRESS YES clause of CREATE TABLESPACE, as delivered with Version 3. This compression works in all hardware environments. If the data compression hardware feature is installed (on certain models), the CPU overhead is small indeed. Otherwise, a software emulation is performed, which costs about five times more. A number of measurements are described in *DB2 V3 Performance Topics*. Some conclusions are:

- The *encode* costs are about five times higher than the *decode* costs.
- In an OLTP environment a 1%-5% increase in CPU can be expected, depending on the hardware or software solution. This is a rough guideline, and includes encode and decode costs.

From this we would conclude that for SELECT processing there is no reason to adjust QUBE. For INSERT or UPDATE you might use a different *touch cost*, but only for data-page touches; indexes are not compressed. The increased complexity in QUBE is worthwhile only if you hide this complexity in a spreadsheet or other tool, and on a *per table* basis.

Parallel I/O
From Version 3 parallel I/O is possible for certain queries on a table in a partitioned table space. In QUBE terminology the likely candidates are the VP table scans. As we have demonstrated, these can become I/O bound (″prefetch not fast enough″). Those cases, certainly for the very large tables that are in partitioned table spaces, could be reviewed by a specialist anyway. If so, this specialist could consider the effects of parallel I/O.

Parallel CPU
From Version 4 parallel CPU is possible in certain situations. Similar to parallel I/O, we currently tend to see the consideration of parallel CPU as something for the specialists, and therefore not immediately affecting QUBE.

Dynamic SQL
You can adopt a QUBE style approach for the BIND costs of dynamic SQL. As BIND is a process on data in the Catalog, it consists of *touches* and *synchronous reads*. Some experiments are needed to establish the numbers; at present the author is thinking in terms of ″10 touches and 4 SRs per table in the statement.″ The complexity of the statement would not play a part (″average query″). In a dedicated query environment we suggest minimizing I/O on the Catalog, in other words, have enough BP0; if so, the number of estimated SRs can be lowered.

DDL and DCL
All statements in this category could be QUBEd in the normal way. For this, consider the DB2 Catalog to be a normal collection of tables, indexes, and table spaces, and break down each statement into elementary INSERT, UPDATE, or DELETE parts (see Chapter 3, “Applying QUBE to Updates”). For the most part, we would not see much point in this. An exception could be the (repeated) creation and dropping of work tables from within a program. If you embark on ″DDL-QUBE,″ keep the following points in mind:

- There are special clustering schemes in the DB2 Catalog. For instance, a CREATE TABLE statement inserts at least one row in SYSIBM.SYSTABLES and *n* rows in SYSIBM.SYSCOLUMNS. For QUBE that would be *n + 1* touches, but only one SR, because the *n + 1* rows are clustered on the same 4K page.
- If your DDL statement internally creates or deletes a physical object (VSAM: DELETE/DEFINE CLUSTER), there may be substantial extra elapsed time. This is purely wait time, accounted as Service Task Suspensions in Accounting Class 3 data.
- The cascading effects of DROP <object> can go further than you think!

Version 4 Index Manager
A very important feature of Version 4 is the new Index Manager. It means:

- No locking on indexes
- No SUBPAGES
- Better handling of RID-chains
- Full partition-independence
- Support for .. UNIQUE WHERE NOT NULL indexes
- Support for row-locking.

Obviously very interesting for a DB2 specialist. For QUBE, it might mean that the cost for an index touch becomes so much lower than that for a table touch, that you would be tempted to make a difference. This still has to be investigated, however; also, all of your indexes would have to be TYPE 2 indexes first.

Locking
The conceiver of QUBE, Tapio Lahdenmäki of Finland, has recently written a locking addendum for the QUBE courses that looks at the impact of the Version 4 improvements. The general line of thought, if you want to consider locking in QUBE, is this:

1. Establish LRT (as usual).

2. Assume that LRT is also a COMMIT scope. This means that the locks are kept for this period.
3. Set a time limit for a single page to be locked (say, 5 s).
4. If LRT exceeds this time limit, you have a problem. You must change the logic, lower the LRT, or COMMIT more often.
5. If any page is locked for more than 10% of the time, it is a "hot page" and deserves extra attention. Whether you have such pages can be established by the formula:

$$\text{LockedPercentage} = \frac{100 \times PL \times TR \times LD}{PT}$$

PL The number of pages locked by the transaction
TR The number of transactions per second
LD The lock duration (commit scope) in seconds
PT The number of pages over which the touches are spread

The formula is assessed per page set, indexes first. If the outcome exceeds 10%, you have a hot page and a possible problem.

In Figure 97 suppose that, for whatever reason, all of the transactions are concentrated on a limited number of ITEMs, contained in two index leaf pages.

"Update ITEMs with price information"
(1 ITEM per transaction, 5 transactions/second)

INDEX	T	SR	TABLE	T	SR
XIT_1	1	1	ITEM	1	1
CPU = 3.4 ms		LRT (CICS) = 95 ms			

Figure 97. A Possible Locking Problem

If you apply the formula:

$$\text{LockedPercentage} = \frac{100 \times 1 \times 5 \times 0.1}{2} = \mathbf{25\%}$$

the outcome exceeds 10%, so, there are hot pages in the XIT_1 index. The most likely solutions are to:

- Enhance your DB2 logging. The 95 ms LRT contains 50 ms for logging.
- Spread out the rows over more index pages, for example, five, to give you an outcome of 10%.

It is possible that a future version of this book will elaborate on this topic. It is also possible that by that time, Version 4 will have relieved us of so many locking problems that nobody cares.

Utilities
Somewhere during design and build it may be necessary to say something about utilities, not only if they are used for their housekeeping purposes, but also if they play a part in some application process.

An estimate of utility CPU and elapsed time can be found in various sources, for example, *DB2 V2.3 Nondistributed Performance Topics*, with some adjustments provided in *DB2 V3 Performance Topics*.

The IBM product, DB2 Estimator, also provides good estimates; a quick one requiring little input, or an advanced one. See also 9.4, “DB2 Estimator” below.

9.3 Other DB2 Environments

QUBE, being a method (or an approach), should be applicable in other DB2 environments as well. The most likely platform is DB2 for OS/2 or DB2 for AIX, Version 2. These products (actually almost one product) have similarity in general principles and access paths with DB2 on MVS. For a structured debate on such topics as which indexes and which clustering to use, QUBE is always useful, even without having exactly the right formulae. All formula factors, as discussed in Appendix B, “The QUBE Constants” on page 107, can in principle be determined by a workstation specialist.

