PRINCIPLES OF DATABASE AND KNOWLEDGE-BASE SYSTEMS

VOLUME II: The New Technologies

Jeffrey D. Ullman
STANFORD UNIVERSITY

PRINCIPLES OF DATABASE AND KNOWLEDGE-BASE SYSTEMS

VOLUME II: The New Technologies

Jeffrey D. Ullman
STANFORD UNIVERSITY

COMPUTER SCIENCE PRESS

Library of Congress Cataloging-in-Publication Data

Ullman, Jeffrey D., 1942–
Principles of database and knowledge-base systems.

(Principles of computer science series, 0888-2096; 14–)
Includes bibliographies and indexes.
1. Data base management. 2. Expert systems (Computer Science) I. Title. II. Series: Principles of computer Science series; 14, etc.
QA76.9.D3U443 1988 005.74 87-38197
ISBN 0-7167-8069-O (v. 1)
ISBN 0-7167-8162-X (v. 2)

Printed in the United States of America

Computer Science Press
1803 Research Boulevard
Rockville, MD 20850

An imprint of W. H. Freeman and Company
41 Madison Avenue, New York, NY 10010
20 Beaumont Street, Oxford OX1 2NQ, England

1 2 3 4 5 6 7 8 9 0 RRD 7 6 5 4 3 2 1 0 8 9

PRINCIPLES OF COMPUTER SCIENCE SERIES

Series Editors

Alfred V. Aho, Bell Telephone Laboratories, Murray Hill, New Jersey
Jeffrey D. Ullman, Stanford University, Stanford, California

Computer Organization
Michael Andrews
Trends in Theoretical Computer Science
Egon Börger, Editor
The Turing Omnibus
A. K. Dewdney
Formal Languages and Automata Theory
Vladimir Drobot
Advanced C: Food for the Educated Palate
Narain Gehani
C: An Advanced Introduction
Narain Gehani
C: An Advanced Introduction, ANSI C Version
Narain Gehani
C for Personal Computers: IBM PC, AT&T PC 6300, and Compatibles
Narain Gehani
An Introduction to the Theory of Computation
Eitan Gurari
Theory of Relational Databases
David Maier
An Introduction to Solid Modeling
Martti Mäntylä
Principles of Computer Design
Leonard R. Marino
UNIX: The Minimal Manual
Jim Moore
A Logical Language for Data and Knowledge Bases
Shamim Naqvi and Shalom Tsur
The Theory of Database Concurrency Control
Christos Papadimitriou
Algorithms for Graphics and Image Processing
Theo Pavlidis
Data Compression: Methods and Theory
James A. Storer
The Elements of Artificial Intelligence
Steven Tanimoto
Computational Aspects of VLSI
Jeffrey D. Ullman
Principles of Database and Knowledge-Base Systems, Volumes I and II
Jeffrey D. Ullman

OTHER BOOKS OF INTEREST

Jewels of Formal Language Theory
Arto Salomaa
Principles of Database Systems, Second Edition
Jeffrey D. Ullman
Fuzzy Sets, Natural Language Computations, and Risk Analysis
Kurt J. Schmucker
LISP: An Interactive Approach
Stuart C. Shapiro

PREFACE

The second volume of *Principles of Database and Knowledge-Base Systems* is, to a large extent, a personal statement about what I think database system theory gives to the development of database and knowledge-base systems. Having covered the basics in the first volume, I offer the reader a vision of the ideas that will make knowledge-base systems feasible. These ideas surfaced beginning in the mid-1980's, and they form a beautiful example of how a body of theoretical notions can be translated into practice very quickly.

The reader of the first volume will note that I do not have a very ambitious definition of "knowledge." Recall that, as defined in Section 1.6 of Volume I, knowledge-base systems

1. Must deal with massive amounts of data,
2. Must do so efficiently, and
3. Must offer the user a declarative language in which to express queries of a general nature.

I have avoided grandiose notions of what knowledge and reasoning are, in favor of something one realistically could expect would have efficient algorithms for execution of queries. Thus, we have used as our declarative language a simple form of logical rules, and we continue to do so in this volume.

Synopsis of the Book

If we are to compile a declarative language, that is, a language in which we state what we want, without giving an algorithm for obtaining it, then optimization must play an important role in the system. Thus, Chapter 11, the first in this volume, covers optimization of queries in relational database systems. It is fair to say that relational query languages are the most commonly used declarative languages, and that they could not be used without well-designed query optimizers. Yet it is common in books on database systems (or in books on compilers for that matter) to ignore or minimize the subject of optimization. I hope that Chapter 11 will convince the reader there is an interesting and important body of knowledge relevant to optimization of relational query languages.

