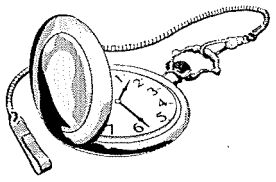
MASTER EN SCIENCES INFORMATIQUES

Database performance tuning or the quest for indexes

Ferber, Guy

Award date:
1996

Awarding institution:
Universite de Namur



Database Performance Tuning.

or

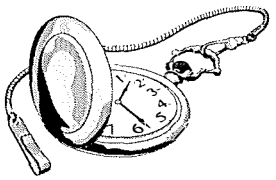
The Quest for Indexes.

Written by: Guy FERBER
Directed by: Jean-Luc HAINAUT
Year: 1996

IMITATION

A DARK unfathomed tide
Of interminable pride -
A mystery, and a dream,
Should my early life seem;
I say that dream was fraught
With a wild and waking thought
Of beings that have been,
Which my spirit hath not seen,
Had I let them pass me by,
With a dreaming eye!
Let none of earth inherit
That vision of my spirit;
Those thoughts I would control,
As a spell upon my soul:
For that bright hope at last
And the light time have past,
And my worldly rest hath gone
With a sigh as it passed on:
I care not though it perish
With a thought I then did cherish.

Edgar Allan Poe



Acknowledgment

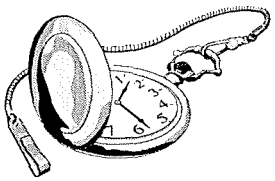
At this stage, I would like to express my gratitude to all the people who were contributing actively or passively to this master thesis.

I'm thinking in particular of Mr. Jean-Luc Hainaut and his team, who have been directing and supporting me during the whole process. I also appreciated the warm welcome and assistance of Mr. Jean-Marc Zeippens, director of the training, and his colleagues at OBLOG Software in Lisbon.

Thank you to my fiancé and Mr. Romain Nilles, who have been helping me in editing and correcting of this piece of work.

Thank you to Jean-Noel Mathon, who has been a great partner.

Thank you also to my parents, who have been giving me the opportunity of studying at the 'Facultés Universitaires Notre-Dame de la Paix'.

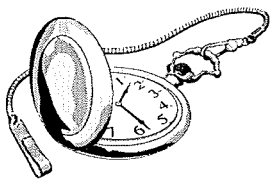


Abstract

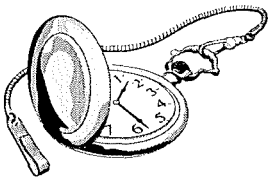
This document is not meant to reinvent the wheel, it mostly is a compilation of database tuning concepts, howevere with a personal touch. It is based upon considerations made by various writers, such as [Hainaut 1986], [O'Neill 1994], [Date 1990], [Finkelstein 1988] and [Elmasri 1994]. It gives an overview of possible parameters aimed to physically tune a database. Moreover, it concerns with the problem of index selection.

Chapter 1, introduces the process of physical database tuning within the process of data modeling. It reveals the pitfalls of selecting the appropriated indexes.
Chapter 2, describes the data operations and deals with execution methods for queries and joining tables. It abstracts the data access operations into a small set of easy to understand and to analyze query types.
Chapter 3, deals with physical data allocation parameters and access structures. It lists a set of parameters that might be helpful during database allocation. It describes and evaluates various data access structures, such as B-Trees, Clusters and Hash indexes.
Chapter 4, tries to consolidate chapter 2 and chapter 3 into a small set of I/O cost relations. It lists a set of relation that might be used to determine rapidly I/O costs for a given query type and access structure.
Chapter 5, is based upon a study made by [Finkelstein 1988] to implement a physical design tool for relational database. It describes a methodology for physical database tuning and lists some tuning guidelines and heuristics used to reduce query execution time.
Chapter 6, gives a practical overview off various considerations that might arise during the quest for the optimal index solution. However, we will not pretend the case study to be exhaustive, as we start with a limited set of data entities, requirements and queries.

This document is aimed to introduce, help and guide the database designer in its first attempts of database tuning.



Contents



Chapter 1. Introduction

The following section introduce concepts and basic considerations about physical data modeling, also known as physical database design. The physical data design is part of a more global design process, the database design. Anticipating an optimal database design at the end of data modeling, the physical database design cannot be considered as a design process on its own. It is constantly interacting with other processes, getting input and giving feedback. Hence, it is important to keep in mind that physical *data modeling is not a process on its own*. Nevertheless, throughout this document we focus our efforts essentially on the physical data modeling process.

1.1. Database Design Process

First, we introduce, with reference to [Elmasri 1994], the physical data modeling and its place within the database design process. As mentioned before the physical design is part of a more global design process, called the database design.

For small databases that interact with few users and little data, database design is not always a complicated topic. However, when medium-size or large databases are designed or redesigned for large organizations and information systems, database design becomes quite complex. This is because the system must satisfy business objectives, dynamic and often complex by nature. Careful design and testing phases are imperative to ensure that all these requirements are satisfactorily met. Medium and large databases are usually used by about 25 to hundreds of users, managing millions of information entities. They also involve hundreds of queries and application programs. Such databases are used in government, industry, banks and large commercial organizations. Service industries such as banking, insurance, travel, hotel and communication companies are totally reliant on successful around-the-clock operation of their databases.

Throughout this document we will try to point-out some rules, guidelines and heuristics that might be helpful, during the search for an optimal data modeling.

We can state the problem of database design as follows:

Design of a logical and physical structures of one or more databases to accommodate the information needs, of the users, in an organization for a defined set of applications [Elmasri 1994]**.**

The *objectives* of database design are multiple. Satisfy the information requirements of specified users and applications. Provide a natural and easy-to-understand *structuring* of the information. Support *processing requirements* and performance objectives such as response time, processing time, and storage utilization. In real world conditions, these goals are hard to measure and accomplish. That is why we will list some helpful guidelines.

The problem of optimal design is worsened by the informal and poorly defined requirements.

The general database design process can be identified throughout six leading phases:

1. Requirements collection and analysis,
2. Conceptual database design,
3. Choice of a Database Management System,
4. Logical database design,
5. Physical database design,
6. Database system implementation.

The design process consists of **two parallel activities**, as illustrated in Figure 1.1.. This first, involves the design of data and structures of the databases; the second is related to the design of database processing and software applications. These activities are closely related. For example, we can identify data entities that have to be stored in database throughout the analysis of database applications. The same way, physical database design, which allows us to choose data storage structures and data access paths, depends highly on applications that use or access the data. On the other hand, the design of database applications is specified by referring to the database schema, which are defined in the first activity. Clearly these two activities strongly influence one each other.

The six phases mentioned above do not have to be processed in sequence. In many cases you may have to modify the design from an earlier phase during a later phase. These feedback loops among phases, and also within phases, are common during database design. Figure 1.1 does not show feedback loops, to avoid complicating the diagram. Phase 1 is concerned with collecting information about the intended use of the data, whereas phase 6 is dedicated to implement the database in a given environment.

- ***Conceptual Database Design*** *(phase 2).* During this phase, we formalize user requirements into a set of *Local Conceptual Schemas.* Each dedicated to a subsystem of the organization. At this stage we consolidate them into a *Conceptual Schema*, which is independent of any specific database management system independent (DBMS). Using a high-level data models, such as Entity-Relational (ER) models [Bodart 1989]. In addition, we identify all possible and known database applications and/or transactions that will use the data, using a formal language, above the specification of any particular DBMS.
- ***Logical Database Design*** *(phase 4).* Throughout this phase we convert the conceptual schema, into an *efficient logical data model,* corresponding to the DBMS chosen throughout phase 3. This phase can take place right after we choose the data model, rather than waiting for the choice of a specific DBMS. For example, we can start the phase after we decide to use a relational DBMS but have not yet decided on a particular one. On market we can find various data models. Based upon hierarchical, relational (DB2, ORACLE, INGRES, SYBASE, etc.), or object oriented (O2, VERSANT) technologies. All these models have specific design characteristics that will drive our logical data model [Hainaut 1986].
- ***Physical Database Design*** *(phase 5)*. During this phase we change the logical schema into computer code documents, namely the DMS-DDL global schemes and the Host-Language code fragments. The first document defines data structures managed by the DBMS, expressed in its Data Description Language (DDL), while the second document implements, most often procedurally, the management of structures, like integrity constraints, that have not been or could not be translated into DDL. This unfortunate splitting is due to weakness in expressing the contents of DBMS compliant logical schema and therefore of the conceptual schema. This phase also includes the design of storage specifications for physical items, such as memory allocation, record placement, and access paths. The design phase ends up with an efficient physical schema in terms of response time, space usage and processing time.

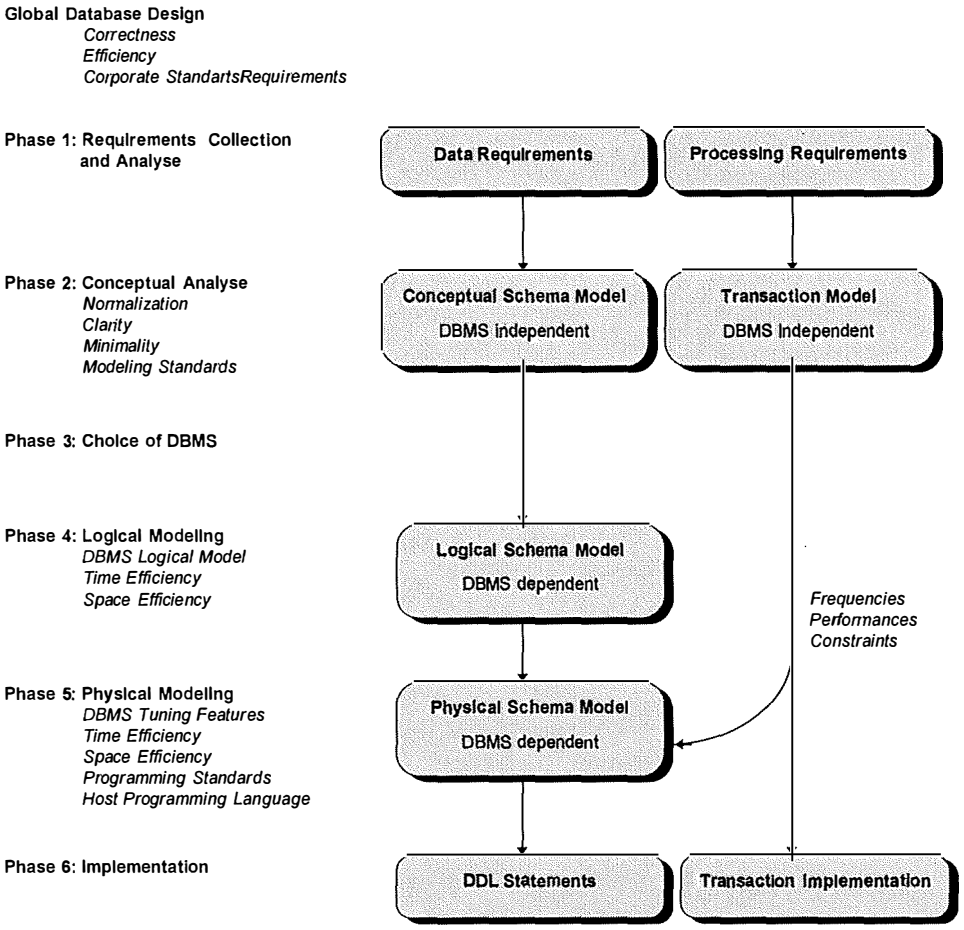


figure 1.1.: Database Design Phases[1]

In the following subsections we discuss briefly each of the six phases of the database design process. We will take a deeper look at physical data modeling and its problems in section 1.2..

1.1.1. Requirements Collection and Analysis

Before we can start modeling an efficient database, we must know the expectations of the users and the intended uses of the data in as much details as possible. We call this process: *Requirements Collection and Analysis*. To specify the requirements, we must first identify all the parts that interact with the information system. This means identifying new and existing users as well as their applications. The requirements of these users and applications are then collected and analyzed [Hainaut 1986]. During requirements analysis, the user requirements are documented in objective hierarchies and events, operations, data, and constraints glossaries.

Typically, the following activities are part of this phase:

[1] [Elmasri 1994]

- *Identify the major application areas and user groups* that will use the data. Key individuals within each group are chosen as the main participants in the subsequent steps of requirements collection and specification.
- *Inspect existing documentation* (policy manuals, forms, reports, and organization charts) to determine their influences on requirements collection and specification process.
- *Study the current environment and intended use of information.* This involves pointing out all *transactions types* and their *frequencies*, as well as the *flow of information* within the system. The input and output data for transactions are documented at this stage.
- *Written responses to a set of questions are collected from the potential database users*. These questions involve the users priorities and the importance they place on various applications and queries. Key individuals may be interviewed for estimating the worth of information and setup transaction priorities.

The requirements collection constitutes a summary, for each table and for each access module, of all used operations [Hainaut 1986]. For each operation the table contains:

- the ***number of activations per time unit*** (day, hour, etc...), noted as Nact/d
- the ***average number of records qualified*** for one activation, noted as Ns
- the ***number of records treated per time unit***, noted as NR/d. It is derived from the following formula: (Nact/d) * (Ns/a) = Ns/d.

All this summarized information is globalised for the whole application (or set of applications), after what it can be quantified into a synthesis according to the rules described in [Hainaut 1986]. The synthesis contains for each table the number of accesses, updates, deletes and updates.

1.1.2. Conceptual Database Design

The second phase of database design process involves two parallel activities. The first activity, *conceptual schema design*, examines the data requirements resulting from phase 1 and produces a conceptual database schema. The second activity, *transaction design*, examines the database applications analyzed in phase 1 and produces high-level specifications for these transactions [Hainaut 1986].

The *conceptual schema design* results in a DBMS independent high level data model which cannot be used directly to implement the database. The importance of such a schema should not be underestimated, for the following reasons:

- The goal of conceptual schema design is a complete and correct understanding of the database structure, meaning (semantics), interrelationships, and constraints. This is best archived without relying on specific DBMS. Each DBMS typically has its own particularities that should not be allowed to interfere with the conceptual design.
- The conceptual schema is invaluable as a stable description of the database contents.

- A good understanding of the conceptual schema is crucial for database users and application designers. Use of high level data models, which are more expressive and general than a given DBMS data model, is important and helpful.
- The graphical description of the conceptual schema serves as an excellent vehicle of communication among database users, designers, and analysts [Bodart 1989].

In this design phase it is important to use a high-level data model (e.g. Entity-Relationship model [Bodart 1989] for example) which respects the following characteristics:

- ***Expressiveness***. The data model should be expressive enough to distinguish different types of data, relationships, and constraints.
- ***Simplicity***. The model should be simple enough for non-specialist users to understand and use its concepts.
- ***Minimality***. The model should have a small number of basic concepts that are distinct and non-overlapping in meaning.
- ***Diagrammatic representation***. The model should have a graphical notation that is easy to understand.
- ***Formality***. A conceptual schema expressed in the data model must represent a formal and exact specification of the data. Hence, the model concepts must be defined accurately and unambiguously.

The purpose of the *transaction design* is to design the characteristics of known database transactions in a DBMS independent way. When a database system is designed, the designers are aware of many known applications and/or transactions that will run on the future database. An important part of database design is to specify the functionality of these transactions as soon as possible in the design process. This ensures that the database schema will include all the facts needed by these transactions. Further, *knowing the relative importance of various transactions and the expected rates of activation play a crucial part in physical data modeling (phase 5)*. As usual, only some of the transactions are known at design time, after the database system is implemented, new transactions are continuously identified and added. However, the most important transactions are often known in advance and should be specified at early stages [Hainaut 1986].

One common technique for specifying transaction at a conceptual level is to identify their *input/output* and *functional* behavior [Hainaut 1986]. By specifying the input data, output data, and internal functional flow of control, designers can specify a transaction in a conceptual and system-independent way. Transactions usually can be grouped into three categories: ***retrieval***, ***update*** and ***mixed*** transactions.

- *Retrieval transactions* are commonly used to retrieve data for display on screen or for production reports.
- *Update transactions* are used to enter data or modify existing data in the database.
- *Mixed transactions* are used for more complex applications that do some retrieval and some update.

Both conceptual design activities should go in parallel, using feedback loops for refinement, until a stable design of schema and transactions is reached.

1.1.3. Choice of a Database Management System

The choice of a DBMS is governed by a various of factors. Some factors are technical, others are economical, and still others are concerned with the organizations policy. The technical factors are concerned with the suitability of DBMS for the task at hand. Issues to consider here are the type of the DBMS (relational, network, hierarchical, object-oriented, etc...), the storage structures and access paths that are supported by the DBMS, the user and programmer interfaces available, the types of high-level query languages, and so on. The reader can find an overview of the technical factors relevant to these data models in [Elmasri 1994]. Let us take a look at the economical and organizational factors which lead the DBMS choice.

The following cost may be considered during DBMS acceptance:

- ***Software acquisition cost.*** This is the 'up-front' cost of buying a software, including language options, different interfaces such as forms and screens, recovery and backup options, special access methods, and documentation.
- ***Maintenance cost.*** This is the recurring cost of receiving standard maintenance service from the vendor and for keeping the DBMS version up to date.
- ***Hardware purchase cost.*** New hardware may be needed, such as additional memory, terminals, disk units, even up to a new environment.
- ***Database creation and conversion cost.*** This is the cost of either creating the database system from scratch or converting an existing system to the new DBMS software. In the latter case it is customary to operate the existing system in parallel with the new system until all new applications are fully implemented and tested. This cost is hard to project and often underestimated.
- ***Personal cost.*** Acquisition of DBMS software for the first time by an organization is often accompanied by reorganization of data-processing. New positions of the database administrator (DBA) and staff are created in most companies that adopt DBMSs.
- ***Training cost.*** Because DBMSs are often complex systems, employees have to be trained to use, deal and program with the DBMS.