9.4 DB2 Estimator

In June 1994, IBM announced DB2 Estimator (5622-539), a PS/2-based tool for OS/2 or Windows that provides performance estimates on SQL, transactions, and utilities. How does this product relate to QUBE?

DB2 Estimator can give very accurate estimates; it is obvious that to get those, you may have to feed it with accurate and detailed input. DB2 Estimator by the way does have a facility that makes it possible to quickly define tables from scratch (without having to give details on all columns).
QUBE on the other hand, in a way gives an estimate, but this estimate is actually used to raise a trigger (*”is my alarm limit exceeded?”*). If there is no trigger, so much the better; if there is one, you look for design alternatives. QUBE can be used very early, and with very sketchy input.

From this it follows that you will probably not use DB2 Estimator for all your SQL, but only for critical ones. QUBE can be used for all your SQL.
DB2 Estimator is a very *valuable tool* for an expert to do an expert calculation. QUBE is a *valuable method* for any application developer, from the early design stage to production monitoring and tuning. It is a way of thinking.

9.5 How to Introduce QUBE?

For QUBE to be successful in your projects, management understanding is important. For that group we repeat the purpose and value of the method:

What is QUBE?

Quick Upper Bound Estimate is one single methodology, to be used from the earliest design stages to production monitoring. It is used by different people and provides a common method for handling otherwise complex issues. It provides real assistance in taking the right decisions at the right time.
QUBE itself is not complex and highly satisfactory. In just a few days people get convinced that performance issues are not necessarily complex, and that it is possible to adjust a design before you build, rather than to repair something after taking it into production.

On a project team, in our opinion about 1 per every 10 people should be a QUBE specialist; the other 9 can apply the method in the vast majority of cases and know when to seek advice from the specialist.

Appendix A. The Order-Entry Database

Figure 98 shows a *Version 0* order-entry database.
All tables have been given a short two-character name: CU, OR, OI and IT. These names appear in the name of the indexes. The suffix _1 is always for a primary key, and _2 for a foreign key (_n if more than one foreign key).

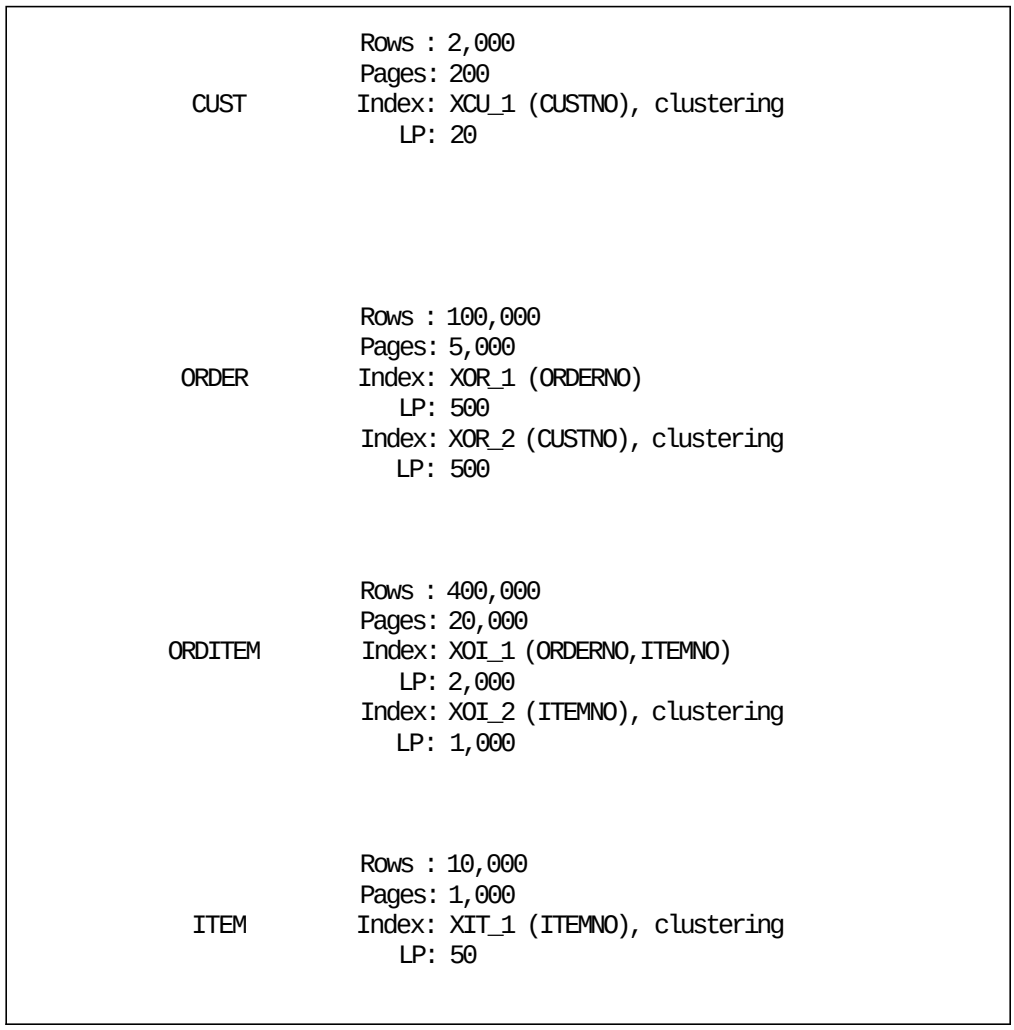


Figure 98. The Order-Entry Database

The clustering is an initial one (assumed), but it does not show in the names of the indexes. Although such a naming convention would be useful, certainly in understanding some of the more complex examples, you would have to change the index name whenever you change clustering.

In some of the examples extra indexes are used, with names like XCU_ZIP, indicating an index on CUST (ZIPCODE).

Appendix B. The QUBE Constants

Figure 99 shows the general format of the QUBE formulae.

$$\textbf{CPU} = M \times (A + (\mathrm{T} \times 0.2) + (\mathrm{NS} \times 1) + (\mathrm{NSR} \times 0.06))$$

$$\textbf{LRT} = \mathrm{CPU} + (\mathrm{CPU} \times Q) + (\mathrm{SR} \times D) + L$$

Figure 99. The QUBE Formulae

It is important that you adjust the constants of the formulae to your specific environment. Before we explain how to do that, we want you to consider the following.