Most of the balance of the book is devoted to the technology of optimization for logic programs. Recall that logic languages can express many queries that relational query languages cannot (see Section 3.7 of Volume I). Thus,

optimization of logic languages is predictably harder than optimization for relational query languages, and systems capable of optimizing logic languages are in an experimental state. Nevertheless, much has been achieved recently, and this book covers the ideas that look most promising. Chapter 12 is preparatory to this study, covering material on logic not found in Volume I, such as top-down query answering (resolution-based schemes that try to satisfy a goal). Also included is some of the specialized notation needed for further study in Chapters 13 through 16.

Chapter 13 introduces the "magic-sets" technique for processing logic. The artificial-intelligence community has long known that there are advantages both to top-down processing and to bottom-up processing (where we deduce all possible facts from the database). Very recently, a new methodology, called "magic-sets," has been developed; it lets us enjoy the advantages of both top-down and bottom-up processing, with the disadvantages of neither. This technique is rapidly finding its way into implementations of knowledge-base systems and needs to be understood by anyone working in the area. It is interesting to conjecture that resolution-based methods have had their day, and that, when fully explored, "magic sets" will prove superior for all applications, from theorem proving to query answering in knowledge-base systems.

In Chapter 14 we cover "conjunctive queries," which have an interesting and useful theory; this body of techniques is used at several points in the following chapter. Chapter 15 covers linear recursion, a restricted but very common and important special case of logic, where algorithms are known that evaluate queries even faster than the magic sets algorithm does. Then, in Chapter 16 we meet several experimental knowledge-base systems and see how the ideas of Chapters 13 through 15 are being put into practice.

The final chapter covers universal-relation systems. The universal-relation idea was developed by a number of people, the author included, in the late 1970's and early 1980's. Its intent is to provide a way of talking about databases without knowing the details of the database scheme. The canonical example is a natural-language interface to a database, where the naive user might be expected to know the attributes, that is, names for the elementary concepts of the database, but could not be expected to know the relations into which they were grouped. With this goal in mind, an extensive body of theory has been developed, concerning what connections among attributes it is reasonable for a system to infer.

At the time of the writing of this book, a few systems based on universal-relation concepts have been built, and the number is growing. In fact, a leading purveyor of relational software now offers an enhancement that allows one to treat several relations with common keys as a single universal relation. However, as we shall see in Chapter 17, this idea merely scratches the surface of what can be achieved.

Unfortunately, universal-relation systems are still quite rare in practice. Moreover, there have been a number of public attacks on the concept (see the bibliographic notes to Chapter 17), often by people who misinterpreted the concept. It takes less time to read Chapter 17 than it does to write a polemic about what is wrong with the idea, and I hope that in the future we shall see commercial systems using the universal-relation concept routinely.

Prerequisites

The reader of this volume is assumed to have familiarity with the concepts found in the first volume, although Chapters 11 and 12 begin with reviews of relevant notation and the most central concepts. Below is a list of the most important prerequisites.

Chapter 11: The relational model (Section 2.3) and relational algebra (Section 2.4); relational query languages from Chapter 4, especially QUEL from Section 4.3 and SQL from Section 4.6; physical storage and index structures, as in Sections 6.1 through 6.8.

Chapter 12: Chapter 3 (logic), especially Sections 3.1 through 3.6.

Chapter 13: Most of Chapter 12; Chapter 6, including the material on index structures and on partial-match structures (Sections 6.12 through 6.14).

Chapter 14: Relational database theory, especially Sections 7.4 and 7.11.

Chapter 15: Chapters 13 and 14; Section 2.7 on the object model; relational query languages as in Chapter 4; index and partial-match structures from Chapter 6.

Chapter 16: Chapters 11, 13, and 15; the nature of a knowledge-base system (Chapter 1); hierarchical and object-oriented models (Sections 2.6, 2.7, and 5.6); relational languages (Chapter 4, especially Section 4.3 on QUEL); Section 9.5 on lock modes.

Chapter 17: Chapter 11; QUEL from Section 4.3; most of Chapter 7 on dependency theory and normalization.

Exercises

As in the first volume, singly starred exercises require some significant thought, and the hardest exercises are doubly starred.