The benefits of acquiring a DBMS are not so easy to measure and quantify. A DBMS has several intangible advantages over traditional file systems, such as ease of use, wider availability of data, and faster access to information. More tangible benefits include reduced application development cost, reduced redundancy of data, and better control and security. Based on a cost/benefit analysis, an organization has to decide when to switch over to a DBMS. This move is generally driven by the following factors:

- ***Data complexity.*** As data relationships grow and become more complex, the need for a DBMS is felt more strongly.
- ***Sharing among applications.*** The greater the sharing among applications, the more the redundancy among files is present. The more it becomes complex to keep integrity and coherency among data. Hence, the greater the need for Database Management System.

- ***Dynamically evolving or growing data***. If data changes constantly, it is easier to cope with these changes using a DBMS, because we reduce redundancy and all problems that go in hand with redundant data, like coherence and integrity.
- ***Frequency of ad hoc requests for data***. File systems are not at all suitable for ad hoc data retrieval.
- ***Data volume and need for control***. The sheer volume of data and the need to control goes for DBMS systems.

Finally, several economical and organizational factors also affect the choice of one Database Management System over another:

- ***Structure of the data***. If the data to be stored follows a hierarchical structure, a hierarchical based technology is likely to be suitable. For data with many inter-relationships, a network or relational system may be more appropriate. For complex data structures or data types, like Binary Large Objects (BLOBs) or multi-media objects, an object-oriented system may be suitable.
- ***Familiarity of the staff with the system***. If programming staff within the organization is familiar with a particular DBMS, it may be of benefit to reduce training cost and learning time.
- ***Availability of vendor services***. The existence of near at hand vendor service facilities is desirable to assist in solving any problems with the system. Moving from a non-DBMS to a DBMS driven environment is generally a major undertaking and requires much vendor assistance at the start.

In some cases it may not be appropriate to use a DBMS; instead, it may be preferable to develop in-house software for applications. This may be the case if applications are very well defined and are all known in advance. In such a case, an in-house custom-designed system may be appropriate to implement the known application in the most efficient way. In most cases, however, new applications that were not foreseen at design time come up after system implementation. This is precisely why DBMSs have become very popular: they facilitate the incorporation of new applications without major changes to the existing system.

1.1.4. Logical Data Modeling

The next phase of database design is to create a logical schema in the data model of a selected DBMS.

During the logical modeling phase, the user's data and constraints requirements are represented as a logical data model. In most cases, a straightforward, well-documented, and normalized entity-relationship model is enough to represent the users requirements. In some situations, however, we must use extensions to the basic entity-relationship model, specifically where complex structures and inter-relationships must be modeled.

Data and operations requirements are also converted into entity-life histories. An entity-life history is a logical, execution independent model that represents the interactions between entities and operations. Entity life histories are very useful for ensuring that sufficient attention has been paid to the life cycle of each entity, including data archiving and stripping (removing unnecessary data from the database).

Finally, we map the conceptual schema into a logical one, which includes the following properties. It is *correct*, *optimal* and *independent* of any existing system. Correctness is achieved by mapping all semantics (including integrity constraints) present in the conceptual schema into the logical one, no semantic is added nor retrieved. In search of the logical data access optimum the schema should only hold the data accesses which are mandatory for correct and optimal execution of the required transactions. These data accesses have to be mapped to a data structure which permits optimal data access. For more detail on logical database optimization the reader should consult [Hainaut 1986] [Mathon 1994]. Throughout following lines we only give an overview of a four-level design process.

- ***Schema simplification***. This process transforms the conceptual schema into a simpler, better suited schema for optimization reasoning. For instance, N-ary relation types are transformed into binary ones, multivalued attributes are reduced to single-valued ones, IS-A links are transformed into one-to-one relation types.
- ***DBMS independent optimization***. This process uses transformations through which the schema can be first optimized according to general rules that can apply independent of the chosen DBMS. More generally, the schema can be restructured according to design requirements concerning access time, distribution, data volume, availability, etc.... Schema transformation such as vertical and horizontal splitting or merging, denormalisation or structural redundancy are commonly used to satisfy these requirements.
- ***DBMS translation***. Transforms the schema into structures in accordance with the target DBMS data model [Hainaut 1986]. For instance, for relational DB (or standard files) Many-to-Many relation types are transformed into tables (record types) while Many-to-One relation types are transformed into foreign key (reference fields).
- ***DBMS dependent optimization***. This process performs further *optimiz*ation transformations according to the specific rules of a particular DBMS.

For this pre-physical database management phase, most CASE tools, today, provide extensive entity-relationship modeling capabilities, some with dictionaries to document the models completely and consistently, with cross-references between the various objects and diagrams. Only a few of the more advanced CASE tools provide for specialization and generalization hierarchies. Some of the more advanced CASE tools provide proper entity-life history models and these models are cross-checked to the entity-relationship diagrams. We must be aware, that the more the CASE tools support such features the more the frontier between conceptual design and logical modeling becomes hazy, and the more de design process becomes a logical-conceptual modeling process.

1.1.5. Physical Data Modeling

The logical design results in a schema of all mandatory accesses, as well as their static and dynamic quantification. All this will serve as input to the physical database design which ends up in a physical schema, labeled as *correct*, *optimal* and *executable on a real world system*. Correctness implies that the DB structure and/or the files structure expresses the semantic and the access mechanisms of the mandatory access schema. The fact that the schema is to be optimal does not mean that the data structures and the access paths have to end up in an optimum, but that the schema should achieve an *overall good performance* for all users and applications.

The physical design process can be branched into three points. The first branch is dedicated to the production of the executable schema, the second one is aimed at generating the user views, while the third is concerned with physical database tuning.

- ***DBMS-DDL and Host coding***. Translates the DBMS compliant specifications into the DBMS's Data Description Language (DDL). The rejected specifications are translated into languages, systems and procedures that are out of DBMS control (host language, user interface-manager and human procedures are some examples).
- ***User Views***. This branch determines a subset (views) of schemes that concerns each application and end-user category. The views are translated into executable code, according to the DBMS and programming standards and habits. The code may be divided into two sections: the first one is made of DDL text which translates some of the view structures, while the second one expresses, in host-language code, structures excluded from the DDL texts.
- ***Database Physical Tuning***. This branch defines the storage and/or access structures and parameter settings in order to *optimiz*e the database with respect to user requirements. These choices define the optimal physical schema, with respect to the DBMS. In extention of the DDL schema, for instance, the physical schema will include the specification of indexes, physical file assignment, disk contention, device assignment, record type space mapping, page size, free space definition, clusters, storage nodes, access modes, buffer size and management, etc....

The data structure expressed and the access in the physical schema is equivalent to the semantic in the logical and the conceptual schema.

1.2. Problem of Physical Database Design

During the past decades, DBMSs based on the relational model have moved from the research laboratory to the business place. One major strength of relational systems is its ease of use. Users interact with the systems in a natural way using non-procedural languages that specify what data are required, but do not specify how to perform the operations to retrieve data. Statements specify which tables should be accessed as well as conditions restricting which combinations of data from those tables are desired. They do not specify the access paths (e.g. indexes) to be used, to retrieve data from the tables, or the sequence in which tables are to be accessed. This is the job of the so called DBMS optimizer module. Hence relational statements can be run independent of the set of existing access paths.

There has been controversy about the relational systems (R-DBMSs) performances compared to other DBMSs. Especial in the transaction-oriented environment. Critics of relational systems point out that their non-procedural way prevents users from navigating through data the way they believe to be the most efficient. Developers of R-DBMSs claim that their system is capable of making the best decisions on how to execute the user requests based on statistical models of the database and cost estimating formulas. The system carries out analysis on executing cost alternatives. A software module, known as the ***optimizer***, makes execution decisions based on a statistical model of the database. It performs analysis of alternative execution plans for each statement and choose the one that appears to have the lowest cost. Optimizer efficiency, in choosing optimal execution plans, is critical to system response time. Initial studies[2] on the behavior of optimizers have shown that the choices made by the optimizer are among the best possible for the set of access paths.

The relational database system does not automatically determine an optimal set of access paths. The access paths must be created by the database designer. Access path selection is not trivial, since a database designer, more precisely the index designer, must balance the advantages of access paths for data retrieval versus their disadvantage in maintenance costs, incurred for database inserts, deletes, and updates and space utilization. For example, indexing all table *attribute*s is seldom a good choice. Updates will be very expensive in that design, and moreover, the index will probably require more total space than the table. A poor choice of physical designs can result in poor system performance, far below what could be expected if a more suited set of access paths was available. Hence a design tool and/or guidelines are needed to help designers selecting the right access paths that support efficient system performance for a set of applications and users.

Such a design tool would be useful for initial database design and for major reconfigurations of a database. A design tool might be helpful when:

[2] [Finkelstein 1988]

- The costs of future database must be evaluated,
- the database is to be loaded,
- the workload on a database changes substantially,
- new tables are added,
- the database has been heavily updated, or
- DBMS performance has degraded.

In this document we do not pretend to give the solution for such a tool. We only want the reader to get an overview of the major aspects related to physical database tuning and how they interact with one another. All recent DBMSs and CASE tools integrate features that help the database designer to model an efficient database.

Remember, that the major problem during physical design is the definition of an optimal set of access paths (indexes) according to the user requirements.

Data in a table can be accessed by scanning the entire table (brute force table scan). The execution of a given statement may be speeded up by using auxiliary access paths, such as indexes. However, the existence of certain index, although improving the performance of some statements, may reduce the performance of other statements (such as updates), since the indexes might be modified when tables are. In relational systems, some indexes, called *clustered index*, enforce the ordering of records in the table they index. All other indexes are called *nonclustered index*. The overall performance of the system depends on the set of all existing index, as well as on the way the tables are stored.

Given a set of tables and a set of statements, together with their expected frequencies of use and their filter factor or selectivity, the index-selection problem involves selecting for each table.

- The physical ordering for records (clustered index), and
- a set of secondary access paths (nonclustered index),

To minimize the total processing cost, subject to a limit on total index space.

Defining the total processing cost as, the frequency weighted sum of the expected costs for executing each statement. Including access, record update, and index maintenance costs. A weighted index space cost might also be added in.

Clustered indexes often provide excellent performance when they are on attributes referenced in a given statement. This might indicate that the solution to the design problem is to have a clustered index on every attribute. Such a solution is not possible, since records can be ordered only one way. On the other hand, nonclustered index can exist on all attributes and may help to process some statements. A set of clustered and nonclustered index on tables in a database is called an *index configuration*. An index configuration[3] is defined as, as set of indexes so that no table has more than one clustered index and no attributes have at same time clustered and nonclustered index. We will only be interested in index designs which are configurations. Let us call *index solution* the configuration proposed for a particular index selection problem.

It might seem that finding solutions to the design problem consists of choosing one attribute from each table as the ordering attribute, putting a clustered index on that attribute, and putting nonclustered index on all other attributes. This however, fails for three reasons.

- For each additional index, auxiliary maintenance cost is induced, every time updates are made upon an indexed attribute (inserting or deleting records, updating the value of the indexed attribute). Because of the maintenance cost, a solution with indexes on every attribute of every table usually does not minimize processing costs.
- Storage costs must also be considered even when there are no updates. According to [Finkelstein 1988] index use 5 to 20 percent of space used by the table indexes, so storage costs are not negligible.
- A global solution can not generally be obtained for each table independently. Any index decision that is made for one table (e.g. which index is clustered) may affect the best index choices for another table.

These considerations show that the design problem presented at the beginning of this section has no simple solution. According to [Finkelstein 1988] even a restricted version of the index-selection problem is in the class of NP-hard problems. Thus, there appears to be no fast algorithm that will find the optimal solution. However, we must question whether the optimal solution is the right goal, since the problem specifications and the problem the designer actually wants to solve are usually not identical. Input specifications to the design process are often rude estimation.

Specifications may include:

- Statements, as input to the problem, they usually represent estimations of actual load, that will be submitted to the system.
- Activation frequencies associated to the statements. They are commonly rude estimations as well.
- Statistics (dynamic and static) based upon the data, as it exists at a given time, are estimations that may not reflect future changes.

[3] [Finkelstein 1988]

- Statistical models used by the optimizer to estimate I/O costs. However, it is only correct for some given data distributions.

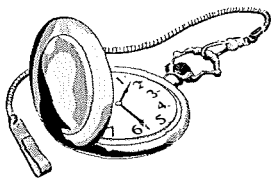
For these reasons, instead of striving for the optimal solution in the index design problem. We would like to get a set of *reasonable models*, each of which has a relatively low performance cost. From this set the designer can choose the one he thinks best, based on considerations that may not have been completely modeled. By an appropriate use of some heuristics and guidelines, combined with more exact techniques, the database designer can find rapidly a set of reasonable solutions.

1.3. Objective of the paper

Remember that an index represents a data structure that allows to increase the access performances to a set of given records. Therefore, proper index tuning is a must for performances hungry applications and reasonable query response times. Improper index definitions may lead to the following mishaps.

- Index that are maintained but never used.
- Tables that are entirely scanned in order to return a single record.
- Joins that take forever.
- Concurrency bottlenecks.

At the end of this paper the reader should be able to display and analyze an access plan chosen by the database system for specific queries and specific access structures. He should also understand what makes up a "good" or "bad" access structure (index solution). As a result, he will have a much better understanding of what "tuning" steps he can undertake to improve query execution and I/O costs: where and which kind of index can be added to improve access performances. Later on, we will give some guidelines in choosing, maintaining and using the right index structures for reasonable data access. All hints apply directly to the relational systems and can in most cases be applied to any commercial relational database system.



Chapter 2. Data Operations

In the chapter 1 we have seen that a major input to physical data modeling and tuning is based on requirements collection and analysis. Let us go further and say that physical database tuning used as input analysis and statistics on data requests, the so called *queries.* When tuning a database, we should define what tables the queries access. What are the query types, how many data do they qualify and how often are they activated?

It seems obvious that the usefulness of an index depends on how the queries use the index. For example, if there is an index on attribute A, but no query ever mentions A, then the index entails overhead (for maintenance on inserts, deletes and space usage) without yielding any benefit. This is obviously not correct. Less obvious misleading sources can result from placing the wrong kind of indexes on wrong attributes, however, with respect to the queries performed on those attributes. Placing the right index on the right attributes is not an easy job, because an infinite number of queries are possible and can be performed on a given database. Therefore, let us abstract the transactions and the queries into of classes or types.

Data operations, generally expressed in the Data Manipulation Language (DML), act upon data within the database. Relational DBMSs use Structured Query Language (SQL) as DML. Using SQL we can execute two major kind of operations, *data requests* (SELECT) and *data updates* (INSERT, UPDATE, DELETE). A third kind, is possible, the *macro operations*, they combine queries and updates, and in most cases are related to the notion of data integrity (protection against incidents, concurrency regulation, etc...).

Data operations are based on three major concepts.

- ***The database***. Representing the data. It is a mapping of the conceptual, the logical and the physical schema into real world. For relational DBMSs, databases are composed of tables and access structures, like indexes.
- ***The data object*** *(or table record).* Is a description of logical table entities. For example, it is possible that the object represents a record of a conventional file system or a table record in a relational system. An object stands also for an elementary unit of information asked by the user or retrieved by an application. Its content and composition varies corresponding to the data operation. For example the object corresponding to the data request *"Select all employees"* is the record type EMPLOYEE itself, whereas the object corresponding to the *query "Select all employees and their departments"* is composed of the record type EMPLOYEE and DEPARTMENT.

- ***The attribute and its value***. Attributes are proprieties given to an object, they define the record type. They have associated values, which are elements of a data within a data domain. It is possible to access the values of attributes throughout its associated object. At any time 0,1 or many values can be attached to one object. The value is typically a sequence of symbols that can be assigned to application or user variables. For example the record type EMPLOYEE having the following attributes EMPID, EMPNAME, EMPADR, EMPDEP, etc.... A specific object can be identified by the attribute values: EMPID = 1234, EMPNAME = ' Dupond Jean', EMPADR =' 14 Place Wiertz, B-5000 Namur', EMPDEP = 'COM'.

2.1. Data Access Operations

To understand the importance of the query classification for the physical database tuning, we first have to understand the concepts and the basic algorithms that take place when executing data access operations. We shall define two classes of query operations, depending on the size of the data object they reference, more precisely the number of records they qualify. After what we are able to abstract the queries into significant query "types"[4].

2.1.1. Data Access Processing

When a database system receives a query, it goes through a set of query ***compilation steps***, before it begins execution. In a first phase, we have what we call the ***syntax-checking***. The system analyses the query and checks its syntax, then it matches elements of the query syntax with views, tables, and attributes listed in the database system catalog, and performs appropriate query modification. During this process the system validates that the user has privileges and that the query does not disobey any relevant integrity constraints. A second phase, called the ***query optimization phase*** takes care of examining existing statistics for the tables and attributes. How many rows exist in the tables, how big are the records and how many records can an I/O block hold. Relevant access structures (indexes) are located within the database, and memory buffers are scanned for already existing data. After what, a complex procedure, which we can think of as *"figuring out what to do"*, produces a procedural access plan, the *execution plan*. The access plan is then executed during a third step, the ***execution phase***, wherein the indexes and tables are scanned to extract and/or derive the requested data object from the database.

2.1.2. Basic Algorithms for Executing Query Operations

A relational DBMS with a high-level query language interface (like SQL and Embedded-SQL) includes algorithms that translate the types of relational operations, which can appear in a query execution strategy. These strategies include the basic relational algebra operations (restriction, projection, product, union, intersection, difference, join, divide) [Date 1990] [Elmasri 1994] and in many cases combinations of these operations. The database system must also have algorithms for processing special operations such as aggregation and grouping functions. An algorithm may apply only to a particular storage structure and/or access path; if so, it can only be used in case where the tables involved in the operation include these specific storage and/or access structures.

[4]Keeping in mind that there exists an infinite number of query types.