- Developing the constants is a specialist's job. An absolute requirement is a knowledgeable MVS system programmer, who can be assisted by a DB2 specialist (the SYSADM).
- What if the constants are wrong?
  That's easy: your QUBE outcome will be wrong too. But, is that always a concern? It is important to understand that QUBE is primarily intended to check *whether* an alarm limit is exceeded, but not to answer the question *"by how much?"*
  And, if you are comparing different solutions for a problem, QUBE still has *relative value*, even if the *absolute answers* are way out of range.
- In the early design phase you still have some time before the application goes into production; in a major project this can easily be more than a year. So if you use QUBE, the constants should in fact reflect the situation *as it will be at production time*. Whether you succeed in this will depend on your system programmer's ability to extrapolate a variety of (complex) input.
- The constants should be reviewed for correctness after any major changes (usually hardware), and at least once a year.
- No matter how skillful you are in developing the QUBE constants, the outcome of any QUBE formula is still a QUBE outcome. It cannot be used for absolute correctness, benchmarking, or capacity planning.

In general you can set the constants in three ways:

- Use technical information that is supplied with relevant hardware and software.
- Use measurement tools outside DB2.
- Use measurement tools inside DB2.

Not all methods may be applicable for all constants.

B.1 Machine Factor (M)

Figure 100 shows all of the *M* values for the different processors.

4381-90E : 17.33 4381-91E : 13.87 4381-92E : 14.51	3090-120E : 8.21 3090-150E : 6.18 3090-180E : 3.80 3090-200E : 4.01 3090-280E : 4.16 3090-300E : 4.28 3090-400E : 4.50 3090-500E : 4.74 3090-600E : 5.08 3090-100S : 11.56 3090-120S : 8.21 3090-150S : 5.24 3090-170S : 4.22 3090-180S : 2.93 3090-200S : 3.08 3090-250S : 5.52 3090-280S : 3.18 3090-300S : 3.24 3090-380S : 3.35 3090-400S : 3.31 3090-500S : 3.46 3090-600S : 3.63 3090-110J : 8.21 3090-120J : 6.50 3090-150J : 4.66 3090-170J : 3.80 3090-180J : 2.66 3090-200J : 2.79 3090-250J : 4.99 3090-280J : 2.86 3090-300J : 2.89 3090-380J : 2.98 3090-400J : 2.98 3090-500J : 3.09 3090-600J : 3.22 3090-15T : 4.16 3090-17T : 3.45 3090-18T : 2.66 3090-25T : 4.39 3090-28T : 2.86	9121-180 : 11.35 9121-190 : 7.90 9121-210 : 5.47 9121-260 : 3.97 9121-320 : 3.09 9121-440 : 4.13 9121-480 : 3.28 9121-490 : 3.28 9121-570 : 3.37 9121-610 : 3.47 9121-311 : 3.12 9121-411 : 2.61 9121-511 : 2.03 9121-521 : 2.71 9121-621 : 2.16 9121-622 : 2.15 9121-732 : 2.25 9121-742 : 2.35	9021-330 : 2.93 9021-340 : 2.66 9021-500 : 2.79 9021-580 : 2.89 9021-620 : 2.98 9021-720 : 3.23 9021-520 : 1.32 9021-640 : 1.41 9021-660 : 1.37 9021-740 : 1.49 9021-820 : 1.49 9021-860 : 1.54 9021-900 : 1.60 9021-711 : 1.00 9021-821 : 1.05 9021-822 : 1.05 9021-831 : 1.12 9021-941 : 1.18 9021-942 : 1.15 9021-952 : 1.21 9021-962 : 1.25 9021-972 : 1.29 9021-982 : 1.32

Figure 100. M Values for Different Processors

If you compare models that differ only in the number of processors, for instance 3090-400J and 3090-600J, you will see that with more processors the *M* factor is slightly higher. Does this mean that if you install an extra processor, your response time will be *worse?*
No, of course not; if you adjust your QUBE formula, the CPU number will increase – although in most cases the difference is negligible – but you will get a decreasing *Q* factor. There are more processors, so less CPU contention. The *M* factor is slightly higher because MVS needs a little more time to manage the additional processor. Therefore the net amount of CPU available to applications decreases.

A slight omission in the QUBE formula for CPU is the notion of the Sort Assist hardware feature. This feature is available on a number of processors and reduces the sort costs by 50%. So you must write:
(... + (*NS* × 0.5) + (*NSR* × 0.03))

B.2 Application Factor (A)

This is an admittedly ″loose″ number. It is there to remind you that:

- Initialization is not free.
- COMMIT is not free.

In a CICS or IMS environment the application factor would include the transaction monitor overhead. For COMMIT, the reasoning is that DB2 takes a bit of CPU, and the application might too, perhaps for restart logic.

The application factor is *not* for normal application logic, which is assumed to take only *microseconds* and hence is irrelevant.

The suggested number of 3 ms (still to be multiplied by the machine factor - M) is a starting point.

B.3 Queueing Factor (Q)

The amount of CPU available to you depends on:

- MVS priority settings (SYS1.PARMLIB)
- Amount of CPU that you need
- Current CPU load.

In the end we want different factors for such situations as:

- CICS or IMS transactions during the day
- TSO or QMF during the day
- Batch or utilities during the day
- Batch or utilities at night.

The MVS system programmer should be able to provide numbers, based on RMF reports, but these numbers only show how good or bad the situation is today. QUBE actually works the other way; it assumes a ″reasonable″ environment, which could be for instance:

- Less than 50% CPU queueing for transactions
- Less than 300% CPU queueing otherwise.

If a reasonable environment is not achieved, it is an MVS problem, not your problem. The role of the MVS system programmer is therefore to ensure that the reasonable environment is always there.

With purely DB2 means, we can also guess what the queueing factor is, namely, by looking at the ″unaccounted″ time in a DB2 accounting report. Please refer to Chapter 8, “QUBE and Accounting Trace Data.” If you do that, you must take care not to look at a single accounting report, but at averages over longer and significant periods.

B.4 DASD I/O Factor (D)

In principle, you should be able to obtain the DASD I/O factor via an ″RMF DASD Activity Report.″ You can discern between ″day″ and ″night″ response time, but this is probably overdoing things.