Bibliography and Index

The bibliography includes all works cited in this volume and/or Volume I. Likewise, the index contains entries for both volumes. Pages 1 through 587 refer to Volume I, and pages 633 through 1069 refer to Volume II.

Acknowledgements

The following people are thanked for comments that improved this manuscript: Surajit Chaudhuri, Kwong Choy, Isabel Cruz, William Harvey, Chen-Lieh Huang, Bill Lipa, Alberto Mendelzon, Inderpal Mumick, Jeff Naughton, Geoff Phipps, Raghu Ramakrishnan, Ken Ross, Shuky Sagiv, and Yumi Tsugi.

The writing of this volume was partially supported by a John Simon Guggenheim fellowship. It was facilitated by computing equipment donated to Stanford by IBM Corp. and AT&T Foundation.

J. D. U.
Stanford CA

TABLE OF CONTENTS

CHAPTER 11

Query Optimization for Database Systems

In this chapter, we shall learn the basic techniques for query optimization. When queries are expressed in a declarative language, such as a relational query language, it is not easy for the database system to execute queries quickly. Rather, an extensive optimization phase must select, from among the many possible ways to implement a given query, one of the few efficient implementations; ideally, the best implementation must be selected.

Section 11.1 introduces the model that we use for describing and evaluating query-optimization algorithms. Section 11.2 studies an important example of an optimization algorithm that considers an exponential number of possibilities before choosing the most efficient evaluator—the System R algorithm for selection queries. The following three sections consider different ways of computing products and joins and evaluates the cost of each.

Then in Sections 11.6 and 11.7, we consider techniques for manipulating algebraic expressions to produce expressions that are more efficiently evaluated. Sections 11.8 to 11.10 are devoted to a description of the Wong-Youssefi algorithm used to optimize queries in QUEL, the language of the INGRES relational DBMS. Section 11.8 discusses an algorithm used to take the join of more than two relations, and Section 11.9 introduces a concept that is important for several different query-optimization algorithms—the representation of select-project-join queries by hypergraphs. Finally, in Section 11.10, the QUEL query optimization algorithm itself is presented.

Section 11.11 introduces the modifications to our basic model that must be made to handle distributed queries. In Section 11.12, we introduce "acyclic hypergraphs," a very interesting and important concept. In the context of distributed query optimization, the property of queries involving joins that is called "acyclicity," allows much more efficient query evaluation than is possible in the general case. Section 11.13 applies the acyclicity concept to computation

of distributed joins, and the next section presents Yannakakis' algorithm, a technique for computing projections of acyclic joins that is efficient both in the distributed environment and in the uniprocessor environment. Finally, Section 11.15 discusses an algorithm used in the distributed DBMS System R* for taking distributed joins. This algorithm, like the System R algorithm discussed in Section 11.2, is one that searches an exponential number of possibilities to choose the method it prefers.

Notation for Relations

We assume the reader is familiar with the relational data model, as described in Section 2.3 of Volume I, and with relational algebra from Section 2.4. We shall mention briefly some of the notation and conventions that we use in this book to describe relations, relation schemes, and tuples.

1. Names of relations are generally capital letters, for example, R. The current value for relation R is typically denoted by the corresponding lowercase letter, for example, r. Frequently, we use the relation scheme (set of attributes) as the name of the relation; for example, ABC could be the name of a relation with attributes A, B, and C. We also treat relation names as if they were their schemes; for example, $R \cap S$ stands for the set of attributes that are common to R and S, while $r \cap s$ is the set of tuples common to the current relations for R and S. However, the context will always differentiate these two uses of intersection, so we might use $R \cap S$ to refer to the intersection of the tuples if the meaning is unambiguous.
2. Sets of attributes are often denoted by concatenation, rather than by the usual set notation. For example, ABC stands for the set of attributes $\{A, B, C\}$, and XY stands for $X \cup Y$, if X and Y are sets of attributes.
3. Tuples are named by Greek letters, for example, μ. When we want to specify the components of tuples, we write them $a_1 \cdots a_n$ or $(a_1, \ldots, a_n)$, whichever looks better in context. Either notation stands for the tuple with n components, the ith of which is a_i, for $i = 1, 2, \ldots, n$. The components of tuple μ for set of attributes X is denoted $\mu[X]$.
4. The relational algebra operators are denoted $\cup$ (union), $-$ (difference), σ (selection), π (projection), and $\times$ (Cartesian product). We also use $\bowtie$ for the natural join, $\overset{F}{\bowtie}$ for the join with condition F on the attributes (for example, $R \underset{\$1=\$3}{\overset{F}{\bowtie}} S$ for the equijoin that equates the first column of R to the third column of S), and $\ltimes$ for the semijoin, defined by:[1]

$$R \ltimes S = \pi_R(R \bowtie S) = R \bowtie \pi_{R \cap S}(S)$$

[1] Note that that R and $R \cap S$ refer to sets of attributes in the projections below.