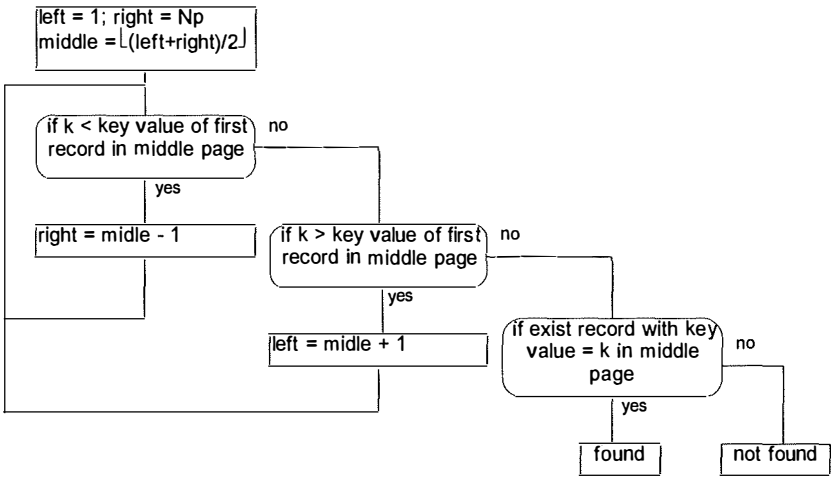
There are many options for executing a SELECT statement. They depend on the table access paths and may apply only to certain types of select predicates. The database optimizer chooses the best execution plan during the query *optimiz*ation phase depending on database statistics and on the data structures. The next sub-section will present some of the typical algorithms used by the database system to implement a SELECT statements.

2.1.3. Search Methods for Selection

Numerous search algorithms are possible for qualifying records within a table. One of the simplest is known as ***brute force table scan***, it scans all the records in the table to search and retrieve records that satisfy a given predicate. An other one involves the use of indexes, called the ***index scan***. Literature lists the following search methods (M1 through M5) to implement the data retrieval. Note that these ***are examples,*** to illustrate some of the major search methods that can be used by any commercial database system, to implement and execute queries.

M1. *Linear search (or brute force).* Scans all the data pages to qualify all records that satisfy the select filter; hence Np page[5] accesses. In case, we have an equality predicate on a unique key attribute, only half of the data pages are searched on the average before finding the records, do Np/2 accesses [Baudoin-Meyer 1984].

M2. *Binary search.* Can only be used if there is a physical ordering constraint on the data records. It is often used when the filter involves a comparison predicate. Assuming that the physical data page addresses are available in the file header, the binary search can be described by algorithm 2.1.. A binary search usually accesses $\log_2 Np$ pages, whether the record is found or not. Thus an improvement over brute force search, where in best case (when the record is found) an average of Np/2 pages are accessed and in worst case (when the record is not found) all Np pages are accessed [Date 1990] [Baudoin-Meyer 1984].



algorithm 2.1.: Binary search on an (unique) ordering key

[5] Np is the number of disk pages required to store Nr records. Later one we see how to calculate the number of pages required for the data file.

M3. *Using a primary index*[6] *to retrieve a single record.* If the select condition involves an equality predicate on an attribute with a primary index then the primary index structure is used to retrieve the record.

M4. *Using a primary index to retrieve multiple records.* If the comparison condition is $<, \leq, >$ or $\geq$ on a key attribute with primary index[7] then the index will be used to find the records satisfying the condition.

M5. *Using a secondary (B^+-tree) index.* On an equality predicate, this search can be used to retrieve a single record, if the indexing attribute has unique values (is a key) or to retrieve multiple records, if the indexed attribute is not a key. In addition, it can be used to retrieve records on conditions involving $<, \leq, >$ or $\geq$.

Method M1 applies on all kind of tables, whereas all other methods depend on the attribute access path involved in the select_filter. Methods M4 and M5 can also be used to retrieve records in a certain range. For example:

```
SELECT Name
FROM Employee
WHERE Salary > 30000 AND Salary < 50000.
```

If the condition of a SELECT operation is a conjunctive condition; that is when it is made up of several simple predicates connected with the logical operator AND, then the DBMS can use one of the following additional methods to evaluate the operation:

M6. *Conjunctive selection.* If on of the predicates involves an indexed attribute, the DBMS chooses first this predicate to access the table records using one of the above described methods. After retrieving the qualified records the DBMS checks the remaining predicates on those records.

M7. *Conjunctive selection using a composite index.* If many attributes are used by on of the connected predicates and a composite composite index, or hash structure exists on the concerned attributes, then the DBMS uses the index directly. For example in case where the index was defined on the composite index (Id, Name) of table.

M8. *Conjunctive selection by intersection of record pointers.* This method may be used when secondary indexes are available on all (or some of) the attributes involved in on of the connected predicates and when the indexes uses record pointers rather than block pointers[8]. Each index can be used to retrieve the record pointers that satisfy the individual predicates, then the intersection of the sets of qualified record pointers is used to retrieve the qualified records. Note that if only part of the predicates have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining predicates.

---

[6] A primary index is an index in which the record placement in the table is determined by the index values. Examples of primary indexes are clustered B-tree, ISAM indexes [O'Neill-1994] and hash index.

[7] Excluding the hash index.

[8] Note that, it is not necessary that the index uses record pointers it also work with page pointers. However, the retrived pages still has to be scanned for the records satifying all predicates, as the set of qualifed pages includes all pages that satisfy at least on of the predicates.

Whenever a single predicate specifies the selection, we can only check whether an access structure exists for the attribute involved in the predicate. If an access path exists, one of the access path methods is used, otherwise the brute force table scan method is used.

In all recent DBMSs, a module known as optimizer chooses the query execution plan based on the access structures, to retrieve data in the most efficient way. Therefore, during physical tuning, we need the notion of predicate filter factor, selectivity to helpful in the search for the right indexes. It allows us to foresee and to estimate approximations of the query I/O cost.

2.1.4. The Database Optimizer

To get a better understanding on data access costs, we have to shed some light on the database optimizer module. We already know that the optimizer comes into play during the query optimization phase. Out of this phase there are normally a large number of competing access plans, that can be executed to fulfill a given query, just as there are large number of ways to play a chess game with the object of winning (or at least not losing). The system query optimizer tries to choose an access plan with a minimal access cost plan, based on minimizing run time as well as various other resources, such as CPU time, number of disk I/Os, and so on. The optimizer uses a set of information and statistics that human programmers typically do not have or of which they only have rude approximations. The optimizer uses statistical information, such as the *cardinality of each attribute domain*, the *cardinality of each table*, the *number of values for each attribute*, the *number of times each value occurs for each attribute*, and so on. This information is kept in the system catalog [Date 1990]. Nevertheless, the query optimizer will probably not choose the optimal execution plan for complex queries, no more than a chess player plays the perfect game.

The query optimizer attempts to minimize the use of certain resources by choosing the best query execution plans. The resources are CPU time and physical accesses (the number of I/O required to execute the query). Though computer memory is an important resource, memory capacity for various purposes is not taken into account here, because normally buffer size is determined at system initialization by the DBA. Since the optimizer can have no affect on this feature, it usually reacts in a relatively simple way by choosing different types of behavior in query plans at various thresholds of memory availability.

The CPU and I/O resources are under control of the optimizer. For each alternative execution plan there is an associated *CPU cost*, noted $\mathsf{COST}_{\mathsf{CPU}}(\mathsf{PLAN})$, and an *I/O cost*, noted $\mathsf{COST}_{\mathsf{I/O}}(\mathsf{PLAN})$. Whenever, there are two incomparable costs it is possible that two query plans, PLAN_1 and PLAN_2, will be incomparable in resource usage. See figure 2.1..

	$Cost_{CPU}$	$Cost_{I/O}$
$PLAN_1$	9.2 CPU sec	103 I/Os
$PLAN_2$	1.7 CPU sec	890 I/Os

figure 2.1.: Two execution plans with incomparable CPU and I/O cost pairs[9]

Clearly $PLAN_1$ is superior to $PLAN_2$ in terms of CPU costs, but $PLAN_1$ is superior in terms of I/O costs. To provide a single measure that can be minimized unambiguously, the optimizer defines the *total cost* of execution plans, COST(PLAN), as the weighted sum of I/O costs and CPU costs.

$$COST(PLAN) = W_1 * COST_{I/O}(PLAN) + W_2 * COST_{CPU}(PLAN)$$

Where W_1 and W_2 are positive numbers, weighting the relative importance of each measure within the total cost.

The optimizer chooses the lowest COST(PLAN) value within all alternative execution plans that can answer the query. In chapter 4 we discuss how to analyze alternative access plans to derive relatively accurate associated I/O costs. It is not easy from a theoretical point of view to determine the associated CPU usage, as this feature depends strongly on details of CPU instructions, the efficiency of database system implementation and OS features. Of course, the optimizer for a specific DBMS an OS is able to compute CPU costs, using CPU statistics measured for internal functions. As a rule of thumb, we assume that *CPU costs do not vary as much from one access plan to another*. Note that in many situations CPU costs are linked to I/Os, so we can assume with no great harm that minimizing I/Os minimizes CPU as well. Thus implying that situation, for I/O and CPU costs, like the one in figure 2.1. are quiet unusual.

2.1.5. Filter Factor, Selectivity and Database Statistics

To define the selectivity let us use the definition made by [ORACLE 7.0]:

'Selectivity is the percentage of records in a table that the query selects. Queries that select a small percentage of a table's records have good selectivity, while a query that selects a large percentage of records has poor selectivity.'

Throughout this document we assimilate to the notion of selectivity, s, the term of filter factor, ff.

Before we can give estimations of the filter factor we have to make the following assumptions.

- Uniform distribution of individual attribute
- Independent join distribution of values from any two unallied attributes.

To determine the filter factor of a query predicate the optimizer uses the following sources of information.

[9] [O'Neil 1994]

- Operators used in the WHERE clause.
- Key and non-key attributes uses in the WHERE clause.
- Table and data statistics.

Let us go through some examples, to paint out how to estimate the filter factor for different predicates.

Consider an attribute A1, with 100.000 distinct values, noted CARD(A1) = 100.000. Assuming that all A1 values are equally distributed within table T1 (the uniform distribution assumption), we estimate the filter factor for the equality predicate, A1 = 5, to:

$ff_{T1.A1 = 5} = 1/100.000 = 0.00001$

Making the same considerations, we are able to estimate the filter factor of a between predicate, A1 between 5 and 505:

$ff_{A1 \text{ between } 5 \text{ and } 505} = 501 * (1/100.000) = 0.005$

Similarly consider the key attribute A2, having 100 distinct values CARD(A2) = 100. And the predicate, A2 = 20, having a filter factor of:

$ff_{A2 = 20} = 1/100 = 0.01$

Let us assume that the join distribution of values from *two unallied attributes* is independent, meaning that the filter factor for compound AND predicates multiply, so:

$ff_{A2 = 20 \text{ And } A1 \text{ between } 5 \text{ and } 505} = (1/100) * (500/100.000) = 0.00005$

Note that we borrowed the filter factor terminology from DB2 and [O'Neil-1994], which base there optimizer filter factor estimations on statistics gathered by the DB2 RUNSTATS utility.

Figure 2.2. lists some of the statistics given by the RUNSTATS utility[10]. For each statistic we list its name, the DB2 catalog table and the attribute which holds it. Each of these statistics have default values in case where RUNSTATS has not been run.

Catalog Name	Statistic Name	Default Value	Description
SYSTABLES	CARD	10.000	Number of records in the table
	NPAGES	Ceil(1+CARD/20)	Number of data pages that contain rows of the table
SYSCOLUMNS	COLCARD	25	Number of distinct values for this attribute
	HIGH2KEY	N.A.	Second highest value for the attribute
	LOW2KEY	N.A.	Second lowest value for the attribute
SYSINDEX	NLEVELS	0	Number of levels of the index
	NLEAF	CARD/300	Number of leaf pages
	FIRSTKEY-CARD	25	Number of distinct values in the first attribute, A1, of the key
	FULLKEY-CARD	25	Number of distinct values in the full key, (A1,A2,A3)
	CLUSTERED-RATIO	0% if non-clustered 95% if clustered	Percentage of records of the table that are clustered by the index values

figure 2.2.: Statistics given by RUNSTATS utility for Access Plan Determination

[10] Similar statistics can be gather by similar statements for different database system, for example the EXPLAIN statement in ORACLE 7.

Note that these statistics can only be used on existing databases, however, they give us reference of what we have to consider when estimating the filter factor and I/O costs.

Considering the above statistics, DB2 considerations and our reference to [O'Neil 1994], we can give a list, in relation 2.1., of different predicates and their corresponding relations for filter factor estimations.

Predicate Type	Filter Factor	Notes
Attr = const	1/COLCARD	'Attr <> const' is equal to 'Not(Attr = const)'
Attr θ const	Interpolation relation	θ is a comparison predicate other than equality.
Attr < const or Attr > const	$\frac{(const - LOW2KEY)}{(HIGH2KEY - LOW2KEY)}$	LOW2KEY and HIGH2KEY are estimations for extreme points of range of Attr values
Attr BETWEEN const1 AND const2	$\frac{(const2 - const1)}{(HIGH2KEY - LOW2KEY)}$	'Attr NOT BETWEEN const1 AND const2' is equal to 'NOT(Attr BETWEEN const1 AND const2)'
Attr in list	size(liste) / COLCARD	'Attr NOT in list' is equal to 'NOT(Attr in list)'
Attr is Null	1/COLCARD	'Attr is NOT Null' is equal to 'NOT(Attr is Null)'
Attr like 'patern'	Interpolation relation	Based on the alphabet
Pred1 and Pred2	$ff_{Pred1} * f f_{Pred2}$	As in probability
Pred1 or Pred2	$(ff_{Pred1} + f f_{Pred2}) - (ff_{Pred1} * ff_{Pred2})$	As in probability
NOT Pred1	$1 - ff_{Pred1}$	As in probability

relation 2.1.: Filter Factor relations for Various Predicate Types

Knowing the estimation of the filter factor, for a given SQL query predicate, we are able to estimate, per se, the average number of records qualified by the predicate, k, by multiplying the filter factor and the number of records in the table.

$k = ff * Nr$

where

ff: Filter Factor for predicate P

Nr: Number of records in table

relation 2.2.: Number of Records Qualified knowing the Filter Factor

It is obvious that the lower the value of the filter factor is the fewer records the predicate qualifies. It is also true that the lower the filter factor is the higher is the probability that the predicate will be used first during query execution.

At the beginning of the section we made the assumption of uniform distribution, however, this assumption is not always valid. For example, take an extreme situation where we have a sex attribute in a table containing all residents at a boy school. Although there are occasional residents with sex = 'F', staff and faculty members for example, it is clear that a filter factor estimated in terms of 1/CARD(sex) = 1/2 is misleading. A query optimizer that uses this assumption may very well make incorrect decisions. For this reason, DB2 and a number of other database systems, such as INGRES, provide statistics on individual attribute values that deviate strongly from the uniform assumption. It is therefore important to locate and to note such attributes during the requirements collection phase.

Note that for the rest of the paper we assume that all attribute values are uniformly distributed among the data pages. Note also that it is easy to imagine that exact estimations of the filter factor for all predicates are not available per se. They are often kept in the DBMS catalog tables. Thus, for databases at conceptual level, estimations have to be performed. The database designer uses its requirement collection to foresee the query predicate filter factors.

2.1.6. Description of SQL Select Statement

Before we start classifying the query data operations into classes, we should examine the structures and the possibilities of the SQL Select statement. Figure 2.3, gives a general form of the Select statement, and we will develop its syntactic elements within the following lines.

Subselect Statement

```
SELECT [all | distinct] expression {, expression}
FROM tablename [corr_name] {, tablename [corr_name}
[WHERE select_filter]
[GROUP BY column {, column}]
[HAVING select_filter]
```

Full Select Statement

```
SUBSELECT
[UNION [ALL] SUBSELECT]
[ORDER BY result_column [asc | desc] {, result_column [asc | desc]}
```

The ORDER BY clause allows us to place qualified records in order by one or more result_column values appearing in the target list. The [asc | desc] choice enables us to place records in ascending or descending order; asc is the default and means that smaller values are placed before higher values.

Note, that the UNION clause comes before the ORDER BY clause. The order of clauses within the Select statement allows us to define a conceptual order of query evaluation. Remind that the following order of evaluation might be different from the actual order chosen by a DBMS optimizer.

Step 1: First the Cartesian product of all tables in the From clause is formed.
Step 2: From this, records not satisfying the Where condition are eliminated.
Step 3: The remaining records are grouped according to the Group By clause.
Step 4: Groups not satisfying the Having clause are then eliminated.
Step 5: The expressions of the Select clause are evaluated.

Step 6: If the key word Distinct is present, duplicate records are eliminated.
Step 7: The Union is taken after each Subselect is evaluated.
Step 8: Finally, the set of retrieved records is sorted if an Order By is present.

2.1.6.1. Expressions, Predicates and select_filter

The select_filter is the condition used in the WHERE clause to eliminate records and in the HAVING clause to eliminate groups: records are retained in step 2 and groups in step 4 when the corresponding select_filter evaluates to TRUE.

Let us start, by describing the syntax element known as an expression (expr); that is either an arithmetic or a character expression (expr = aexpr | cexpr). An expression occurs in a select_filter, for example, when we compare an attribute value to a constant: T1.A1 > 100: both T1.A1 and 100 are simple expressions. Note, that expressions, as we define them here, can also appear in the target list of the select statement.

An aexpr is an arithmetic expression, made up of constants, table attributes, arithmetic operators, built-in arithmetic functions, and/or set of functions. Figure 2.4. gives the definition of an arithmetic expression.

aexpr	Examples
constant	6, 7.00
column_name	Dollars, Price, Percent
qualifier.column_name	Orders.Dollars, P.Price
aexpr arith_op aexpr	7.00 + Product.Price
(aexpr)	(7.00 + Price)
function(aexpr)	sqrt(7.00 + Price)
set_function(aexpr)	sum(Price)

figure 2.4.: Recursive Definition of the Arithmetic Expression (aexpr)

Similarly as for the arithmetic expression figure 2.5. gives the definition of a character expression, cexpr:

cexpr	Examples
constant	'Namur', 'Dupond'
column_name	Id, Salary, City
qualifier.column_name	Employee.Id, City
cexpr op cexpr	Employee.Id + 'Namur'
(cexpr)	(Employee.Id + 'Namur')[11]
function(cexpr)	right ('Namur',4)[12]
set_function(cexpr)	count(distinct Employee.Salary)

[11] concatenate two strings with a + operator

[12] result = 'amur'

figure 2.5.: Recursive Definition of a Character Expression (cexpr)

Out of the SQL standards we can group (figure 2.6) seven kinds of *predicates*, which is the simplest form of logical statements.