As we are interested here in the I/O time for a 4K (or 32K) page, you must be aware that there are other I/Os as well. In a DB2 environment, there is a mix of synchronous I/Os, list prefetch, sequential prefetch, and write I/Os. From a

device point of view, and therefore also on RMF reports, all of these I/Os are one I/O at a time. For prefetch, a typical time would be 50 to 70 ms (128K). If such times are included in reported averages, the result is clearly misleading.
It is therefore better to look at *device utilization*, also reported in RMF. A typical synchronous I/O on a 3390 device, with a measured elapsed time of 20 ms, has the following elements:

```
Device queueing : 4 ms
Seek            : 8
Rotational delay: 7
Transfer        : 1
```

The critical figure here is *device queueing*: the more requests there are (for this device), the longer the queueing time. The example assumes a *device utilization* of 20%, as is apparent from the (standard) queueing formula in Figure 101.

$$\text{Time}_{wait} = \left(\frac{\text{util\%}}{100 - \text{util\%}} \right) \times \text{Time}_{service}$$

or:

$$\text{Time}_{wait} = \left(\frac{20}{100 - 20} \right) \times 16 = \mathbf{4\ ms}$$

Figure 101. Device Queueing Time

You can use this theory in establishing your QUBE *D* factor. It is better, and quite similar to our remarks on the *Q* factor, that you *assume* (or demand) 20 ms, leaving it to production monitoring and tuning to *achieve* it. If it then turns out that the utilization is 30%, it would mean a queueing time of 7 ms, and hence (on average) an I/O elapsed time of 23 ms.

It is also possible, and perhaps easier, to obtain your I/O times from DB2 accounting reports. These should be taken in a relevant period, for example, from 9:00 to 16:00. Be aware that this technique requires accounting trace CLASS 3 to be active. The relevant counter is CLASS 3 SYNC.IO SUSPENSIONS Because that counter might include other events as well, you must isolate a case where CLASS 3 SYNC.IO SUSPENSIONS is equal to SYNCHRONOUS READS in the buffer pool statistics.

Finally, you can also use the -DISPLAY BUFFERPOOL command and the AVERAGE DELAY and MAXIMUM DELAY numbers.

B.5 Logging Factor (L)

Tuning of the log system is crucial in any nontrivial environment. A little explanation of the log system might help here.

Any data change is reported in a log record. These records are moved into *log buffers*, which are configured outside the normal buffers.

For the most part, log buffers are written when:

- An application commits

  or

- The amount of unwritten log buffers exceeds a system threshold.

You must therefore reckon on physical I/O time at COMMIT time, but this is almost unrelated to how many changes *you* made. Most of your changes are already physically written to the log by another application (that committed) or by the system trigger.

What remains is typically one I/O per log file, or in a *two-phase commit* scenario, two I/Os. Since most installations have *dual logging*, you must still multiply the number of I/Os by 2.

If the log files are on a device with 20 ms elapsed time per I/O, your *L* factor could be as high as 80 ms. Currently the best support for fast logging is a 3990 controller with *fast write*; the elapsed time can then be as low as 2 ms per I/O. This is so low that in fact you could ignore it, but we suggest that it is better to have something in your formula than nothing at all, lest you forget the log aspect.

The logging time can also be seen on accounting reports, but you must be aware of some release and environment differences:

- In DB2 Version 3, the event counter for CLASS 3 SYNC.IO is incremented per COMMIT. The corresponding timing counter is incremented with the total wait time for the Log Manager. You cannot see how many I/Os the Log Manager actually needed. What you can see is that the wait time in a dual logging environment is about twice the amount of a single logging environment.
- In DB2 Version 4, the event counter for CLASS 3 SERV.TASK SWITCH is incremented per COMMIT. The corresponding timing counter is incremented in the same way as above.

  However, the exception is that commit phase-1 processing from any two-phase commit protocol is still accounted as CLASS 3 SYNC.IO, as in Version 3.
- In all circumstances, you must be aware that the aforementioned CLASS 3 fields are also used to monitor other events. Only if you can exclude those events, can you deduce log waits from Accounting reports.

Appendix C. Using a Specialist Formula

This appendix explains how an existing *specialist formula* can be used to derive a (simplified) QUBE formula. The example is for the very pessimistic table scans and nonmatching index scans. Please refer to 2.5, “Very Pessimistic Cases” on page 24 for the introduction to and outcome of this approach.

Specialist formulae can be found in various publications. The formula in Figure 102 is taken from *DB2 V2.3 Nondistributed Performance Topics*.

0.11 * NP	Page component
+ 0.33 * NIO	CPU time for I/O
+ 0.43	Fixed overhead
+(NDMS * (0.005 + (0.0033 * NSAN)))	DM component
+(NQ * (0.04 + (0.0015 * NC)))	RDS component
+(NF * (0.12 + (0.0015 * NC)))	Fetch component

Figure 102. Specialist CPU Formula for Multiple Rows

Note: This formula is in fact a combination of two formulae, one for a table scan, one for an index scan. It assumes a model 3090-180J; this is different from all of our previous QUBE examples, which assume a 9021-711. The fact that the formula is based on V2.3 is not relevant for present purposes.

The formula is not be to used ″as is″; it serves as a basis for simplification. Using the specialist formula itself should be done only after studying the original publication.

The constants have the following meaning:

NP Scanned leaf and data pages

NIO CPU cost per page in case of a physical I/O. Use NP/5 for all types of prefetch and use 1 for each SR

NDMS Number of rows scanned by Data Manager *after* applying the index predicates. For a table scan, NDMS is equal to the number of table rows.

NSAN Number of search arguments, excluding matching index predicates

NQ Number of qualifying rows after passing the Data Manager

NC Number of columns

NF Number of fetches

It should be obvious that, even if you hide the formula in a spreadsheet, it still would have too many parameters to make it practical for a technical designer. So we will take a ″QUBE-ish″ approach to simplify.

C.1 Very Pessimistic Table Scans

For a table scan we assume:

- Number of rows per page is 20, so NP=0.05*T
- NIO = NP/5, so NIO=0.01*T
- NDMS=T
- NSAN=3
- Number of qualifying rows is equal to number of fetched rows, which means that there are no ″stage 2″ predicates, NQ=NF
- NC=10.

If we substitute these we get the formula shown in Figure 103.

0.0055 * T	Page component
+ 0.0033 * T	CPU time for I/O
+ 0.43	Fixed overhead
+ (T * (0.005 + 0.0099))	DM component
+ (NF * (0.04 + 0.015))	RDS component
+ (NF * (0.12 + 0.015))	Fetch component

Figure 103. Simplified Specialist Formula for Table Scan

After some reduction and rounding we end with the formula in Figure 104.

$$\textbf{CPU} = (T \times 0.02) + (NF \times 0.3)\ \text{ms} \qquad [A]$$

Figure 104. Simplified QUBE Formula (Table Scan)

Note that we have discarded the ″fixed overhead,″ because it is negligible for the table scans that we have in mind.