5. Columns of relations can be represented by attribute names if we know them, or by number. We generally use $\$i$ to denote column i, although in projections, where there can be no confusion, we use i by itself. For example, $\pi_1(\sigma_{\$2=3}(R))$ asks for the first component of all those tuples of R whose second component has value 3. If several relations have the same attribute name, we can use $R.A$ to refer to attribute A belonging to relation R.

11.1 BASIC ASSUMPTIONS FOR QUERY OPTIMIZATION

Let us recall the model of computation costs we introduced in Chapter 6 (Volume I). A relation is represented in memory by a collection of records; each record corresponds to one tuple of the relation. Tuples are divided among blocks of secondary storage. Depending on the physical storage organization, the tuples of one relation may be packed into blocks used for that relation alone, or the tuples may appear in blocks that are used for tuples of several relations.

For a relation R, let T_R be the number of tuples in R, and let B_R be the number of blocks in which all the tuples of R could fit, if they were packed tightly. Note that, in general, B_R will be significantly smaller than T_R. Also, T_R/B_R is the number of tuples of R that will fit on one block.

Example 11.1: A typical storage system uses blocks of 1024 bytes. Suppose we have a relation R with 1,000,000 tuples, each 100 bytes long. Then T_R = 1,000,000. As ten tuples can fit on one block, R can fit in 100,000 blocks; that is, B_R = 100,000. □

The cost of reading or writing a relation is the number of blocks that must be read from secondary storage into main memory or written from main memory into secondary storage. We call each such operation a *block access*. The physical organization of a relation can affect greatly the cost of operations on R. For instance, if the relation R of Example 11.1 is *packed*, that is, stored in roughly as few blocks as can hold all its tuples, then we can scan all the tuples of R in B_R block accesses. On the other hand, if the tuples of R are stored on blocks, most of which contain only one tuple of R (along with tuples of other relations), then about T_R block accesses are needed to scan R, about ten times the cost of reading a packed relation.

We should understand that relations can be packed, and yet appear with a few tuples of other relations. A typical example is that discussed in Section 6.9, where we saw how, in the DBTG proposal, one relation R might be stored "via set" with another relation S. Then, each tuple of S is followed by all the tuples of R that it "owns," leading to a pattern like that of Figure 11.1, on the assumption that each tuple of S owns several tuples of R.

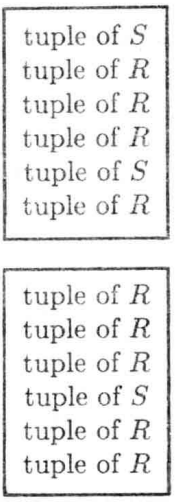

Figure 11.1 Packed relation R sharing blocks with another relation.

The Use of Indices

There are many structures, such as hash tables or B-trees, that are suitable as indices. Such structures let us perform selections on a relation quickly, that is, with a number of block accesses not too much greater than the number of blocks on which the desired tuples are stored. Index structures were covered in Chapter 6, so here we shall not specify any particular structure for indices. We shall only assume that whenever a relation R has an index on attribute A, then the number of block accesses to perform a selection, $\sigma_{A=c}(R)$, is roughly the number of blocks on which we find tuples of R whose value for attribute A is the constant c. In particular, we shall neglect the number of blocks we must access to examine the index itself; that number is usually comparable to, or smaller than, the number of R's blocks we must access.

Clustering and Nonclustering Indices

We can divide indices into *clustering* and *nonclustering* types, and these two types offer significantly different performance. If we have a clustering index on A, then the number of blocks holding tuples with A-value c is approximately the number of blocks on which those tuples can be packed. For example, an ISAM index used as a sparse, primary index is normally a clustering index. The tuples with $A = c$ will appear consecutively, spread over at most one more block of the main file than will hold them.[2] As long as we can expect the number of

[2] We shall ignore the possibility that blocks are purposely given a fraction of available space for future expansion. If such a strategy is followed, it is as if all blocks had a smaller capacity to hold tuples than we would expect by counting bytes.

tuples with $A = c$ to fill at least one block, then the number of main-file block accesses could not be more than double the theoretical minimum, and we can regard the index as "clustering."