Predicates	Form	Examples
comparison predicate	expr1 θ[13] (expr2 | SUBSELECT)	Employee.Salary > (SUBSELECT)
between predicate	expr1 [not] between expre2 and expr3	Salary between 40000 and 70000
quantified predicate	expr θ [all | any] (SUBSELECT)	Salary >= all (SUBSELECT)
in predicate	expr [not] in (SUBSELECT)	Id in (SUBSELECT)
	expr [not] in (val (, val))	City in ('Namur', 'Liège')
exists predicate	[not] exists (SUBSELECT)	exists ((SUBSELECT)
is null predicate	column_name is [not] null	discnt is null
like predicate	column_name [not] like 'pattern'	Employee.Name like 'A%'

figure 2.6. Predicate of Standard SQL

Given these predicates, we define the select_filter as it is in figure 2.7.:

select_filter	Examples
predicate	Employee.Id = 12345, exists (SUBSELECT)
(search condition)	(Employee.Id = 12345)
not select_filter	not exists (SUBSELECT)
select_filter and select_filter	not (Employee.Id = 12345) and Dep = 'C001'
select_filter or select_filter	not (Employee.Id = 12345) or Dep = 'C001'

figure 2.7.: Recursive Definition Select_Filter

2.1.7. Two Query Classes

Now that we have defined and described the Select statement, and its aspects of referencing data objects (records) within the database, we will classify the query operations.

The access is often only part of a general step which leads to data extraction or data modification. In general the data access operation can be defined as the *access to objects of a sequence*. All possible and imaginable data accesses can be based upon this definition.

Note that we point the data access definition on the notion of sequence. A sequence of records can be defined as a set of records following a logical ordering. However the ordering is not mandatory. The set of qualified records is defined by a select_filter, predicate within the WHERE clause. The ordering of the records is often the natural ordering of the database (the order of a table or the order of an access key). But it can also be explicitly specified in the Select statement using the ORDER BY clause. On a more common way, the user can specify an *implicit* or *explicit* ordering of the qualified set of articles.

[13] The comparison operator belongs to the following set of operators {=, <>, >, >=, <, <=}

A set of qualified records is defined by a group expression, a select condition, a predicate. The set of records can be qualified using an *access mechanism*, like index or hash structures, or by a *filter*, like "brute force" table scans. Both mechanisms and there respective cost estimations will help us to understand and solve the index selection problem.

In practice we encounter two kind of access classes:

- First kind, is the *access to a set of records*: this operation gives access to a set (or sequence) of records. The requested set of records can be ordered in an implicit or explicit way.
- Second kind, is the *access to one record*: this operation gives access *to one and only one record* of a given set of records. The access is specified by a sequence, embodying the questioned record, and a position of the record.

2.1.8. Abstract the Queries into a Few Query "Types"

The preceding sections have revealed that there exists an infinite number of queries that can be constructed using predicates as building blocks. Therefore, it is helpful to abstract the queries into the *8 most used query types* [Shasha 1992]. Later on, we will examine the strengths and the possibilities of each kind with regard to the different data access structure that we will encounter.

1. ***Point query.*** The query returns at most one qualified record (or part of a record), based on an equality comparison predicate (=). If we assume that attribute A1 is a key (uniqueness of values) then the following query is a Point Query:

```
SELECT A1, A2
FROM T1
WHERE A1 = 5
```

2. ***Multipoint query.*** The query returns a list of qualified records based on an equality predicate. The list of records can be order depending on an implicit or explicit ordering logic. If we assume that attribute A2 is not a key (non-uniqueness of values) then the following query is a Multipoint Query:

```
SELECT A1, A2
FROM T1
WHERE A2 = 20
```

3. ***Range query or Between query.*** The query returns a set of qualified records where A2 values lie within an interval or half-interval. Assuming that attribute A2 is not a key the following queries are Range Queries:

```
SELECT A1, A2
FROM T1
WHERE A2 BETWEEN 20 AND 60
```

```
SELECT A1, A2
FROM T1
WHERE A2 >= 20
```

4. ***Prefix match query.*** The query returns a set of qualified records based on a set of attributes, A, by specifying a prefix on A. For example, consider the sequence of attributes A3, A4, A5 (with respect to the ordering). The following are prefix match predicates for the set of attributes:

```
A3 = 'Gates',
```

```
A3 = 'Gates' AND A4 = 'Bill'
A3 = 'Gates' AND A4 LIKE 'Bi%'
```

5. ***Extremal query:*** the query returns a list of qualified records (or parts of records) whose attribute values (or set of attributes values) are a minimum or maximum.

```
SELECT A1, A2
FROM T1
WHERE A2 = MAX(SELECT A2 FROM T1)
```

6. ***Ordering query.*** The query includes the ORDER BY clause, it returns a set of qualified records in ascending or descending order (asc | desc) of a set of specified attribute.

```
SELECT A1, A2
FROM T1
ORDER BY A2
```

7. ***Grouping query.*** The query uses the GROUP BY clause, it partitions the results of a query into groups. This kind of queries is often used during report generation and applies many cases a function to each set of grouped values, records. For example, find out the average salary for each department.

```
SELECT A1, AVG(A2), A3
FROM T1
GROUP BY A1, A3
```

8. ***Join Query or Query involving one ore multiple Subselect(s).*** The query links two or more tables. Loosely spoken, the query retrieves data from more than one table. The evaluation of join queries is explained during a future section. Note that join queries represent one of the most powerful features of the relational system.

   If the predicates linking the table are based on an *equality comparison predicate*, the join query is called an *equijoin query*.

```
SELECT T1.*, T2.*
FROM T1, T2
WHERE T1.A1 = T2.A7
```

   The result of this query is said to be a join of table T1 and T2 over matching attributes T1.A1 and T2.A7 values. The equijoin by definition must produce a result containing two identical attributes, if one of those two attributes is eliminated we speak about *natural join.*

   There is no need that the comparison predicate within a join condition is an equality comparison predicate, though, in most cases it will be.

   Many additional predicates can exist in concordance with a join predicate. The following query is an example: join table T1 and T2 with over attribute T1.A1 and T2.A7, but omitting all T1 records with A2 = 20.

```
SELECT T1.*, T2.*
FROM T1, T2
WHERE T1.A1 = T2.A7
AND T1.A2 <> 20
```

2.1.9. Methods for Joining Tables

The join query is one of the most powerful features of relational systems. Therefore, it is important to study the meaning of joining and the algorithms[14] for joining two tables. First, let us give an accurate definition [O'Neil 1994] of the *join operation*.

We define a join of two tables to be a process in which we combine records of one table with records of another to answer a request. In the following definition we represent the join operation by the symbol ⊕.

Consider the table R and S, defined as

$T1 = A_1 \ldots A_n B_1 \ldots B_k$
$T2 = B_1 \ldots B_k C_1 \ldots C_m$

where $n, k, m \geq 0$. Note that $B_1 \ldots B_k$ is the complete subset of attributes shared by the two tables[15]. The join of the table R and S is the table represented as $R \oplus S$, defined as

$$T1 \oplus T2 = A_1 \ldots A_n B_1 \ldots B_k C_1 \ldots C_m.$$

A record is in the joined table, *if and only if* there are two records u in R and v in S, such that $u[B_v] = v[B_u]$ for all v, $1 \leq v \leq k$; then attribute values on the record t are defined as follows:

$t[A_i] = u[A_i]$ for $1 \leq i \leq n$,
$t[B_i] = u[B_i]$ for $1 \leq i \leq k$,
and $t[C_i] = u[C_i]$ for $1 \leq i \leq m$.

When the record u in T1 and v in T2 gives rise to a record t in $T1 \oplus T2$, the two records are said to be *joinable*.

By this definition, a join occurs whenever two or more tables appear in the FROM clause of a Select statement. Even if we are taking a simple cartesian product of records from two tables (a table product); we refer to it as a join. As we will see, a Select statement with a single table in the FROM clause, and a WHERE clause that contains a SUBSELECT from a different table, is often converted by the query optimizer to an equivalent query statement that joins tables.

Now that we have seen a definition of joining tables, let us examine three algorithms used to join tables. The algorithms are known as ***nested loop join***, ***merge scan join***, and ***hybrid join*** [O'Neil 1994]. Each of these methods has performance advantages in a certain class of situations that can arise during join query evaluation. Other methods have been developed for performing joins[16], they surely have performance advantages in special circumstances, but we will restrict our attention to the three denoted join methods.

To begin with, we consider the situation where exactly two tables appear in the FROM clause.

---

[14] We concentrate on three algorithms used by DB2.

[15] This subset may be empty if k = 0.

[16] A special method for example is the hash join.

A join of two tables usually occurs in two steps. During the **first step**, only one table is accessed; this is referred to as the *outer table*. In the **second step**, records of the second, *inner table* are combined with records of the first, outer table. Other predicates, involving attributes of the two tables that have not been retrieved through an access structure, are used to qualify records as they are scanned. As a result of all this, a *composite table* is generated that contains all the qualified records of the join. If a join with a third table is now necessary, the composite table becomes the outer table for the succeeding join step. Otherwise, specified attributes of the composite table provide the result of a join being fully materialized in a disk work file, it is important to realize that we may be able to avoid such wasteful materialization. For example, if the user is only likely to look at the first 20 or 30 records of the resulting output, in this case it would be terribly inefficient to put to disk all records of the composite table. Thus in Embedded SQL, when a cursor on a join query is first opened and the first record is retrieved, we avoid materializing tables whenever possible.

2.1.9.1. Nested Loop Join

Consider the following query:

```
SELECT T1.A1, T1.A2, T2.A3, T2.A4
FROM T1, T2
WHERE T1.A1 = 5 AND T1.A2 = T2.A3
```

In a nested loop join, the table referred to as the outer table corresponds to the 'outer loop' in a nested pair of loops, as we see in figure 2.7.. Assuming that table T1 is the outer table, the first step of the nested join determines records in T1 that verify predicate T1.A1 = 5. We are able to retrieve the requested records from table T1 using one of the search methods seen in section 2.1.3..

Now that the records of the outer table have been qualified (note that they have not actually been extracted), a loop is performed to retrieve each of these records. For each qualified record of the outer table, a request is performed on the second table, T2, and all records that satisfy the join predicate, T1.A2 = T2.A3, are retrieved.

Note that because the record of T1 is fixed for this request, we can treat the value T1.A1 as if it were a constant, K. Therefore, the records retrieved from T2 are exactly those that satisfy a predicate of the form, T2.A3 = K, and an index on attribute T2.A3 would probably make the retrieval more efficient.

```
R1:  FIND ALL RECORDS T1.* IN THE OUTER TABLE T1 WHERE A1 = 5;
     FOR EACH RECORD T1.* FOUND IN THE OUTER TABLE;
R2:      FIND ALL RECORD T2.* IN THE INNER TABLE WHERE T1.A2 = T2.A3
         FOR EACH RECORD T2.* FOUND IN THE INNER TABLE
             RETRIEVE: T1.A1, T1.A2, T2.A3, T2.A4
         END-FOR;
     END-FOR;
```

algorithm 2.2.: Algorithm for Nested Loop Join

Note that label R1 and R2, in algorithm 2.2., designated retrievals in join processing. Additional predicates limiting the records of either table can be added to the relevant retrieval. Either retrieval can be performed using an index scan or a table scan. The outer table has only one retrieval, while the inner table has a number of retrievals equal to the number of qualifying records in the outer table. The I/O cost of the join operation is therefore given by relation 2.4..

$$\text{COST}_{I/O}(\text{NESTED LOOP JOIN}) = \text{COST}_{I/O}(\text{OUTER TABLE RETRIEVAL}) + \text{NUMBER OF QUALIFYING RECORDS IN OUTER TABLE} * \text{COST}_{I/O}(\text{INNER TABLE RETRIEVAL})$$

relation 2.4.: Cost estimation for Nested Loop Join

Figure 2.8. illustrates the method of nested loop join for the above query using table T1 and T2.

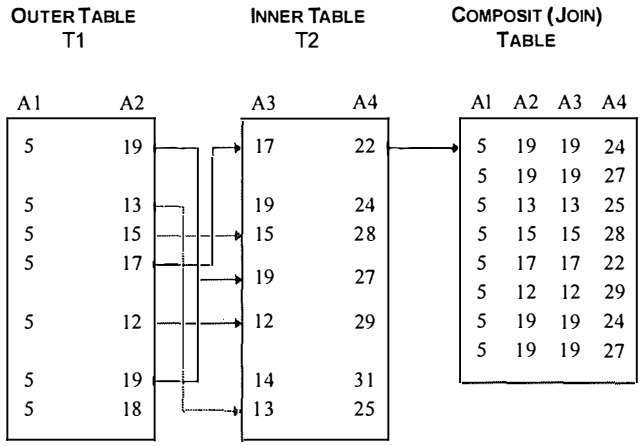


figure 2.8.: Illustration of Nested Loop Join

To speed up the execution of the nested loop join it is helpful to define *an index on the matching attributes of the inner table*. Nested loop join is particularly efficient when only a small number of rows qualify from the outer table after the limiting predicates are applied. This means that *the predicate on the outer table should have a small filter factor or a large selectivity*, or when the inner table is small enough so that the entire index and data disk blocks could be held in memory buffers after they have been accessed once during the join.

2.1.9.2. Merge Join

To examine the merge join method, let us consider the following query:

```
SELECT T1.A1, T1.A2, T2.A3, T2.A4
FROM T1, T2
WHERE T1.A1 = 5 AND T2.A4 = 6 and T1.A2 = T2.A3
```

The merge join scans the two tables, T1, T2, only once, in the order of their join attributes. The execution plan starts by applying the two non-join predicates and creating two intermediate tables, IT1, IT2. In the execution plan, we first evaluate 'SELECT A1, A2 FROM T1 WHERE A1 = 5 ORDER BY A2', placing the requested records in the intermediate table IT1 with attributes A1 and A2, and in sorted order of A2. Then we evaluate 'SELECT A3, A4 FROM T2 WHERE A4 = 6 ORDER BY A3', to get the intermediate table IT2 with attributes A3 and A4, and sorted on A3'. Note that these intermediate tables are usually written to disk work files as temporary tables, as they are generally too large to be hold in memory buffer.

Now that we have two smaller tables, than the original tables, we are prepared to perform the merge join. To perform the merge join on IT1 and IT2, we associate a pointer to the first records for each intermediate table. As the algorithm proceeds, the two pointers move forward in a way that any matching (A2, A3) values for records in both tables are detected. Except for cases where multiple A2 identical values in IT1 match multiple A3 identical values in IT2, both pointers move steadily forward through the records of both tables, and detect all matchings, IT1.A2 = IT2.C3, that exist. The pseudo-code of algorithm 2.3. describes the execution algorithm of the merge join method, the A2 value of record in table IT1 pointed to by pointer P1 is represented by of P1 → A2, and similarly for P2 → A3 in table IT2.

```
CREATE TABLE ITL AS: SELECT A1, A2 FROM TL WHERE A1 = 5 ORDER BY A2;
CREATE TABLE IT2 AS: SELECT A3, A4 FROM T2 WHERE A4 = 6 ORDER BY A3;

SET P1 POINTER TO FIRST RECORD OF IT1;          /* OUTER TABLE          */
SET P2 POINTER TO FIRST RECORD OF IT2;          /* INNER TABLE          */

MJ: WHILE (TRUE) {                              /* LOOP UNTIL EXIT MJ LOOP */
  WHILE (P1 → A2 > P2 → A3) {                   /* IF P2 NEEDS TO ADVANCE */
    SET P2 TO NEXT RECORD IN IT2;               /* ADVANCE IT           */
    IF (P2 PAST LAST RECORD) EXIT MJ LOOP;      /* OUT OF ROWS, EXIT    */
  }
  WHILE (P1 → A2 < P2 → A3) {                   /* IF P1 NEEDS TO ADVANCE */
    SET PL TO NEXT RECORD IN IT1:               /* ADVANCE IT           */
    IF (PL PAST LAST RECORD) EXIT MJ LOOP;      /* OUT OF ROWS, EXIT    */
  }
  IF (P1 → A2 == P2 → A3) {                     /* FOUND MATCH ON JOIN  */
    MEMP = P2;                                  /* REMEMBER P2 START POINT */
    WHILE (P1 → A2 == P2 → A3) {                /* LOOP                 */
      RETRIEVE: ITL.AL, ITL.A2, IT2.A3, IT2.A4;
      SET P2 TO NEXT RECORD IN 1T2;             /* ADVANCE P2 */
    }                                           /* LOOP CONTINUES IF P1 → A2 */
                                                /* UNCHANGED            */
  }                                             /* DONE WITH JOIN MATCH */
  /* SINCE FELL THROUGH, P2 → A3 IS NEW OR BEYOND END OF TABLE          */
  SET PL TO NEXT RECORD IN ITL;                 /* ADVANCE PL */
  IF (PAST LAST RECORD) EXIT MJ LOOP;           /* OUT OF ROWS, EXIT    */
  IF (P1 → A2 == MEMP → A3)                     /* IF NEXT P1 → A2 IS SAME */
    P2 = MEMP;                                  /* START OVER WITH P2   */
}                                               /* END OF MJ LOOP       */
```

algorithm 2.3.: Algorithm for the Merge Join Method

Once a match has been found during execution, we keep P1 fixed and advance P2 through all duplicate values. Then we advance P1; if we find a duplicate, this is the only situation in which a pointer moves backward, we set P2 = MEMP to run through all duplicates of P2 again. Clearly if there are a lot of occurrences where A2 and A3 have the same values, a large number of records will be joined. However, it is more common that there will be a small number of records in one table matching more than one record with another, since we normally do not perform joins on attributes with a large number of duplicate values. In any event, the query optimizer can determine the likely number of duplicates facing each other using statistics, and it is likely that most computer resources will be used in finding any match at all.

Note that it is *not always necessary to extract the records* from table T1 or T2 into intermediate tables. If for example, there was an index on attribute A1 of table T1, which allows us to qualify the records with predicate 'T1.A1 = 5', the execution plan would use an intermediate table as it can access the qualified records in order by T1.A1. This would be possible, for our Select statement, if T1 had an index on (A1, A2): the matching scan through the index with the given predicate would provide all records of T1 in order by A2. The same consideration holds for table T2.