Compare the simplified formula with our original formula (see also Figure 7 on page 11):

$$\textbf{CPU (QUBE)} = M \times (T \times 0.2) = (T \times 0.532)\ \text{ms} \qquad [B]$$

The *M* factor for a 3090-180J is 2.66. So now we can make some comparisons (Figure 105).

Touches	Fetches	CPU s [A]	CPU s [B]
1000	100	0.05	0.53
1000	10	0.02	0.53
10000	100	0.23	5.32
10000	10	0.20	5.32
100000	100	2.00	53.2
100000	10	2.00	53.2

Figure 105. CPU Formula Comparisons (Very Pessimistic Table Scans)

Based on these comparisons we could say that a revised QUBE formula should result in a 10 to 25 times reduction in CPU.
Before we go on to actually construct such a formula, we must repeat the process for a nonmatching index scan.

C.2 Very Pessimistic Nonmatching Index Scan

Taking a similar approach:

- Number of rows per leaf page is 200, so NP=(0.005*T) + NF
- NIO = (prefetch for leaf pages) + (NF), so NIO=(0.001*T) + NF
- NDMS=NF
- NSAN=3
- Number of qualifying rows is equal to number of fetched rows, which means that there are no ″stage 2″ predicates, NQ=NF
- NC=10.

After substitution (Figure 106):

(0.11 * NF) + (0.00055 * T)	Page component
+ (0.33 * NF) + (0.00033 * T)	CPU time for I/O
+ 0.43	Fixed overhead
+ (NF * (0.005 + 0.0099))	DM component
+ (NF * (0.04 + 0.015))	RDS component
+ (NF * (0.12 + 0.015))	Fetch component

Figure 106. Simplified Specialist Formula for Nonmatching Index Scan

Yielding (Figure 107):

$$\textbf{CPU} = (T \times 0.009) + (NF \times 0.64)\ \text{ms} \qquad [A]$$

Figure 107. Simplified QUBE Formula (Nonmatching Index Scan)

And again comparing the two (Figure 108):

Touches	Fetches	CPU s [A]	CPU s [B]
1000	100	0.07	0.53
1000	10	0.02	0.53
10000	100	0.15	5.32
10000	10	0.10	5.32
100000	100	0.96	53.2
100000	10	0.91	53.2

Figure 108. CPU Formula Comparisons (Very Pessimistic Nonmatching Index Scans)

The difference between (A) and (B) is more pronounced now. For the interesting cases (many rows) it is a factor of 50. The reason for this, by the way, is that in standard QUBE the touch cost for an index row is probably too high anyway.

We are now ready to propose a revised QUBE formula. Before doing so we remark that although the method of starting with a specialist formula may have impressed some readers as ″accurate″ or ″scientific,″ it is not so at all. It just takes some shortcuts to arrive at a QUBE style solution for some known ″very pessimistic″ cases. Or, if you want, a rather loose method to get an order-of-magnitude feeling.

C.3 Very Pessimistic, Revised QUBE Formula

Because our standard QUBE formula does not contain a fetch factor, we do not want to introduce one now; besides, if you would carefully review the last few pages, you would probably agree that the fetch factor is not that important, *under the condition that we are dealing with ″very few″ fetches in a ″nontrivial″ table*.

So in the original format, we propose a change (Figure 109):

$$\textbf{CPU} = M \times (A + (T \times TC) + NS + (NSR \times 0.06))\ \text{ms}$$

Figure 109. New QUBE Formula for CPU

For the new element, *TC*, or touch cost use:

0.2 For normal cases
0.01 For ″very pessimistic″ table scans
0.004 For ″very pessimistic″ nonmatching index scans

To simplify matters even further, we eliminate the difference between the last two cases, and use $TC = 0.01$ for all very pessimistic cases.

Appendix D. Calculating the Buffer Pool Size

We demonstrate how the size of a buffer pool can be determined with a statistical approach. We are interested in the size that a (dedicated) buffer pool should have to satisfy the QUBE axiom, *"don't count SRs for nonleaf pages"*.

It is best to look at an example. For this we use an imaginary index X1, a three-level index with one root page, six intermediate pages, and a large number of leaf pages.

The following notation is used:

R The root page
Nn With *n* between 1 and 6, one of the nonleaf pages[39]
Lx An unspecified leaf page

Assume a buffer pool of *m* pages and that we already have had initial random requests, enough to fill the buffer pool. Next, assume that the last six requests needed all 6 *N* pages. The *LRU chain* at this point is that shown in Figure 110.

```
LRU : (m-13 Lx¢s),N1,Lx,N2,Lx,N3,Lx,N4,Lx,N5,Lx,R,N6,Lx
```

Figure 110. An LRU Chain

The last 13 pages ("most recently used") are spelled out; the "front" of the LRU chain therefore consists of $m-13$ leaf pages.

If the next cycle is (R,N4,Lx) the LRU chain becomes that shown in Figure 111.

```
LRU : (m-12 Lx¢s),N1,Lx,N2,Lx,N3,Lx,Lx,N5,Lx,N6,Lx,R,N4,Lx
```

Figure 111. An LRU Chain, Next Cycle

Please check the correctness of this before reading on. If you look at the position of the N1 page, you will see that *per cycle it advances one page* toward the "top" of the LRU chain.

Going back to Figure 110, the question now is: *"Which value must* m *have, so that the chance of N1 becoming the LRU page is very small?"*

This question can also be expressed as the probability that N1 is not accessed in *c* cycles, which is:

$$Pr_{NotAccessed} = \left(\frac{5}{6}\right)^{c}$$

In statistics it is not unusual to define a "small chance" as 0.05 (or 5%). On that assumption, *c* must be 17, giving us a buffer pool size of 17 + 13, or 30 pages.

[39] Nonleaf in fact includes the root page, but the letter *N* is much better than the *I* of *Intermediate*, for readability.

For a general case you must realize that '13' in the example is equal to $(N \times 2) + 1$. Therefore the general formula becomes as shown in Figure 112:

$$\text{Bufferpool size} = c + (N \times 2) + 1$$

$$c = \frac{\log 0.05}{\log\left(\frac{N-1}{N}\right)}$$

Figure 112. Size of a Buffer Pool

It can be shown (no doubt also proven) that the value of c is actually very close to $(N \times 3) - 1$, even for small N values. We can therefore simplify everything to a rule:

RULE 1

If the buffer pool size is five times the number of nonleaf pages, you can safely count on only one physical I/O (for the leaf page).

And if you accept our earlier observation that N is 1% (sometimes much less) of the number of leaf pages, you can also put it as:

RULE 2

If the bufferpool size is 5% of the number of leaf pages, you can safely count on only one physical I/O (for the leaf page).