With a nonclustering index, we must assume that the tuples of R having $A = c$ each appear on a separate block. Thus, the number of block accesses needed to find all of these tuples is about equal to the number of tuples retrieved. That is, the cost to retrieve the tuples with $A = c$ is about T_R/B_R times as great if the index is nonclustering as if it is clustering. For the relation of Example 11.1, this ratio is 10.

An example of a nonclustering index is a B-tree used as a dense index, pointing to the tuples, which are packed into the blocks of the main file. Presumably, the tuples are organized in the main file according to some primary index on an attribute other than A. For example, these tuples might be hashed into buckets according to their values in attribute B. Then we would not expect two tuples with the same A-value to appear on the same block, except by coincidence.

Image Size

There is another parameter of data that is often useful when estimating the cost of a particular operation. The *image size* of an index on attribute A of relation R, denoted $I_{R.A}$, or just I_A when R is understood, is the expected number of different A-values found in R. On the assumption that values are equally likely to occur, we can estimate the number of blocks retrieved in response to the selection $\sigma_{A=c}(R)$ by T_R/I_A if there is a nonclustering index on A, because that is the expected number of tuples retrieved. If there is a clustering index on A, we retrieve about B_R/I_A blocks, because the selected tuples are packed onto about that number of blocks.

Sources of Optimization

There are two basic kinds of optimizations found in query processors, algebraic manipulation and cost-estimation strategies. Algebraic simplification of queries is intended to improve the cost of answering the query independent of the actual data or the physical structure of the data. We shall study algebraic optimizations in the abstract in Sections 11.6 and 11.7, and in Sections 11.8 to 11.10 we consider the optimization algorithm used in the original implementation of QUEL, which is primarily algebraic. The following is an example of an important algebraic optimization: "do selections as early as possible."

Example 11.2: Suppose we have relations AB, with attributes A and B, and BC, with attributes B and C, and we wish to compute

$$\sigma_{A=d}(AB \bowtie BC) \tag{11.1}$$

Such a query will almost always be answered faster, independent of the values of relations AB and BC, and independent of the presence or absence of indices on their attributes, if we perform the selection before the join. That is, we should compute (11.1) as if it were

$$(\sigma_{A=d}(AB)) \bowtie BC \tag{11.2}$$

The reason why (11.2) is usually faster is that joins are generally much more expensive than selections. We shall, in Section 11.5, estimate the cost of particular joins performed in particular ways, but for the purposes of this example, it suffices to note that computing the join $AB \bowtie BC$ requires accessing every block on which tuples of these two relations reside at least once, perhaps more. We also must create the result of the join and store all of the blocks on which it resides. The result can be much bigger than either AB or BC, if the typical tuple of AB has a B-value that is shared by several tuples of BC.

On the average, taking the selection first will cut down the number of tuples in the first argument of the join by a factor I_A. If we pack the tuples in $\sigma_{A=d}(AB)$ tightly into blocks, then we cut down the number of blocks we must access to get the first argument of the join by at least factor I_A. We also can expect to cut down the size of the result of the join by factor I_A, which is often a more significant saving. The cost of the selection in (11.1) might be less than that in (11.2), if the result of the join is much smaller than the relation AB, that is, if the typical tuple of AB has a low probability of joining with even one tuple of BC. However, ordinarily, the cost of the selection in (11.2) will be no greater than in (11.1), and it may be much less, if the join result is larger than AB and/or if there was an index on A in AB that can be used in (11.2) but not in (11.1).

We should understand, however, that the rule "do selections as soon as possible" is no more than a sensible heuristic; there are exceptions. For example, suppose BC is empty. If the method we use for the join checks this condition immediately, then following (11.1) we immediately discover that $AB \bowtie BC$ is empty, and therefore, the value of (11.1) is the empty relation. However, following (11.2), we would perform the selection, taking some nonnegligible amount of time, and only when we proceeded to perform the join, discover that there are no tuples in the answer. □

The second class of optimization strategies is those that consider issues, such as the existence of indices, to select from among alternatives the strategy that is best for the data and structure at hand. The next section covers the way System R does optimization, focusing on selections as a simple example. The System R approach is oriented toward the estimation of different strategies on the current database. Then, in Sections 11.3 to 11.5, we discuss methods for computing products and joins, and we show how to estimate the costs of the different techniques.