2.1.9.3. Hybrid Join

The hybrid join method is used less frequently than both other join methods, and to avoid confusing the reader the reader we shall give only a short description of the method.

The hybrid join uses also an outer table and an inner table as the nested loop and the merge join. The first step is the same as that of merge join for the outer table. The table is scanned once according to the join attribute order, either through an index or after extracting a set of records, qualified by a predicate, into an intermediate table IT1. As records of the outer table are being scanned in the join attribute order, matching join attribute values of the inner table are looked up through an index on the join attribute. The records of the inner table are not accessed yet, however; instead the records from the outer table, with an additional attribute giving the RID value of each matching join record in the inner table, are written to an intermediate table IT2. Records of IT2 are stored in record pointer (RID) order, and the technique of list prefetch can be used to retrieve records from the inner table to join with the outer table records.

The advantage gained over the nested loop join arises from the fact that all records from the inner table can be performed using list prefetch I/O with large I/O pages.

2.1.9.4. Example[17] of cost estimation for Nested, Merge and Hybrid Join

Assume that we have two tables T1 with attributes A1 and A2, and table T2 with attributes A3 and A4, each with a cardinality of 1.000.000 records of 200 bytes each. We are going to estimate the I/O cost of the query, in figure 2.9 and 2.10. for all three join methods. We will measure the estimations in terms of elapsed time in seconds and pages requested, broken down into random (R), sequential (S) and list prefetch I/Os (L)[18].

```
SELECT T1.A1, T1.A2, T2.A3, T2.A4
FROM T1, T2
WHERE T1.A1 = 5 AND T2.A4 = 6 AND T1.A2 = T2.C3
```

figure 2.9.: Query used to Estimate I/O Cost for Nested Loop, Merge Join

Let us make some assumptions on the tables. We assume that a non-clustering index A1X exists on attribute A1 of table T1, and an index A3X and A4X on table T2 respectively on attribute A3 and A4. Assume that the *filter factors* for these predicates are given as follows:

- $ff_{A1 = const} = ff_{A4 = const} = 1/100$;
- $ff_{A2 = const} = 1/250.000$; and
- $ff_{A3 = const} = 1/500.000$.

We will estimate the I/O cost in terms of elapsed time for a nested loop, a merge and an hybrid join plan to answer the query, where the outer table is T1 and the inner table is T2.

2.1.9.4.1 Estimating I/O cost for Nested Join Method

For the merge join plan we consider the following three steps[19]:

(1). Using index A1X, retrieve all records from table T1 which verify predicate T1.A1 = 5.

(2). Think of T1.A2 as being a constant, K. For each record retrieved from the outer table T1 all records using index A3X from the inner table T2 such that T2.A3 = K. As the records from this index scan are retrieved, further restrict the records using predicate T2.A4 = 6.

(3). Print out T1.A1 and T1.A2 from outer table record and T2.A3 and T2.A4 for the qualified inner table record.

---

[17] [O'Neil 19994]

[18] Rules of thumb for I/ rates, Random I/O 40 pages/sec, Sequential I/O 400 pages/sec, List prefetch I/O 100 pages/sec

[19] [O'Neil 19994]

Using the filter factor and the number of records in T1, we are able to estimate the number of records retrieved from table T1 in step (1): (1/100) * 1.000.000 = 10.000 records, likely to be all on different pages (because of the non-clustering assumption). We can assume that the optimizer will use a list prefetch[20] to retrieve the qualified records. The index I/O cost for this retrieval is assumed to be insignificant next to the data page I/O cost. We therefore assume that $COST_{I/O}$(OUTER TABLE RETRIEVAL) = 10.000 L, where L is the fraction of time need to perform a list prefetch read[21]. Thus the elapsed time is equal to 10.000/100 seconds.

For each outer table record qualified, we assume that the value T1.A2 is in the range of values for attribute T2.A3. Since f(A3 = const.) = 1/500.000, we expect to retrieve 2 records out of a table with CARD(T2) = 1.000.000 records. This requires for each new value of T2.A3 one random I/O to the leaf level of index C3X, assuming that upper-level nodes are in memory buffers. Plus two I/Os, on average, to retrieve the two pages containing the two qualified records. Thus, the I/O cost for the 10.000 different inner loop steps is 10.000 * (1R + 2R), where R is the fraction of time need to proceed a random read[22]. We would normally perform a list prefetch I/O to retrieve the data pages in this situation, but recall that it is quiet misleading to think of a list prefetch of two pages as taking place in 2/100 seconds, since there are too few pages retrieved to amortize the retrieval as equivalent to 2R, and therefore the elapsed time for inner loop is calculated from 10.000 * 3R, or 30.000/40 = 750 seconds. An approximation of the elapsed time is 100 sec + 750 sec = 850 seconds.

Now to determine how many records are retrieved in the query, we see that there are 10.000 records retrieved from T1 and for each qualified records in T1 there are two records joined to it, on the average, from table T2. Therefore, there are about 20.000 records retrieved at this point, after which a qualification test takes place to see if predicate T2.A4 = 6 is verified. With a filter factor of 1/100, the final number of records retrieved is (1/100)*20.000 = 200 records.

2.1.9.4.2. Estimating I/O cost for Merge Join Method

We consider the same assumptions than for the nested join example. The strategy for answering the query in figure 2.9. with a merge join consists of following steps.

(1). Using index A1X, retrieve all records from table T1 which verify predicate T1.A1 = 5. Again, there will be 10.000 records retrieved. Output the A1 and A2 values to an intermediate table IT1, and sort the result by A2 values.

(2). Using the index A4X, retrieve all records from T2 where predicate T2.A4 = 6 is verified, there are 10.000 qualified records, output the resulting A3 and A4 values to intermediate table IT2, and sort them by A3 values.

Now that two intermediate tables have been created, the merge join step may take place, following algorithm 2.3.

---

[20] Note that the list prefetch is an access techniques used to speed up data retrieval. It will be given a deeper look in section 3.1.2..

[21] In section 3.1.2. we see that the a list prefetch fraction, L, is valuated to 1/100.

[22] See section 3.1.2..

As for the nested loop join, step (1) requires a read of 10.000 entries from the T1.A1 index leaf level, which we treat as insignificant, followed by an I/O cost of 10.000 accesses to retrieve the indexed records in the different data pages. If we assume that each of the A1 and A2 values extract requires 10 bytes , the materialized table IT1 requires 200.000 bytes, or about 50 data pages. Without no great harm we can think of performing the sort in memory, so the total I/O cost of step (1) is equal to 10.000L. The same considerations apply to step (2). The total cost for the plan is therefore 20.000L, with an elapsed time of 200 seconds, an improvement over the nested loop join, which required 850 seconds. The advantage comes from using the index A4X in the merge join for more efficient batch retrieval from table T2.

We saw that in this case the merge join is better in I/O performance than the nested loop join. However consider the query in figure 2.10. where predicate T2.A4. = 6 does not exist.

```
SELECT T1.A5, T1.A2, T2.*
FROM T1, T2
WHERE T1.A5 = 5 and T1.A2 = T2.C3
```

figure 2.10.: Query used to Estimate I/O Cost for Nested Loop, Merge and Hybrid Join

We assume that indexes, A2X, A3X and A5X exist on the corresponding attributes of table T1 and T2. The filter factors are the same than the preceding ones, plus a new filter factor $ff_{T1.A5 = const} = 1/1000$.

The I/O cost for a nested loop join with T1 as the outer table can be calculated as follows. With 1000 records to look up in table T1 verifying predicate T1.A5 = 5, we have an index look up cost of 1R access for the index and 1000L accesses for the data pages. For each record in the outer table T1, we set T1.A2 = K, then look up records in the inner table T2 having T2.A3 = K, about two records, requiring 1R access for the index leaf entry look up and 2R accesses for the record retrieval. The I/O cost for the inner loop is calculated as 1000*3 = 3000R accesses. Thus the total nested loop join cost is 3001R + 1000L, with elapsed time of 3001/40 + 1000/100, or approximately 85 seconds.

For the merge join, we calculate costs as follows. We can easily calculate that the extraction of IT1 requires 10 seconds for I/O, but as we will see, this is insignificant in comparison to the elapsed time for the nested loop join. Since there were no independent limiting predicate on T2, such as T2.A4. = 6, we have the choice between accessing the records from T2 in order by the index A3X to perform the merge, or materializing and sorting the entire table T2 as an intermediate table IT2. In the first case, we would access all records of T2 through an non-clustering index, at a cost for data page access alone of 1.000.000R, clearly a folly strategy. In the second case, we need to materialize a table IT2 with 1 million records of 200 bytes each and sort the resulting records by A3. We defer consideration of disk sort, but it is reasonable to point out that a disk sort of these records would probably require two passes through disk pages, writing out the results of the first pass and then reading in these results for the second pass, an I/O cost of over 100,000S, with an elapsed time of 100,000/400 = 250 seconds. Clearly the nested loop strategy is superior in this case.

2.1.9.4.3. Estimating I/O cost for Hybrid Join Method

Consider again the query in figure 2.10. and the assumptions on indexes and filter factors of the last example. The I/O cost for the hybrid join with table T1 as the outer table can be estimated as follows. With 1000 records to look up in T1 having T1.A5 = 5, we see an index look up cost of 1R, for the index, and 1000L, for the data pages. We only need to extract A2 and A5 from T1 for each of these 1000 records and write them into an intermediate table IT1, of about 800 bytes big, and sort it by A2. For each record in the outer table IT1, we set T1.A2 = K, then look up index entries in the A3X index for T2.A3 = K, requiring 1R of index leaf I/O for each entry, at a cost of 1000R.

As we perform this look up, we create records of the form (T1.A2, T1.C5, RID) in the intermediate table IT2. This table will contain about 2000 records of 12 bytes each, about 24.000 bytes or buffer space for six disk pages, so once again we assume that no I/O is needed for creating IT2 and following sort by RID values. Finally, we pass through the records of IT2, using the sorted RID values in IT2 as a kind of RID list to retrieve the records from the inner table T2, and matching the A5 and A2 values of T1 with all attributes of T2 to generate the target list records. We are retrieving 2000 records from T2, likely all to be on separate pages, at an I/O cost of 2000L. Therefore, the total I/O cost for this method is 1000L, extracting records from T1, and 1000R , index entries from index A3X, and 2000L, extracting records from T2. The elapsed time is (1000 + 2000)/100 + 1000/40 = 55 seconds, an improvement over the nested loop join calculated some lines up here.

2.1.9.5. Multiple Table Joins

In most database systems, joins of three or more tables are performed by joining two tables at a time; the composite result of the first two joins is written to an intermediate table and then joined with the third table. The resulting composite may be joined with a fourth table, and so on, the order of joins is not determined by the SQL Select statement, but is left to the query optimizer. The proper choice is very important.

Consider a three table join of the form:

```
SELECT T1.A1, T1.A2, T2.A3, T2.A4, T2.A5,  T3.A6, T3.A7
FROM T1, T2, T3
WHERE T1.A1 = 20 AND T1.A2 = T2.C3
      AND T2.A4 = 40 AND T2.A5 = T3.A6
      AND  T3.A7 = 60
```

The execution plan available for such a join has different degrees of freedom. We can start by performing either of the joins T1 ⊕ T2 or T2 ⊕ T3. Assuming that we start with T1 ⊕ T2, we can use a nested loop join or hybrid join with either T1 or T2 in the outer loop , or a merge join. Once the intermediate table T4 := T1 ⊕ T2 has been created, we need to perform the join T4 ⊕ T2, once again using one of the join strategies. The query optimizer needs to consider all such plans to find out the most efficient one, and it is here that efficient algorithms for query optimization begin to become important. For joins involving multiple tables, query optimization can require a great deal of computational effort.

Note that in the plan just mentioned, joining T1 and T2 to create T4 then joining T4 to T3, if the query optimizer decides to perform the join T4 ⊕ T3 by a nested loop algorithm, the intermediate table T4 := T1 ⊕ T2 does not need to be physically materialized before starting the final join step. As each record of T4 is generated from T1 ⊕ T2, the next iteration of the nested loop join T4 ⊕ T3 can be immediately performed. The technique whereby successive record output from one step of an access plan can be fed as input into the next step of the plan is known as pipelining. Pipelining minimizes the physical disk space needed for materialization. Even more important, in cases where only small fractions of initial records are required, because for example the terminal user ceases scrolling through the list of qualified records, minimal materialization by pipelining often saves great deal of effort. However, pipelining is not always possible: if the query optimizer chooses a merge join to evaluate T4 ⊕ T3, where the T4 ⊕ T3 join attributes dictate a different sort order than the join before the initial sort of the next join step can be performed.

2.1.9.6. Transforming Nested Queries to Joins

It is possible to transform most nested queries into equivalent queries involving only table joins. This is an important technique for the query optimizer.

To illustrate the query transformation technique, let us start by considering an example of a Select statement using a *nested subquery* .

Consider the query:

```
SELECT * FROM T1
WHERE A1 = 5 AND A2  in (SELECT A3 FROM T2 WHERE A4 = 6)
```

We can think of the query as being executed in two steps. First, we evaluate the subquery, extracting a set of values for the A3 target list into an intermediate table IT1. Second the outer query, 'SELECT * FROM T1 WHERE A1 = 5 And A2 in IT1', is evaluated with regard that the values of A2 are within the list of values of IT1. The most efficient way to do this, given that there is no index created on IT1, is probably as a merge join: 'SELECT * FROM T1 WHERE A1 = 5', is extracted into an intermediate table IT2, the records are ordered by the T1.A1 values, and then the merge join process is performed between IT2 and IT1. Possible duplicate records must still be removed from the final list. This procedure reminds us much of the following query join form:

```
SELECT * FROM T1,T2
WHERE T1.A1 = 5 AND T2.A4 = 6 AND T1.A2= T2.A3
```

The subquery at the beginning and this join query give identical results and the equivalent join form allows the query optimizer to use other algorithms that are not obvious in the nested form. For example, it is now possible to perform a nested loop join with table T2 as the outer table.

The need for the DISTINCT keyword in transforming nested query into a join is due to the following observation. If a records r1 of T1 obeys the predicate of the join query, then r1.A1 = 5 and there must exist a record r2 in T2 so that r2.A4 = 6 and r1.A2 = r2.A3. But nothing is said about the attributes A3 and A4 forming a relational key of T2, so it is perfectly possible that there is a second record r3 in T2 that has the same values for A3 and A4 as r2. Then in the target list of the join query without a DISTINCT keyword, the record r1 would appear twice. This would clearly not happen in the original nested form of the query, since each single record of T1 is conceptually considered only once and qualified or not by the predicate A2 = 5 and A2 in IT1. Thus the DISTINCT keyword in the join form merely casts out duplicates that would not appear in the nested form.

The aim of a transformation such as the one described is that it reduces the number of different types of predicates the query optimizer needs to consider to get optimal efficacy. Once the nested query has been rewritten as an equivalent join query, it is reduced to a problem previously solved, and the query optimizer can use any of the join strategies we have introduced. To complete our analysis on query transformation for efficacy purposes, let us consider an other example. We consider the case of a *correlated subquery*.

Consider the query:

```
SELECT * FROM T1
WHERE A1 = 5  AND A2  in (SELECT A4 FROM T2 WHERE A5 = 6 AND A6 > T1.A3)
```

As this nested form contains a correlated subquery, it is not possible to evaluate the subquery until the outer records are fixed so that T1.A3 can be evaluated. From this consideration, it seems that the only valid approach is to start by looping on records of T1, then for each records in T1 find all records in T2 through an index on A4, where T2.A4 is equal to the outer T1.A2, and then resolve the predicate clause T2.A6 > T1.A3. Now, notice that the nested query above gives the same result as the following join query:

```
SELECT * FROM T1, T2
WHERE A1 = 5 AND T1.A2 = T2.A4 AND T2.A5 = 6 AND T2.A6 > T1.A3
```

It is much easier to picture the strategy of performing a merge join with the query in this form. Extract records from T1 where T1.A1 = 5 into an intermediate table IT1, and sort it by T1.A2 values. Then sort T2 in order by attribute A4 into an intermediate table IT2. Now merge join IT1 and IT2 on matching values for T1.A1 and T2.A4, casting out duplicates, and qualify records matched by verifying predicate T2.A6 > T1.A3. This strategy might be superior to the nested loop join that seemed most natural for the query in nested form.

Out of both examples we are able to define the following theorem.

The following two query forms give equivalent results:

```
SELECT T1.C1 FROM T1
WHERE [SET A OF PREDICATES ON T1 AND]
      T1.A2 IN (SELECT T2.A3 FROM T2 [WHERE SET B OF PREDICATES ON T2, T1])
```

is equivalent in result to

```
SELECT DISTINCT T1.C1 FROM T1, T2
WHERE T1.A1 = T2.A3 [AND SET A OF PREDICATES ON T1]
      [AND SET B OF PREDICATES ON T2, T1]
```

Note that SET A of predicates can be empty, as well as SET B. Note also, that if no predicate is in SET B from the nested query referring to table T1 in the outer query, the entire nested query is non correlated.

DBMSs perform this kind of transformation only under certain conditions, including the following.

- The subquery target list is a single attribute, guaranteed by a unique index to have unique values.
- The comparison operator connecting the outer query to the subquery is either IN or = ANY (with the same meaning).

Therefore all nested queries involving the not exist predicate are not transformed into join predicates. But most nested queries have equivalent join forms, ant it turns out that the query optimizer usually finds a more efficient execution plan if the query is posed in the join form. This is true even if the transformation into a join plan doesn't take place under DBMS rules of transformation and implies that the person writing queries should make some effort to create a join form rather than the equivalent nested form query if possible. Given that a nested form query is used, it is possible in nearby all DBMSs to tell from the output of the EXPLAIN command if transformation into join form has been performed by the query optimizer.