This rule is very conservative. In Chapter 6, "Implementing the Physical Database Design" you can find a slight variation of this rule, together with additional comments.

Appendix E. The PLAN_TABLE

The VIEW definition in Figure 113 has been used to format the contents of the PLAN_TABLE in such a way that it can be easily compared with a QUBE worksheet.

```
create view pl_view
  (
   index, mc, xo, me, at, table, srtn_ujog, srtc_ujog, pf,
   qno, qbno, pno, mseq
  ) as
 select substr(accessname,1,10),
        substr(digits(matchcols),5,1),
        indexonly,
        substr(digits(method),5,1),
        accesstype,substr(tname,1,10),
        sortn_uniq || sortn_join || sortn_orderby || sortn_groupby,
        sortc_uniq || sortc_join || sortc_orderby || sortc_groupby,
        prefetch,
        queryno,qblockno,planno,mixopseq
   from plan_table ;

explain all set queryno = 1000 for
        select * from order order by orddate ;

select * from pl_view
 where qno = 1000
 order by qbno, pno, mseq ;

INDEX      MC  XO  ME  AT  TABLE       SRTN_UJOG  SRTC_UJOG  PF
---------- --  --  --  --  ----------  ---------  ---------  --
           0   N   0   R   ORDER       NNNN       NNNN       S
           0   N   3                   NNNN       NNYN
```

Figure 113. Use of the PLAN_TABLE

MC Matching columns. For any index scan, the number of (leading) index columns that were useful to narrow down the search, using the index structure. A high value is always ″good.″ The value of 0 indicates a nonmatching index scan, that is, reading all leaf pages.

XO Y/N flag for index-only processing

ME Method, applicable to this row of the PLAN_TABLE. Values are:

- **0** No special method
- **1** Nested loop join
- **2** Merge scan join
- **3** Sort operation
- **4** Hybrid join

AT Access type, R(relational) or I(index). A relational scan is a table scan.

SRTN These four flags indicate a sort on the inner table in a join operation. The inner table is the table that is shown in the PLAN_TABLE row with the join method.
The four separate flags indicate:

U Sort for uniqueness
J Sort for (special) join purposes
U Sort for ordering
U Sort for grouping

SRTC These four flags indicate a sort on the outer table in a join operation. The outer table is the table that is shown in the PLAN_TABLE row just before the row with the join method.
These flags are also used for a normal sort operation, usually as the last step of a query.
The meaning of the flags is the same as above.

The layout of the PLAN_TABLE is subject to version and release changes, the details of which are found in *IBM DB2: Administration Guide*.

Process	T	SR	9021-711 (ms)			Your CPU (ms)		
			CPU	LRT	Sort •	CPU	LRT	Sort •
Matching index scan (1 row)	2	2	3.4	45	N/A			N/A
Matching index scan, clustered (100 rows)	200	2	43	104	5.3			
Matching index scan, nonclustered (100 rows)	200	101	43	2084	5.3			
Nonmatching index scan (1 row) •	t + 1	2	1001	3000	N/A			N/A
Nonmatching index scan, clustered (100 rows) •	t + 1	2	1001	3000	5.3			
Nonmatching index scan, nonclustered (100 rows) •	t + 1	101	1001	3520	5.3			
Table scan, all rows	t	1	20000	30020	4500			
Table scan, 100 rows •	t	1	1000	30000	5.3			
″Table scan,″ via clustered index•	t*2	2	2000	3040	N/A•			N/A
″Table scan,″ via nonclustered index•	t*2	p + 1	2000	203020	N/A•			N/A
Table has 100,000 rows (t), 10,000 pages (p). Index has 1,000 leaf pages.		• Additional sort time (LRT) assuming ORDER BY and Sort Assist hardware. • This is a Very Pessimistic case. The *touch cost* has been adjusted. • The purpose of this access is to avoid a sort.						

Index

R

S

T

U

V

ITSO Technical Bulletin Evaluation RED000

International Technical Support Organization
DB2 - Quick Upper Bound Estimate
An Application Design Methodology
August 1995

Publication No. SG24-2549-00

Your feedback is very important to help us maintain the quality of ITSO Bulletins. **Please fill out this questionnaire and return it using one of the following methods:**

- Mail it to the address on the back (postage paid in U.S. only)
- Give it to an IBM marketing representative for mailing
- Fax it to: Your International Access Code + 1 914 432 8246
- Send a note to REDBOOK@VNET.IBM.COM

Please rate on a scale of 1 to 5 the subjects below.
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction	____		
Organization of the book	____	Grammar/punctuation/spelling	____
Accuracy of the information	____	Ease of reading and understanding	____
Relevance of the information	____	Ease of finding information	____
Completeness of the information	____	Level of technical detail	____
Value of illustrations	____	Print quality	____

Please answer the following questions:

a) If you are an employee of IBM or its subsidiaries:

Do you provide billable services for 20% or more of your time? Yes____ No____

Are you in a Services Organization? Yes____ No____

b) Are you working in the USA? Yes____ No____

c) Was the Bulletin published in time for your needs? Yes____ No____

d) Did this Bulletin meet your needs? Yes____ No____

If no, please explain:

_____

_____

What other topics would you like to see in this Bulletin?

_____

_____

What other Technical Bulletins would you like to see published?

_____

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

_____ _____
Name Address

_____ _____
Company or Organization

_____ _____
Phone No.

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

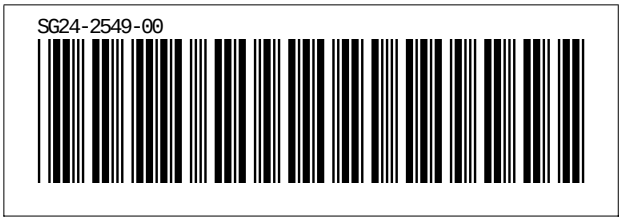
FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM International Technical Support Organization
Department HZ8, Building 657
P.O. BOX 12195
RESEARCH TRIANGLE PARK NC
USA 27709-2195

Fold and Tape **Please do not staple** Fold and Tape

Printed in U.S.A.