2.2. Data Update Operations

Three SQL statements, INSERT, DELETE and UPDATE, are used to modify data upon table data. The ***Insert*** statement is used to insert new record(s) into a table. Whereas the ***Delete*** statement is used to delete records from a table. The ***Update*** statement is used to change the values of some attributes. These three statements are often referred under the name of *update operations*, since they all serve to update table data. There is some risk of confusion here, because the Update statement is the specific name of one of these three operations, and we need to take care to differentiate the two.

Whenever update operations are applied, the integrity constraints specified on the database schema should not be violated. Therefore the update operations have the property of *atomicity*, that is that the operations are executed entirely or not at all leaving the database in a consistent state.

2.2.1. INSERT Operation

As for the Select statement, let us see the general syntax of the INSERT statement.

```
INSERT INTO tablename [column {, column}]
[VALUES (expression {, expression}] | [SUBSELECT]
```

The insert statement inserts records into a given table. One of both form must be used: either the *values form*, where a single records is inserted with given values, or the *Subselect form*, where all records that result from the Subselect are inserted.

The ability to use a Subselect to create input to an Insert statement adds a great deal of power. Only one table receives new records in an Insert statement, but the Subselect can be on any number of tables, as long as it produces the right number of attributes of the right type to serve as new input record.

The insert operation provides a list of attribute values for a new record r that is to be inserted into table T1. Inserts can violate any of the four types of constraints [Elmasri 1994]. *Domain constraints* can be violated if an attribute value is given that does not appear in the corresponding domain. *Key constraints* can be violated if a key value in the new record already exists in another record in table T1. *Entity integrity* can be violated if the primary key of the new record r is null. *Referential integrity* can be violated if the value of any foreign key in record r refers to a record that does not exist in the referenced table.

2.2.2. DELETE Operation

The syntax of the Delete statement is:

```
DELETE FROM tablename
[WHERE search_condition]
```

The delete statement removes record(s) from a table and makes them inaccessible. There are two types of delete operations: delete without condition and delete with condition. The *delete without any delete condition* deletes all records from a table, whereas the *delete with condition* deletes a set of qualified records from the table. In the second type we see that a delete operation includes a data access operation to locate the records to be removed. This makes the delete statement more expensive in terms of I/O costs than the select statement, because it needs a search for the records before it can remove them.

Delete operations can violate referential integrity, this happens when the record to delete is referenced by foreign keys. Three options are available, to keep atomicity, when a delete causes a violation. The first option is to *reject the deletion*. The second option is to *attempt to cascade* (or propagate) the deletion by deleting records that reference the record that is being deleted. A third option is to *modify the referencing* attribute values that cause the violation; each such value is either set to null or changed to reference another valid record. Note that, if a referencing attribute that causes a violation is part of the primary key, it cannot be set to null; otherwise, it would violate entity integrity.

2.2.3. UPDATE Operation

As for both other update operations we give the Update statement syntax.

```
UPDATE tablename
SET column = {expression | null} {, column = {expression | null}}
[WHERE search_condition]
```

The update operation changes information in existing records of a given table. It replaces the values of the specified attributes with the values of the specified expression for all records of the table that satisfy the search_condition.

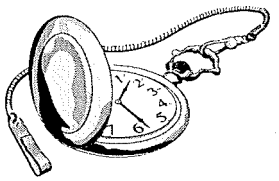
Modifying the values of an attribute that is neither a primary nor a foreign key usually causes no problems; the DBMS need only to check that the new value is of the correct data type and domain. Modifying a primary key value is similar to a delete operation followed by an insert operation, because we use the primary key to identify the records. Hence, the issues discussed under both Insert and Delete come into play. If a foreign key attribute is modified, the DBMS must make sure that the new value refers to an existing record in the referenced table.

Note, that only one table can be object of the Update statement. The limitation of the Update statement, compared with Insert statements, is that we can not compute values using references to other tables.

2.3. Data Macro Operations

Transactions, also known as *macro operations* are contain a sequence of data access and/or update operations. Their execution reflects also the property of atomicity, such as if the database had been in a consistent state before execution, it still is after execution. However it is possible that within the sequence of operations integrity constraints are violated. This mechanism allows us to create high level operations, insuring the respect of integrity constraints. The macro operations play also the role of integrity operations in case of incident and/or concurrency.

As macro operations are composed of various data operations, we might split them into their elements without great effect on our future reasoning. Within the scope of this paper we shall not examine in more details data macro operations.



Chapter 3. Data Access Structures

Computers have been getting faster and cheaper during the last decades. A relatively inexpensive computer today can execute program logic with CPU at a rate of 25 million instructions per second (25 MIPS). Of course different instructions take different execution delays, but this rate is a rough measure of the average speed for a "standard" mix instruction set.

The data structures of a database often influences the way the data and access structures are stored upon physical devices. There exists a wide range of direct access devices, like magnetic hard disks, drums and/or optical disks. For simplicity, we use the term disk to denote all those devices and assume that the whole database fits on a single disk.

Disk access speed, though an extremely important aspect of database system performance, has not kept pace with the enormous improvements in CPU execution speed. A disk is a rotating magnetic recording device, with several platters stacked one above the other moving at the speed of 60 rotations[23] per second (rps), and a disk arm that moves in and out like an old-style record player arm. The disk arm terminates in a set of read-write heads that sit on the surfaces of the various platters. As the arm moves through its range of positions, the heads all move together to address successive concentric cylinders of data, made up of circles or tracks on the stacked set of surfaces. A track is broken up into a sequence of blocks. In order to read or write data on disk, the arm must first move in or out to the appropriate cylinder position, then wait for the disk to rotate until the appropriate block is just about to pass under the read-write head. At this point the disk head reads the data from a sequence of blocks. A disk access cost usually includes three phases, based on its physical components:

- *Seek time*. The disk arm moves in or out to the right cylinder position.
- *Rotational latency*. As the disk rotates the read-write head has to wait until the right block of data passes by.
- *Transfer time.* The disk arm reads or writes the data on the right disk surface.

Because disk accesses require physical actions, the time needed to read in a random piece of data is important, about 1/40 of a second. This is usually divided into the three phases as follows.

Seek time:	0,016 seconds
Rotational time:	0,008 seconds
Transfer time:	0,001 seconds

[23] A new rotation speed of 90 rps is becoming the standard, but we assume 60 rps in what follows.

The seek time is highly variable, depending on the starting position of the disk arm and how far the arm has to move to the corresponding cylinder. If two successive reads from disk are physically close to one each other, the seek time might be very small. The average seek time of 0,016 sec is based on a model where successively accessed pieces of data occur with equal likelihood at any cylinder position between the two extreme disk arm positions. The average rotational latency time takes about half a disk rotation time at 60 rps. This average value assumes that the start sector can be anywhere on the track after the disk arm seek is completed.

Once the data has been brought into memory, it can be accessed by an instruction in 0,04µs by a machine with a speed of 25 MIPS. The disparity between time for memory access and disk access is enormous: we can perform about 650.000 instructions in the time it takes to read or write a disk page. *Clearly we want to avoid performing more disk reads than are absolutely necessary.*

As the disk represents the latent cost in the search for data it is helpful to understand how the database designer can model, tune or use data allocation parameters and data access structures to speed up data retrieval.

3.1. Physical Data Allocation Parameters

3.1.1. Page Oriented Transfer Mode

Considering the overhead time for each disk seek, the data read should be of a certain size. It takes very little time for each additional byte retrieved, because most of disk access time was spent by moving the arm on the right position. Once the arm is on the right place, the transfer rate is millions of bytes per second. As a result, we see that all disk accesses are ***"page-oriented"***, a page I/O is a long contiguous sequence of bytes: 2 Kb, is the standard page size for database systems running on UNIX, while 4 Kb is the standard on IBM mainframes.

3.1.2. Assumptions about I/Os

Recall that a random disk access on a normal disk takes an expected elapsed time of approximately 0,025 seconds (1/40 of a second). But that does not mean necessarily that N successive (14.000R) random page I/Os require a total elapsed time of N/40 (14.000R/40 = 350 seconds) seconds. It is quite possible that these N (14.000) pages could be spread out over separate disks (say 10) and the system could then enlist the service of all disk arms moving concurrently, so that each disk reads only uses a fraction of all pages (1400R pages per disk), in a fraction of the elapsed time (35 seconds). This approach of multiple disk arm acting simultaneously in an execution plan of a query is known as *I/O parallelism.*

The feature of I/O parallelism is nowadays offered by a number of database systems. Technologies such as stripping offer the faculty of *stripping* data pages across several disks. For example, page 1 on disk 1, page 2 on disk 2, ..., page 11 on disk 1, page 12 on disk 2, and so on, with the N^{th} page lying on disk ((N-1) MOD 10) + 1. When stripping the pages this way, the system reading successive pages from a table can make multiple I/O reads and thus keeping all involved disk arms busy most of the time. Since we can easily predict future tablespace page requests when performing a table scan, we merely need an architecture that supports stripping, translating logical page addresses into disk addresses on multiple disks and passing I/O requests to the appropriate device. This is pretty easy to implement, since allocated extents in such a stripping architecture must span multiple disks in a well-defined way.

Note that disk stripping does actually not reduce the number of disk accesses, we *require the same total number of random I/O*, even though the workload is splited upon separate disk devices. With parallelism we surely obtain better response times for query, they reducing wasted time and turnover caused by the frustration of waiting for the query responses. Despite, I/O parallelism still gains in importance in database systems, we generally assume that we are dealing with sequential I/Os. Where one I/O request must be completed before a second request is made during query execution.

Sequential I/Os has many tricks to offer, like the ***mutli-block access***, that will actually save a large amount of disk accesses during query execution. We will have a look at multi block access in chapter 4.

Next to the mutli-block access there exists another kind of access technique, known *as* ***the list prefetch***, in which the disk controller is given a list of pages, usually 32, that need to be read into memory buffers. However the pages are not necessarily in contiguous sequential order as with mutli-block access. With the technique of list prefetch, the disk arm is programmed in advance with the most efficient way to perform successive reads, so that the I/Os occur in a much more efficient way than they would with random I/O requests. The elapsed time for a list prefetch access can not be smaller than the elapsed time for a mutli-block access, as this represents the optimum for disk accesses. The actual speed, for a list prefetch, is determined by how far apart the pages are on disk, but as a rule of thumb we can assume that the list prefetch proceeds at 100 I/Os per second. For future I/O cost estimations we use the approximate time rates given in figure 3.1. Although these are rather rough figures, they usually give reasonable good cost estimations.

Random I/O	*Mutli-block access I/O*	*List Prefetch I/O*
40 pages/sec	400 pages/sec	100 pages/sec

figure 3.1.: Rules of Thumb for I/O Rates[24]

3.1.3. Page Buffering

As disk access is very slow compared to memory access. It is worth, after bringing in a disk page to memory buffers, to make every attempt to keep it there in the hope that it will be referenced again in the near future. Once the page is in the buffer we will save disk I/Os.

[24] [O'Neil 19994]

To support buffering, the system uses an approach known as ***lookaside***, which allows the system responding to a disk page read request to first try to hash to an entry for the page in the *lookaside table* [O'Neil 1994]. The technique of lookaside works as follows: The system reads disk pages through an interface which provides the disk page address, dp, and brings the data page into memory buffers. The dp values might be a logical succession of integers or a construction from the device number, cylinder, surface and starting sector position of the page on disk. Each page that is read in has its disk page address hashed, h(dp), into small entries of the hash lookaside table, which points to the buffer slot where the page is located (see figure 3.2.). From this point on, every time a new page is to be read we start by hashing the disk address and look in the hash lookaside table to see if the desired page is located in the buffer. If it is so then the system skips the disk access. The pages we want to keep in the buffer are the ones that are the most 'popular'. This can be accomplished with a method known as Least Recently Used, LRU, buffering. The idea is that database pages that have been read from disk into the buffer will remain there until a new read page requires the space and all buffer pages are occupied. At this point, the pages that have not been referenced for the longest time are dropped from buffer to free some space.

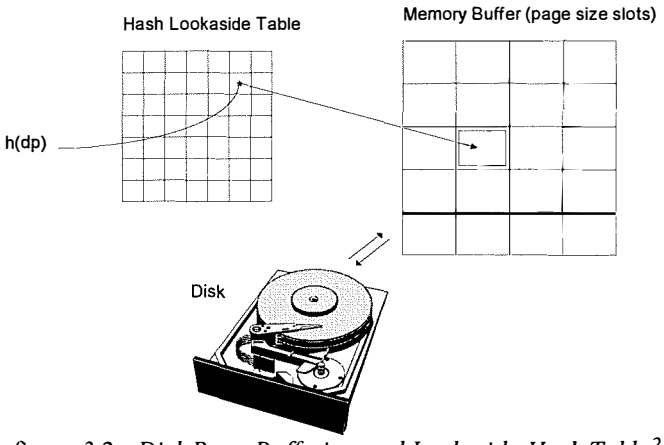


figure 3.2.: Disk Page Buffering and Lookaside Hash Table[25]

Just as we want to keep pages in memory to be read over and over by different data access operations, we do not want to write back pages to disk every time it is updated by some operation. If it is a popular page for update, containing a set of records with bank branch balances for example, we might be able to allow hundreds, or even thousands, of updates without writing back to disk the page in question. Instead we generally allow popular pages to remain in buffer until either they become less popular and drift out of buffer on their own because LRU needs their buffer place, or else we force them to be written back to disk after some period of time has passed.

Most systems allow you to define buffer sizes during system initiation, therefore the buffer size is a helpful tuning parameter.

[25] [O'Neil 1994]

As mentioned, the purpose of the buffers is to reduce the number of physical accesses to disk. The impact of the buffers on the number of physical page I/Os (physical accesses) depends on three major parameters[26].

- *Logical read and writes.* These are the pages that the system accesses via system read and write commands. Some of these pages will be found in the buffer, while others will initiate physical reads or/and writes.
- *Database system page replacement strategy.* These are the physical accesses to a disk, when a page must be brought into the buffer and there is no free page. You ensure that the page replacement occurs rarely, by increasing the number of asynchronous paging daemons which write committed pages to disk [Shasha 1992].
- *Operating system paging.* These are physical accesses to disk (swap disk) that occur when part of the buffer space lies outside random access memory. You should ensure that such swapping occurs rarely.

Assuming that swapping and page replacements occur rarely, the important question is *how many logical disk I/Os become physical disk I/Os when I access data*? We call this the *hit ratio*, defined by the following relation:

HIT RATIO = (Nal - Nap) / Nap

where

Nal: number of logical accesses
Nap: number of physical accesses

relation 2.5.: Hit ratio

The hit ratio is the number of logically accessed pages found in the buffer divided by the total number of logically accessed pages.

By varying the buffer size you can directly influence the number of pages that can be hold in the buffer, the hit ratio varies therefore in correspondence with the buffer size, hence the number of physical I/Os varies also.

According to [Shasha 1992], *increasing the buffer size is beneficiary to access performances, as long as the buffer can be hold entirely in central memory.*

The best strategy, in buffer tuning, is to increase the buffer size until the hit ratio flattens out, while making sure that page replacement and swapping are low. Some systems, like ORACLE, offer utilities that will help you to find out what the hit ratio would be when varying the buffer size.

Other systems, like DB2, give the possibility to dedicate a database buffer to some application X and a second one to other application Y. This feature is surely interesting in case where application X has higher response time requirements, than the application Y. And when both access different sets of data. However, if all applications have basically the same requirements, a rule of thumb is to use only a single memory buffer.

[26] [Shsha1992]

Note, that the buffer technique has dangers. As memory buffer is volatile, imagine the sudden loss of power or a system crash. Some of the pages on disk might be terribly out of date, because they were so popular that they have not been written out from buffer during the last thousand updates that took place. If all those updates existed only in memory, then it would seem they are now totally lost. How can we handle this problem, and be able to recover these lost updates, without going back to the approach of writing out every update as soon as it is committed?

The answer is that all systems use ***redo log entries***, each time an update occurs, the system writes a note to itself into a memory area known as the *redo log buffer*. The log entries contain information about updates to remind the system how to perform the update once again, or to reverse the updates if the operation involved needs to be aborted (atomicity of update operations). At appropriate times the log buffer is written out to disk, into a sequential file known as the ***redo log file***, that contains all the log entries created for some interval of time into the past. In this way, if memory is lost at some point, the recovery process will be able to use the sequential log file to recover updates of records that are out of date on disk. One reason that this log method is preferable to writing out each update, as it happens, is that it is more efficient, the system only needs to write the log buffer out to disk at infrequent intervals, it is usually able to batch together a large number of page updates and thus save disk I/Os.

3.1.4. Tablespaces, Segments and Extents

Although we will not the internal details of data resource allocation, we try to give an idea of aspects and considerations that can arise, based upon [ORACLE 7.0]. Most commercial database systems deal with data allocation in similar way, even though the details can be different.

Before creating a database, you might start by allocating a large number of operating system (OS) files on disk, with names like fname1, fname2, and so on. These are ordinary sequential files, such as those that you encounter when you edit text. Various OSs give you the ability to specify the size of the file in bytes and the disk device on which the file is allocated. Note that we consider disk storage as being contiguous when it consists of sectors on disk that are as close together as possible[27]. Keeping disk space contiguous minimizes seek time, and most systems try to allocate space to file in long contiguous chunks. At the same time, most OS files do not have the flexibility to span disk devices. Given these files, you might use a system command to create *tablespaces*. A tablespace is the basic allocation device of ORACLE database, out of which tables and indexes, as well as other elements requiring disk space, receive their allocations. A tablespace corresponds to one or more files and can span over one or more disks. Many ORACLE databases contain several tablespaces, including the SYSTEM tablespace, which is automatically built when the database is created. The system tablespace contains the data dictionary and may also be used to provide disk space for user defined indexes, as well as other elements. It is up to the database administrator to decide whether to create multiple tablespaces or not. The advantage of multiple ones on large systems is to have a better control over which devices are used for what purposes, and the ability to take some disk space off line without bringing down the whole database.

When a table or and index is created by the DBA, you have the possibility to name the tablespace, otherwise a default tablespace is used. When a table is created, its tablespace allocation is identified with a data segment, in case of an index, it is identified with an index segment. When a data or index segment is first created, it is allocated to an initial *extend.* Each time a data segment comes close to running out of space, it is given additional allocation of space, known as a *next extent*. Figure 3.3. gives a graphical representation of the logical structures just explained above.