Artwork Definitions

id	File	Page	References
ITSLOGO	2549SU		
		i	i

Table Definitions

id	File	Page	References
QT1	2549VARS		
		i	i, i, i, i, 7, 7, 8, 8, 12, 12, 18, 18, 18, 19, 19, 20, 20, 21, 21, 21, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 25, 25, 26, 26, 30, 30, 31, 31, 31, 31, 31, 31, 31, 31, 32, 32, 32, 32, 32, 32, 34, 34, 34, 34, 34, 34, 34, 34, 34, 38, 40, 44, 44, 44, 45, 45, 45, 47, 47, 47, 47, 47, 47, 47, 49, 49, 49, 51, 51, 53, 54, 54, 55, 57, 74, 75, 75, 76, 77, 78, 78, 79, 79, 79, 80, 80, 80, 80, 81, 81, 83, 84, 84, 85, 85, 85, 85, 87, 87, 87, 87, 88, 88, 88, 89, 89, 90, 90, 95, 95, 96, 97, 97, 98, 101
QT2	2549VARS		
		i	i, 12, 12, 12, 12, 12, 18, 18, 18, 19, 19, 20, 20, 21, 22, 22, 23, 23, 23, 23, 23, 25, 25, 25, 25, 26, 26, 26, 30, 31, 31, 32, 32, 34, 34, 34, 44, 44, 45, 45, 45, 45, 47, 47, 53, 54, 80, 83, 88, 90, 98, 101
QTH0	2549VARS		
		i	7, 7, 8, 8, 12, 12, 18, 18, 18, 19, 19, 20, 20, 21, 21, 22, 22, 23, 23, 25, 25, 26, 30, 30, 31, 32, 32, 34, 34, 34, 38, 40, 43, 44, 45, 45, 45, 47, 47, 49, 49, 51, 53, 53, 54, 55, 57, 74, 75, 75, 76, 77, 78, 78, 79, 80, 80, 81, 81, 82, 83, 83, 85, 85, 86, 87, 87, 88, 89, 90, 95, 95, 96, 97, 97, 98, 101
QTH1	2549VARS		
		i	7, 7, 8, 8, 12, 12, 18, 18, 18, 19, 19, 20, 20, 21, 22, 22, 23, 23, 23, 25, 25, 26, 30, 31, 31, 32, 32, 34, 34, 34, 38, 40, 44, 44, 45, 45, 45, 47, 47, 49, 49, 51, 53, 54, 54, 55, 57, 74, 75, 75, 76, 77, 78, 78, 79, 80, 80, 81, 81, 83, 84, 85, 85, 87, 87, 88, 89, 90, 95, 95, 96, 97, 97, 98, 101
QTE1	2549VARS		
		i	i, 74, 75, 75, 76, 77, 78, 78, 79, 79, 79, 79, 80, 80, 80, 80, 80, 80, 81, 81, 82, 82, 83, 83, 84, 84, 85, 85, 85, 85, 86, 86, 87, 87, 87, 87, 87, 89, 89, 90, 90, 90, 95, 95, 96, 97, 97, 98, 98
QTEH1	2549VARS		
		i	74, 75, 75, 76, 77, 78, 78, 79, 80, 80, 81, 81, 82, 83, 84, 85, 85, 86, 87, 87, 89, 90, 95, 95, 96, 97, 97, 98
QTT1	2549VARS		
		i	95, 95, 95, 95, 95, 95, 96, 96, 96, 97, 97, 97, 97, 97, 97, 98, 98, 98, 98
CT1	2549CH1		
		5	5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5
CTH1	2549CH1		
		5	5
IXI1	2549CH2		
		16	16, 16, 16
IXIH1	2549CH2		
		16	16
TBSWS	2549CH4		
		39	39
TECHN0	2549CH6		
		60	60
TECHN1	2549CH6		
		60	60, 60
TECHN2	2549CH6		
		60	60
IXLF0	2549CH6		
		66	66, 66
IXLF1	2549CH6		
		66	66, 66, 66, 66, 66, 66
TA1	2549CH8		
		93	93, 93
TAH1	2549CH8		
		93	93
MACH1	2549AX2		
		108	108
TW1	2549AX6		
		121	121, 121
TWR1	2549AX6		
		121	121, 121, 121, 121, 121, 121, 121, 121, 121, 121, 121
TWR2	2549AX6		
		121	121

Figures

				24
QW2N	2549CH2			
		25	29	
				25
QW2O	2549CH2			
		25	30	
				25
TVARSOL	2549CH2			
		26	31	
				26
TUP1	2549CH3			
		29	32	
				29
QW3A	2549CH3			
		30	33	
				30
QW3B	2549CH3			
		30	34	
				30, 31
QW3C	2549CH3			
		31	35	
				31, 31
QW3D	2549CH3			
		32	36	
				32, 33, 33
QW3E	2549CH3			
		32	37	
				32
QW3F	2549CH3			
		34	38	
				34
QW3G	2549CH3			
		34	39	
				34
QW3H	2549CH3			
		34	40	
				34
FGBATCH	2549CH4			
		38	41	
				37, 54
QW4A	2549CH4			
		38	42	
				38, 39, 40, 43
FGBW	2549CH4			
		38	43	
				38
TRPA	2549CH4			
		39	44	
				39
QW4B	2549CH4			
		40	45	
				40
QW5A	2549CH5			
		43	46	
QW5B	2549CH5			
		44	47	
				44
QW5C	2549CH5			
		45	48	
				45, 45, 46
QW5D	2549CH5			
		45	49	
				45, 45, 46
QW5E	2549CH5			
		45	50	
				45
QW5F	2549CH5			
		47	51	
				47
QW5G	2549CH5			
		47	52	
				47
QW5H	2549CH5			
		49	53	
				49
QW5H2	2549CH5			
		49	54	
				49
FGRI	2549CH5			
		51	55	
				51
QW5I	2549CH5			
		51	56	
				51
QW5J	2549CH5			
		53	57	
				53

QW5K	2549CH5	53	58	53
QW5L	2549CH5	54	59	54
QW5M	2549CH5	55	60	55, 55, 55
QW5N	2549CH5	57	61	57
TECHN	2549CH6	60	62	60, 61
FGTUPD	2549CH6	60	63	60
FGHR	2549CH6	64	64	70
IXSIZE	2549CH6	66	65	66
QW7A	2549CH7	74	66	74
QW7B	2549CH7	75	67	75, 75
QW7C	2549CH7	75	68	
QW7D	2549CH7	76	69	76
QW7E	2549CH7	77	70	77, 77
QW7F	2549CH7	78	71	78, 78
QW7G	2549CH7	78	72	78, 78
QW7H	2549CH7	79	73	79, 79, 80
QW7I	2549CH7	80	74	80
QW7J	2549CH7	80	75	80, 80
QW7K	2549CH7	81	76	80
QW7L	2549CH7	81	77	81
QW7M	2549CH7	82	78	82
QW7N	2549CH7	83	79	83
QW7O	2549CH7	83	80	83
QW7P	2549CH7	85	81	85
QW7Q	2549CH7	85	82	85, 86
QW7R	2549CH7	86	83	86, 86
QW7S	2549CH7	87	84	
QW7T	2549CH7	87	85	87, 88, 89
QW7U	2549CH7	88	86	88
QW7V	2549CH7	89	87	