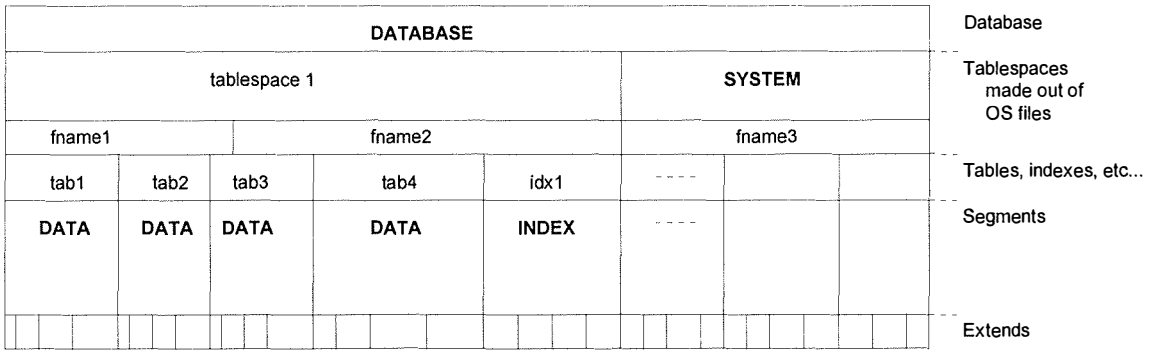


figure 3.3.: Database Storage Structure[28]

An extend must consist of contiguous disk space, and is therefore usually within a single file making up a tablespace. The creation of a tablespace allows you to specify default parameters, governing how space allocation of disk extents is to be handled by the database system. Here below is a short description of these parameters.

---

[27] Successive sectors on successive surfaces of a cylinder within a succession of adjoining cylinders.

[28] [ORACLE 7.0]

- *initial n*. The integer n specifies the size in bytes of the initial extent to be assigned.
- *next n.* The integer n specifies the size in bytes of the next extent numbered 1; the size of subsequent next extents may increase (but not decrease) if a positive *pct-increase* value is specified.
- *maxextents n.* Specifies the maximum number of extents, including the initial extend, that can ever be allocated.
- *minextent n*. Specifies the number of extents to be allocated initially when the segment is created.
- *pctincrease n.* Specifies the percentage by which each successive next extent grows over the previous one. IA value of 0 means that there is no increase, whereas a value of 50 causes successive next extents to grow by a factor of one and a half over the preceding one.

In case many queries tend to scan (large) portions of a table, it seems reasonable to specify sector-sized or larger-sized extents for good read performances. Write performance can also be improved by using extents. For example, redo logs and history files will benefit significantly from the use of extents. If *access to files is completely random, then small extents are better*, because small extents give better space utilization, due to contiguous space allocation.

3.1.5. Pctfree, Pctused and Fill Rate

The CREAT table command allows us to specify two more data allocation parameters, the *pctfree* and *pctused* parameters [ORACLE 7.0]. The pctfree and pctused clauses together determine how much space within each page can be used for new record inserts.

The *pctfree* value must be an integer from 0 to 99, where a value of 0 means that all space can be used for new inserts. A default value of 10, means that new inserts in the page stop when 90% of the page is full. The remaining 10% of space are used for future record size expansions.

The *pctused* value, is an integer from 1 to 99, specifies where new inserts to a page will start again if the amount of space used by stored records falls below a certain percentage to the total. The default value is 40.

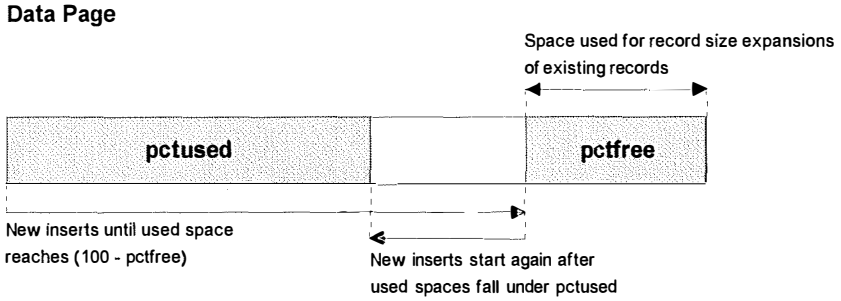


figure 3.4.: Illustration of pctfree and pctused[29]

[29] [ORACLE 7.0]

Note that the sum of pctfree and pctused values can not exceed 100. Together they determine a range in which the behavior with respect to inserts on the disk page remains stable, depending on the last percentage value encountered. Figure 3.4. illustrates the concept of pctfree and pctused.

In some other systems, like INGRES, there are no parameters such as pctfree and pctused. They have a parameter called the *fill rate* or *fill factor*. It is comparable to the pctfree parameter in ORACLE, except that where pctfree gives the percentage of space that should remain unused during initial creation of the table, the fill rate defines the *percentage of space that should be filled.* For the rest of the paper we will refer to the fill rate of data pages.

Using the fill rate, we are able to define the number of records that can be hold in a page. It is a direct function of the page size, the record size and the fill rate. For purpose of simplicity we consider that the record length is fix[30].

$$Nrp = \left\lfloor \frac{(Ps - Hs) * fr}{Rs} \right\rfloor$$

where

Nrp: Number of records per page
Ps: Page size
Hs: Page header size
fr: fill rate
Rs: Record size

relation 2.6.: Number of Records per Page

Assumed that a page only contains data objects of the same type. System like ORACLE, however allow multiple data object types per page. Relation 2.7. generalize relation 2.6. according to the different object types and the balanced average record size for all objects. Relation 2.7. brings into play the proportion of place used by the set of different data objects.

$$Nrp = \left\lfloor \frac{Na * Rs}{\sum_{i} Na_i * Rs} * \frac{(Ps - Hs) * fr}{Rs} \right\rfloor$$

where

$i \in$ {data object types present in the page}
Nrp: Number of records per page
Na: Number of records of a certain type
Ps: Page size
Hs: Page header size
fr: fill rate
Rs: Balanced Average Record size

relation 2.7.: Number of Records per Page for Multiple data objects

Note that if all data objects within the page are of the same kind then relation 2.7. simplifies into relation 2.6..

[30] If the lenght is variable, we take the average length of the records.

3.1.6. Data Pages and Record Pointers

Once a table has been created and the initial extend of disk storage allocated, we are ready to load or insert data into the table. In a typical data architecture for record placement, records are simply inserted one after another on the first page of the first extent. After the first page is all used up the next collocated page is used. When the initial allocated set of pages gets full, the DBMS allocates a new extent and continues its process until the maximum number of extend is reached.

Figure 3.5. gives a graphical representation of a typical data storage page layout for N records. The header info section might contain fields to show what type of disk page it is, the number of the page within the database file, and so on. Each record is a contiguous sequence of bytes, starting at a specific byte offset within the page[31]. Entries in the record directory point the record within a page and give the page offset to where each record begins.

We assume that newly inserted records are placed right to left in the page, and directory entries from left to right, leaving free space for future inserts in the space between. This implies that if a record is deleted from disk page and its space is reclaimed, then the remaining records are shifted to the right and the directory entries to the left, so that all free space remains in between. However, systems like ORACLE can defer shifts, reorganizations of this kind.

figure 3.5.: Record Layout on a Disk Page

A record in the database table can be uniquely identified by its *record pointer* (RID), this specifies the data page on which the record appears and the slot number of the record within the page. This is how indexes will use references to point the records that correspond to a given index attribute value. It turns out that we gain flexibility by pointing to a record using *logical slot numbers* within pages, because of the information hiding that takes place. If RID pointers use record byte offsets the system needs to change the index RID pointers in case where records are moved during reorganization. Therefore using logical slot numbers in record pointers, implies that external index references to records remain unchanged during reorganization.

Note that the concept of record pointers is not part of any standard, however the general form explained up here is very common [O'Neil 1994].

[31] Note that because of *varchar(n)* datatypes, the records might have different lengths.

In DB2 and INGRES[32], the record pointers encode the table page number and slot number into a *4 byte* integer. Successive pages allocated to a table are assigned a page number, P, ranging from 0 to 2^{23}-1, requiring 23 bit positions, a maximum of 8.388.608 pages. We allow at most 512 records per page, so that a slot number, S, ranging from 0 to 511 can fit on 9 bits. A record pointer is computed from the page number, P, and the slot number, S, for the particular record using the following rule:
RID = 512 * P + S

For example, the RID value for a record with slot number S = 4 on page P = 2 a RID value equal to 2 * 512 + 4 = 1028. The slot number within a page is not permitted to exceed the value 511, so the relation for a RID is always unique for records on different pages. The slot number fits in 9 bits and the page number in 23, so the RID always fits in 32 bits, a 4 byte unsigned integer.

In ORACLE[33], a record pointer[34] specifies the page (block) number on which the record falls, the slot number within the page, and the number of the file within which the page exists. Multiple blocks can have the same page number within different files making up a tablespace. A record pointer requires *6 bytes* of storage in a unique index. The record pointer can be displayed as a string of three hexadecimal numbers, like: BBBB.SSSS.FFFF. Where BBBBBB represents the page number within the file, SSSS the slot number, and FFFF the file number.

Until here we assumed that a record can not extend over several pages, but we know that different systems use different rules about whether records can extend over pages. In DB2, the maximum record size is limited to the largest page possible, and each record lies entirely in a single page. Pages and buffer sizes of 4Kb and 32Kb are possible, although the 4 Kb size is much more common. INGRES allows a record size up to 2.000 bytes and each record lies on a single 2Kb page. However, ORACLE allows records to split between pages. If a record on a page grows to a point where no free space remains, then the record is splited. Its record pointer remains the same and it leaves a record fragment in the original slot position on its original page, but a pointer at the end of that fragment points to the RID of the continuation fragment. The continuation part is placed, just as a record is placed, on a new page with a slot position and a record pointer.

---

[32] According to [O'Neil 1994]

[33] According to [O'Neil 1994]

[34] A record pointer is called ROWID in ORACLE database system.

Clearly, we would like to avoid fragmentation of records whenever possible. In ORACLE we minimize fragmentation by leaving extra free space on each page with the pctfree parameter to handle most record enlargements. However, since ORACLE permits records that cannot fit on a single page, extra free space on a page is not a general solution. When in DB2 and INGRES a record grows to a point that it can not fit anymore in its original slot together with neighboring records on the same page, it is moved entirely to a new page. A certain kind of fragmentation exists in at this point. A RID forwarding pointer must be left on the original page, because we don't want to change all index entry RID pointers to the new record position. Fragmentation arising from forwarding pointers often results in performance degradation and database reorganization.

3.1.7. Disk Contention

Let us first explain the concept of disk contention. Disk contention occurs when multiple processes try to access the same disk space simultaneously. Most disks do have limits on both the number of accesses and the amount of data they are able to transfer per second. When these limits are reached, processes may have to wait to access the disk space.

To reduce the activity on an overloaded disk, we can move one or more of its heavily accessed files to a less active disk. This principal of distributing I/O over disks can be applied until all disks have roughly the same number of I/Os.

Within the scope of this paper we shall not explain in more details the concepts of distributing I/O. We will nevertheless give some guidelines for distributing I/O among several disks [ORACLE 7.0].

- ***Separate data files and redo log files on different disks***. The DBMS constantly accesses data files and redo log files. If it happens that these files are located on the same disk, there is a potential for disk contention.

  It is helpful to place each data file on a separate disk, so multiple processes can access concurrently different files without disk contention.

  Note that the redo log files are sequentially written each time a transaction is committed. The sequential writing can take place much more faster if the redo log files are located on separate disks with no concurrent activity. According to [ORACLE 7.0] mirroring the redo log files, or maintaining multiple copies of the files does not considerably slow down the writes, as they are done in parallel to each disk and that the system waits until the parallel write is completed. However, dedicating separate disks and mirroring the redo log files considerably increased the security. The same way, dedicating separate disks to data and redo log files ensures that both files cannot be lost in a single disk failure.

- ***Stripping table data on different disks***. Stripping is the practice of dividing a large table into small data sets and storing them on different disks. This permits multiple processes to access different portions of the table concurrently without disk contention. Stripping is particularly helpful in optimizing random access to tables with many records.

- ***Separate tables and indexes on different disks***. When knowing which of the database structures are often accessed, it is helpful to place these database structures on different files on different disks. This separation distributes the I/O to the table and the index across separate disks.
- ***Reduce disk I/O not related to the DBMS***. If possible, the database administrator should eliminate disk I/O not related to DBMSs on disks that contain the database. This action is helpful in optimizing access to redo log files.

3.2. Physical Data Access Structures

When an SQL query, like those that we have seen in chapter 2, is submitted to a database system, a software module known as the optimizer analyses the non-procedural prescription of the query to determine an efficient step-by-step method to retrieve the qualified records. Throughout this section we give the foundation of index tuning, by learning how indexes are structured and how query effectiveness depends on the existing indexes.

3.2.1. The Concept of Indexing

An ***index*** is a data access structure whose purpose it is to improve the efficiency of data retrieval by using a *keyed access* retrieval method. The indexes we are going to talk about have some familiarity with memory-resident structures that support look ups, such as the binary tree, 2-3 tree, and hash tables. The difference, however, is that our indexes reside on disk and are only made memory resident when they are accessed.

An index consists of a sequence of index entries that are stored on disk, one index entry for each indexed attribute value. The indexed attribute(s) value(s) and a record pointer(s) (RID) compose the index record. The index entries are placed on disk, in sorted order by index key values (although hashed access is also possible), and are used by the system to speed up certain Select statements. Standard look ups using indexes, locate a set of index entries for a given key value or range of key values. And follows the RID pointers to its associated data records. The fact that the index normally resides on disk has an important effect on index structures as we will see.

You can best picture out an index to a table by analogy to a card catalog in a library, indexing the books on the shelves in various categories. One set of cards in the catalog might be placed in alphabetical order by subject name (several subjects are possible) another set of cards might be placed in alphabetical order by author name. Each card in the catalog contains a call number to locate the book indexed, so that once we find a catalog entry for a book with the title "Database Tuning: A Principal Approach", we can immediately locate the book on the shelves. Now, returning to the discussion about indexes, let us try to figure out what might happen when the system receives the following query.

```
SELECT *
FROM T1
WHERE A1 = 'Namur' AND A2  > 12 AND A2 < 14;
```

The query optimizer has now to decide how to access the requested records from table T1. One alternative that always exists is to perform a *table scan*, in which we successively accesses all records and discards the ones that do not satisfy predicate A1 = 'Namur' AND A2 > 12 AND A2 < 14. Sequential examination of the records is rather quick when we have a small sized table, or when selectivity is poor. This will be examined later on, when we describe the problem of index indecision.

However, consider that we have a large sized table T1, a good filter factor of predicate A1 = 'Namur' and that there is an index A1X defined upon attribute A2. In this case, we probably decide first to locate records with predicate A1 = 'Namur', using the index A1X. We can presume that the total number of records accessed is greatly reduced while limiting the query to the records having A1 = 'Namur'. However, we still have to check the retrieved records for the remaining predicate A2 > 12 AND A2 < 14. All in one we can consider that the index reduced our search for the qualified records.

3.2.2. B-Tree Index

Almost all commercial products support the index structure known as B-Tree. It is the only index structure available in DB2, and it was the only one provided by ORACLE until release 7, when a facility known as hash cluster was added. INGRES, on the other hand, has offered a wide set of different tree index structures for some time. It should be said that B-Tree provides a good deal of flexibility for different types of indexed access, and systems like DB2 implemented special features, such as mutli-block access I/O, that make the B-Tree competitive in performance for many types of queries.

A B-Tree is a multilevel keyed index structure, with a root page at the top and the leaf pages at the bottom, like in figure 3.6.. We refer to all nodes above the *leaf level*, including the root, as *index nodes*. Index nodes below the root are sometimes called *internal nodes* of the tree. The root node is also known as level 1 of the B-Tree, and successively lower levels are given successively larger numbers, with the leaf nodes at the highest level (level 3 in figure 3.6.). The total number of levels is called the ***depth*** of the B-Tree. Using this structure we can find our way down to any leaf level entry by first reading in the root page, then finding our way to successive internal pages, which ultimately directs us to the appropriate leaf page. We then examine the leaf page to find the leaf level entry we want, assuming that it exists. In figure 3.6., this means that we require a maximum of three I/Os to access the desired index entry, much fewer than a binary search as we will see. The leaf nodes are usually linked together to provide ordered access, on the indexed attribute, to the records.

Note that a binary search requires $\lceil \text{Log}_2\ n \rceil$ to access the requested data, where n is the number of index nodes at leaf level. Assume that we have 1.000.000 entries at the leaf level, each entry has a size of 8 bytes (4 bytes for the RID and 4 bytes for the key value[35]). Ignoring pages overhead, the number of entries that can fit on a page is at most $\lfloor 2.048/8 \rfloor = 256$. So we get a total of $\lceil 1.000.000/256 \rceil = 3.907$ leaf page. The cost for the binary search is $\lceil \text{Log}_2\ 3.907 \rceil = 12$ accesses, whereas the number of accesses throughout the B-Tree is three.

[35] This size is appropriate for an integer key value, however it could be much more for a character string index or for a composed index.