				89
QW7W	2549CH7			
		90	88	
				90
FGACCT	2549CH8			
		92	89	
				92
TACCTQ	2549CH8			
		93	90	
				93
QW8A	2549CH8			
		95	91	
				95
QW8B	2549CH8			
		95	92	
				95
QW8C	2549CH8			
		96	93	
				96
QW8D	2549CH8			
		97	94	
				97
QW8E	2549CH8			
		97	95	
				97
QW8F	2549CH8			
		98	96	
				98
QW9A	2549CH9			
		101	97	
				101
FGORD	2549AX1			
		105	98	
				105
FGQF	2549AX2			
		107	99	
				107
TMACH1	2549AX2			
		108	100	
				108
FGQUEUE	2549AX2			
		110	101	
				110
TSPEC1	2549AX3			
		113	102	
				113
TSPEC1A	2549AX3			
		114	103	
				114
FGSP1B	2549AX3			
		114	104	
				114
TBCOMP	2549AX3			
		114	105	
				114
TSPEC2	2549AX3			
		115	106	
				115
FGSIMP1	2549AX3			
		115	107	
				115
TBCOMP2	2549AX3			
		115	108	
				115
FGFNEW	2549AX3			
		116	109	
				116
FLRU1	2549AX4			
		117	110	
				117, 117
FLRU2	2549AX4			
		117	111	
				117
FGBPSZ1	2549AX4			
		118	112	
				118
FGPLAN	2549AX5			
		119	113	
				119

Headings

Index Entries

id	File	Page	References
I05	2549CH1	6	(1) SR 6
I02	2549CH1	9	(1) LRT 9, 9
I04	2549CH2	15	(1) RID 15
I01	2549CH2	17	(1) prefetch 17, 17
I03	2549CH4	39	(1) LRU 39
I06	2549CH9	100	(1) DDL 100
I07	2549CH9	100	(1) DCL 100

Footnotes

id	File	Page	References
FN9021	2549CH1	11	7 11
FNCICS	2549CH1	11	8 11

Spots

id	File	Page	References
NOPAGES	2549CH1	4	(no text) 17
SPRID	2549CH2	15	(no text) 29, 46
SPLSTPF	2549CH2	20	(no text) 75
LPARG	2549CH2	21	(no text) 22

Processing Options

Runtime values:

Document fileid	SG242549 SCRIPT
Document type	USERDOC
Document style	IBMXAGD
Profile	EDFPRF30
Service Level	0029
SCRIPT/VS Release	4.0.0
Date	95.08.08
Time	12:28:15
Device	3820A
Number of Passes	4
Index	YES
SYSVAR D	YES
SYSVAR G	INLINE
SYSVAR S	OFF
SYSVAR V	ITSCEVAL

Formatting values used:

Annotation	NO
Cross reference listing	YES
Cross reference head prefix only	NO
Dialog	LABEL
Duplex	YES
DVCF conditions file	(none)
DVCF value 1	(none)
DVCF value 2	(none)
DVCF value 3	(none)
DVCF value 4	(none)
DVCF value 5	(none)
DVCF value 6	(none)
DVCF value 7	(none)
DVCF value 8	(none)
DVCF value 9	(none)
Explode	NO
Figure list on new page	YES
Figure/table number separation	NO
Folio-by-chapter	NO
Head 0 body text	Part
Head 1 body text	Chapter
Head 1 appendix text	Appendix
Hyphenation	NO
Justification	YES
Language	ENGL
Layout	OFF
Leader dots	YES
Master index	(none)
Partial TOC (maximum level)	4
Partial TOC (new page after)	INLINE
Print example id's	NO
Print cross reference page numbers	YES
Process value	(none)
Punctuation move characters	.,
Read cross-reference file	(none)
Running heading/footing rule	NONE
Show index entries	NO
Table of Contents (maximum level)	3
Table list on new page	YES
Title page (draft) alignment	RIGHT
Write cross-reference file	(none)

Imbed Trace

Page 0	2549SU
Page 0	2549VARS
Page 0	2549FM
Page i	2549EDNO
Page ii	2549ABST
Page xiii	2549SPEC
Page xiii	2549TMKS
Page xiv	2549PREF
Page xviii	2549ACKS
Page xviii	2549CH1
Page 7	2549IM01
Page 7	2549IM01
Page 8	2549IM01
Page 8	2549IM01
Page 12	2549IM01
Page 12	2549IM01
Page 14	2549CH2
Page 18	2549IM01
Page 18	2549IM01
Page 18	2549IM01
Page 19	2549IM01
Page 19	2549IM01
Page 20	2549IM01
Page 20	2549IM01
Page 21	2549IM01
Page 21	2549IM01
Page 22	2549IM01
Page 22	2549IM01
Page 23	2549IM01
Page 23	2549IM01
Page 25	2549IM01
Page 25	2549IM01
Page 26	2549IM01
Page 28	2549CH3
Page 30	2549IM01
Page 30	2549IM01
Page 31	2549IM01
Page 32	2549IM01
Page 32	2549IM01
Page 34	2549IM01
Page 34	2549IM01
Page 34	2549IM01
Page 34	2549CH4
Page 38	2549IM01
Page 40	2549IM01
Page 41	2549CH5
Page 43	2549IM01
Page 44	2549IM01
Page 45	2549IM01
Page 45	2549IM01
Page 45	2549IM01
Page 47	2549IM01
Page 47	2549IM01
Page 49	2549IM01
Page 49	2549IM01
Page 51	2549IM01
Page 53	2549IM01
Page 53	2549IM01
Page 54	2549IM01
Page 55	2549IM01
Page 57	2549IM01
Page 57	2549CH6
Page 71	2549CH7
Page 74	2549IM01
Page 74	2549IM02
Page 75	2549IM01
Page 75	2549IM02
Page 75	2549IM01
Page 75	2549IM02
Page 76	2549IM01
Page 76	2549IM02
Page 77	2549IM01
Page 77	2549IM02
Page 78	2549IM01
Page 78	2549IM02
Page 78	2549IM01
Page 78	2549IM02
Page 79	2549IM01
Page 79	2549IM02
Page 80	2549IM01
Page 80	2549IM02
Page 80	2549IM01
Page 80	2549IM02