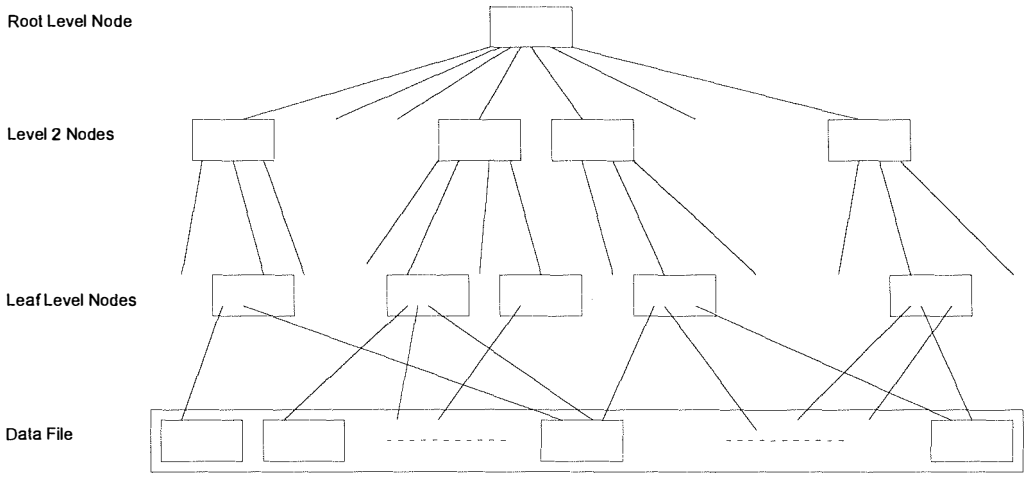


figure 3.6.: Illustration of a Three level B-tree structure

The relative I/O efficiency of a B-Tree over a binary search results from the fact that the B-Tree is structured in a way that gets out the most of every page access. Each index entries references a lower-level node page. For our example some 256 low-level pages can be referenced, whereas during binary search only two nodes are referenced. If we look at the tree of page accesses resulting from a binary search, we notice that the B-Tree is bushy, whereas the binary search tree is sparse. The B-Tree is flatter as a result, because we can reach as many as 256^3 leaf pages in a three level tree. The B-Tree is said to have *fanout* of 256, compared to the fanout of 2 for binary search. The leaf pages have their own fanout, with 256 entry RID pointer values pointing down to index records. The records of the table can be pictured as another level lying below the leaf level of a B-Tree.

If we assume a B-Tree with a fanout of f, we can reference N entries at leaf level in $\lceil \mathrm{Log}_f N \rceil$ probes. Thus with the fanout of 256 we have been assuming, we can access 1 million leaf level entries in $\lceil \mathrm{Log}_{256}\ 1.000.000 \rceil = 3$ probes compared to the 12 due to binary search. It is likely that frequently used nodes will remain in memory buffer, as a rule of thumb we can say that the *root node always stays in memory buffers*, this reduces the number of probes to two. It can happen that even the second level, with its 16 nodes, remains in memory, after all the proportion of space used is quiet small compared to the 3.907 nodes for the leaf level, this would reduce the number of accesses to one. But problems occur when a lot of indexes come into play, than it is possible that space usage could increase a lot.

A number of other access structures, such as hashing, also offer performance advantages for certain SQL queries. However, the B-Tree is the most commonly used index type in database today. What we call B-Tree in this paper is precisely known as B^+-tree, and represents a more recent variation on the original published B-Tree structure. The difference between both is, that in B-Tree, every value of the indexed attribute appears once at some level in the tree, along with the RID pointer [Elmasri 1994]. On the other hand, in B^+-tree, record pointers are stored only at the *leaf nodes*, hence the structure of the leaf nodes is different to the one of *internal nodes*. The leaf nodes, in case of a key attribute, have an entry for every value of the indexed attribute. For a *nonkey* or a *secondary key* attribute, the RID pointer points to a page containing the record pointers (RID), thus creating an extra level of indirection.

3.2.2.1. B-tree Definition

This section gives a formal definition of the B-Tree structure [Date 1990]:

The ***internal B-Tree page structure*** of order p is defined as:

1. Each internal page is structured like:
   $<P_1, K_1, P_2, K_2, ..., P_{fo-1}, K_{fo-1}, P_{fo},>$
   where
   P_i is a pointer to an internal index page
   K_i is a indexed attribute value
   fo is the possible number of entries on an internal page.
2. Within each internal node
   $K_1 < K_2 < ... < K_{fo-1}$ or $K_1 > K_2 > ... > K_{fo-1}$
   depending on the order defined during index creation.
3. For all search field values X in the sub-tree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < fo$, $X \leq K_i$ for $i = 1$, $K_{i-1} < X$ for $i = fo$. [36]
4. Each internal page has at most p index page pointers.
5. Each internal page , except the root, has at least $\lceil fo/2 \rceil$ index page pointers. The root page has at least two pointers if it is an internal page.
6. An internal page with fo pointers, has fo-1 search field values.

The ***B-Tree Leaf Page Structure*** of order p, is defined as:

1. Each leaf page has the following structure
   $< <K_1, RID_1>, <K_2, RID_2>, ..., <K_{fo-1}, RID_{fo-1}>, P_{next}>$
   where
   RID_i is a record pointer
   P_{next} points to the next leaf page of the B-Tree
   fo is the possible number of entries on an leaf page
2. Within each internal node
   $K_1 < K_2 < ... < K_{fo-1}$ or $K_1 > K_2 > ... > K_{fo-1}$
   depending on the order defined during index creation.
3. Each RID_i is a record pointer that points to the record whose search field value is Ki or to a file page containing the record in case of where the indexed attribute is a key. Otherwise, when the index attribute is not a key then the pointer references a page or RID_i pointers.
4. Each leaf page has at least $\lfloor fo/2 \rfloor$ values.
5. all leaf nodes are at the same level.

Figure 3.7. gives a graphical representation of the B-Tree structure.

[36] The definition follows the one give by Knuth. One can define the B-tree differently by exchanging the < and ≤ symbols ($K_{i-1} \leq X < K_i$, $X < K_i$, $K_{i-1} \leq X$), but the principal remains the same.

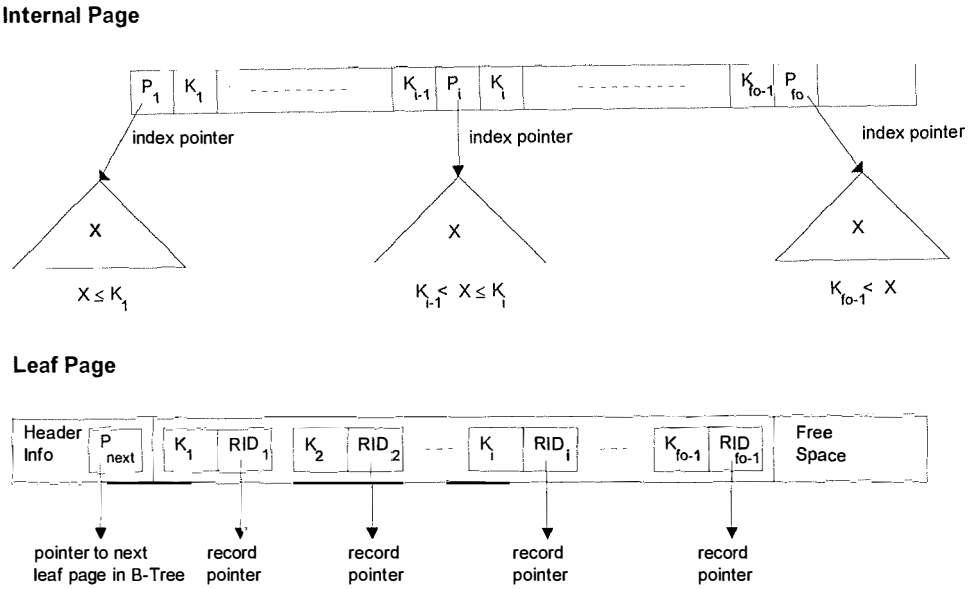


figure 3.7.: The pages of a B⁺-tree of order p: Internal node and leaf node representation[37]

3.2.2.2. Fanout and Depth of the B-Tree

The depth of a B-Tree bears close relationship to the number of disk I/Os used to reach the leaf-level page, where record pointers are kept. It is common to estimate the fanout at each level, where the fanout, fo, represents the expected number of records that can appear in each node. Assuming that there are fo entries for each index page, the number of possible entries at the second level is equal to fo^2, the possible number of entries for the third level is equal to fo^3, and so on. For a tree of depth d, the number of possible leaf page entries is fo^d, when we assume that all index pages can hold up to fo entries. Putting this a different way, if we want to build a B-Tree with entries for Nr attribute values, we need to have an index of d levels:

$$d = \lceil Log_{fo}(Nr) \rceil$$

with
fo: Fanout of index node pages
Nr: Number of values to be indexed

relation 3.1.: Depth of a B-Tree

The depth of the tree bears a logarithmic relationship to the number of values to be indexed. The depth d is then taken to be the number of I/Os, that must be performed to locate an attribute value entry at leaf page level. However, it turns out that both of these statements are misleading, because of buffering. It is common to consider that an active B-Tree of depth three has the root page buffered, so the effective number of I/Os to locate an entry is two, rather than three.

[37] [O'Neill 1994]

Note, that when creating an index, it is usually more efficient to first load the table with the initial set of records and then to create the index. This is advantageous because the creation process first extracts the index entries, then sorts them by key values, and finally loads them into the leaf pages. This process is extremely efficient compared to the dynamic reorganization of the B-Trees structure. The nodes of the B-Tree are loaded in a left to right fashion, so that successive inserts normally occur on the same leaf page, held constantly in the buffer. When the leaf page node splits, the successive leaf page is allocated from the next disk page of the allocated extent. Node splits at every level occur in a controlled way and allow us to leave just the right amount of free space on each page. On the other hand, as records are inserted after the initial index creation, this normally results in B-Tree entries that are inserted to random leaf-level nodes, requiring much more I/Os (because the leaf page affected is often not in memory buffer) and random page splits.

3.2.2.3. Index Page Layout and Free Space

An index page has a relatively simple structure. Figure 3.7. gives a graphical representation of a possible leaf page layout, with unique key values. The layout for an internal page is similar, except that record pointers are replaced by page pointers and that there is no next page pointer (P_{next}).

A ***B-Tree leaf page*** can have up to fo_{leaf} index records of structure $<K_i, RID_i>$ which we assume to fit on a single page. To determine the number of records a leaf page can hold, we use the following relation 3.2.:

Let us illustrate this throughout an example. The fanout is sometimes referenced as the order of the tree. Consider an indexed attribute, As, of 9 bytes long, a page size Ps, equal to 2Kb (2.048 bytes), a record pointer, Rps, equal to 6 bytes and a page pointer, Pps, equal to 6 bytes. Consider also that the page header, Hs, is 24 bytes long and that the fill rate, fr, is equal to 70%, Relation 3.2. defines the fanout for a leaf page.

$$fo_{leaf} * (Rps + As) \leq (Ps - Hs) * fr \Leftrightarrow fo_{leaf} = \left\lfloor \frac{(Ps - Hs) * fr}{Rps + As} \right\rfloor$$

where

fo_{leaf}:	Number of records that can be hold in a leaf page
Ps:	Page size
Hs:	Header size, we assume that the header contains the pointer to the next leaf page
fr:	Fill rate
As:	Indexed attribute size
Rps:	Record pointer size

relation 3.2.: Fanout of a Leaf Index Page

Similarly an ***internal B-Tree page*** can have up to fo page pointers and fo-1 search field values, which also have to fit in a single page. To determine the number of records an internal page can hold, we use the relation 3.3.:

$$(fo_{inter} - 1) * As + fo * Pps \leq (Ps - Hs) * fr \Leftrightarrow fo_{inter} = \left\lfloor \frac{((Ps - Hs) * fr) + As}{Pps + As} \right\rfloor$$

where

fo_{leaf}:	Number of records that can be hold in an internal page
Ps:	Page size
Hs:	Header size
Pps:	Page pointers size
fr:	Fill rate
As:	Indexed attribute size

relation 3.3.: Fanout of an Internal Index Page

We see that both relations are different, however, we will assume without no great harm that an internal page contains as much pointers as search fields, and that the size of a record pointer is the same than the page pointer size. Relation 3.4. defines a simplified relation for estimating the fanout.

$$fo = \left\lfloor \frac{(Ps - Hs) * fr}{Pps + As} \right\rfloor$$

with

Ps:	Page size
Hs:	Header size
Pps:	Page pointers size
fr:	Fill rate
As:	Indexed attribute size

relation 3.4.: Fanout of an Index Node Page

For example, let us see how we can compute the number of index records an index page can hold (internal and leaf pages). Consider an index entry size of 8 bytes and a node page size of 2048 bytes. We allow 48 bytes for the header and assume that the average nodes below the root level is only 70% full. This implies 1400 bytes of entries per page, and 1400/8 = 175 index entries per index page.

Assume that we have 1.000.000 entries at the leaf level, this means we will require at least $\lceil 1.000.000/175 \rceil = 5.715$ leaf node pages. With 5.715 entries at the next-higher directory level, we will have $\lceil 5.715/175 \rceil = 33$ pages on that level. Thus we have 33 entries at the root level, and therefore have a B-Tree of depth three. In general, we encourage rather rough calculations in sizing of this kind; a Header size of 48 bytes is probably incorrect for any particular product, but this gives us a round number for our calculations, and does not harm any of the principals.

3.2.2.4. Duplicate Key Values in an Index

Until now, we assumed uniqueness of the index attribute values, but to be accurate it often happens that we define an index on a non-unique attribute. The problem is than to determine how the system organizes its index leaf pages. We already mentioned that an extra level of indirection is needed to handle the multiple pointers.

In case the same key value is repeated for a large number of records, it is possible to list the key value only once with a pointer to a list of RID values. The entry in a leaf page is of structure <K_i, P_i > where the pointer Pi points to a block of record pointers (RIDs). Each RID itself points to one record with key value K_i. If some value K_i occurs in too many records, so that their pointers cannot fit in a single page, a linked list of blocks is used. This technique is illustrated in figure 3.8..

As the key value is often a relatively lengthy character string and the RID is usually quite short (4 to 6 bytes), this represents a large space saving. It is obvious that access performances are better, than in case where each occurrence entails an entry at leaf level of the B-Tree, because more entries can fit in a leaf page. However retrieval via the index requires at least an additional disk access, because of the extra level of indirection in the index.

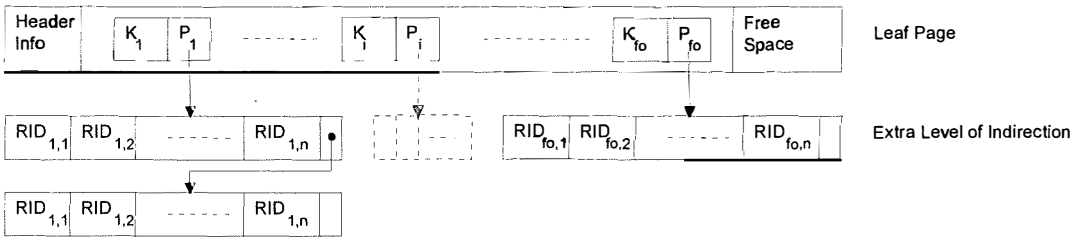


figure 3.8.: Leaf page layout with non-unique key values[38]

In most database systems the number of RID values in a block has an upper limit of 254, but clearly it is possible to create a list of blocks. Note that each additional block entails an disk I/O during access. The calculation of the fanout remains the same as in the section 3.2.2.3..

3.2.2.5. Dynamic Changes in the B-Tree

Let us consider the problem of insert, deletes and indexed value updates. Note that we will not explain the concept of the indexed value updates, we can consider these updates as a deletion followed by an insert.

For ***inserts***, note that a normal *sorted list* of entries on disk can always be reconstructed when a new entry is added. Only by moving all successive entries one position to the right, so to create space for the new insert, implying *I/Os for about half the pages*. Note that when there are frequent inserts such a technique is much to inefficient for large indexes.

[38] [O'Neill 1994]

We will explain, the means by which a B-Tree is kept ordered and balanced as new entries are added, using an example illustrated in figure 3.8. [O'Neil 1994]. Let us assume that free space is left in the pages so that inserts are often possible to pages at any level without new disk space being required. When a new entry is to be added in the index, we follow the index structure down to the leaf page as *if we were simply looking it up* (search like for a query), so that after the insert, the index structure channels us to the leaf level for the new added record. But occasionally the leaf *page is too full* to simply accept the new entry. In this case, for additional space the leaf page is *split* into two pages (the entries are kept in order, lower key values to the left split page and higher ones to the right). This means that the higher level index page must be modified so that a new separator exists, along with a new pointer to the new page (the other page has simply been reused). Occasionally, the modification of adding a new separator and pointer to the next higher level of index will exceed the space available on that index page. In that case the index page is split, in the same way than the leaf level, with possible resulting changes at the next higher level. Eventually an additional entry may be placed at root level. If the root splits, then a new root page is created at a higher level, having as its children the splited pages resulting from the former root.

Now take a look at figure 3.8., we assume that the B-Tree leaf page accepts at maximum up to three entries[39], the leaf level is of order $p_{leaf} = 2$. We also assume that internal pages have room for a maximum of two page pointers, so $fo = 2$.

The B-Tree starts out empty. As the first two entries (5 and 8) are inserted into the B-Tree, we have a simple structure, a leaf page that is also the root. No higher level index entries are needed, since all entries fit on that single page. As soon as more than one level is created, the tree is divided into its set of internal pages and leaf pages. Notice that *every value must exist at leaf level*, because all data pointers are at leaf level. However, only some values exist in internal pages to guide the search. Notice also that every value appearing in an internal page, appears also as the *rightmost value* in the sub-tree pointed.

When a *leaf page is full* and a new entry is inserted there, the page *overflows* and must be split. The first $j = \lceil (fo + 1) / 2 \rceil$ entries in the original page are kept there, and the remaining entries are moved to a new leaf page. The j^{th} search value is replicated in the parent internal page, and an extra pointer to the new page is created in the parent. They must be inserted in the parent page in their ordered sequence. If the parent internal page is full, the new value will cause it to overflow also, so it must be split. The entries in the internal page up to P_j, ,the j^{th} pointer after inserting the new value and pointer, where $j = \lfloor (fo + 1) / 2 \rfloor$, are kept, while the j^{th} search value is moved to the parent, not replicated. A new internal node will hold the entries from P_{j+1} to the end of the entries in the page. This splitting can propagate all way up to create a new root page and hence a new level for the B-Tree.

[39] This is much smaller than is realistic for a disk page, but keeping it small simplifies example.

Note in particular that the only way in which the depth of a B-Tree can increase is when the root page splits. Immediately after a root split, all leaf nodes increase their depth by one, and it is clear that there is no way in which two leaf pages can ever become to be of different depth down from the root in a growing B-Tree. That is why the B-Tree remains *totally balanced.*

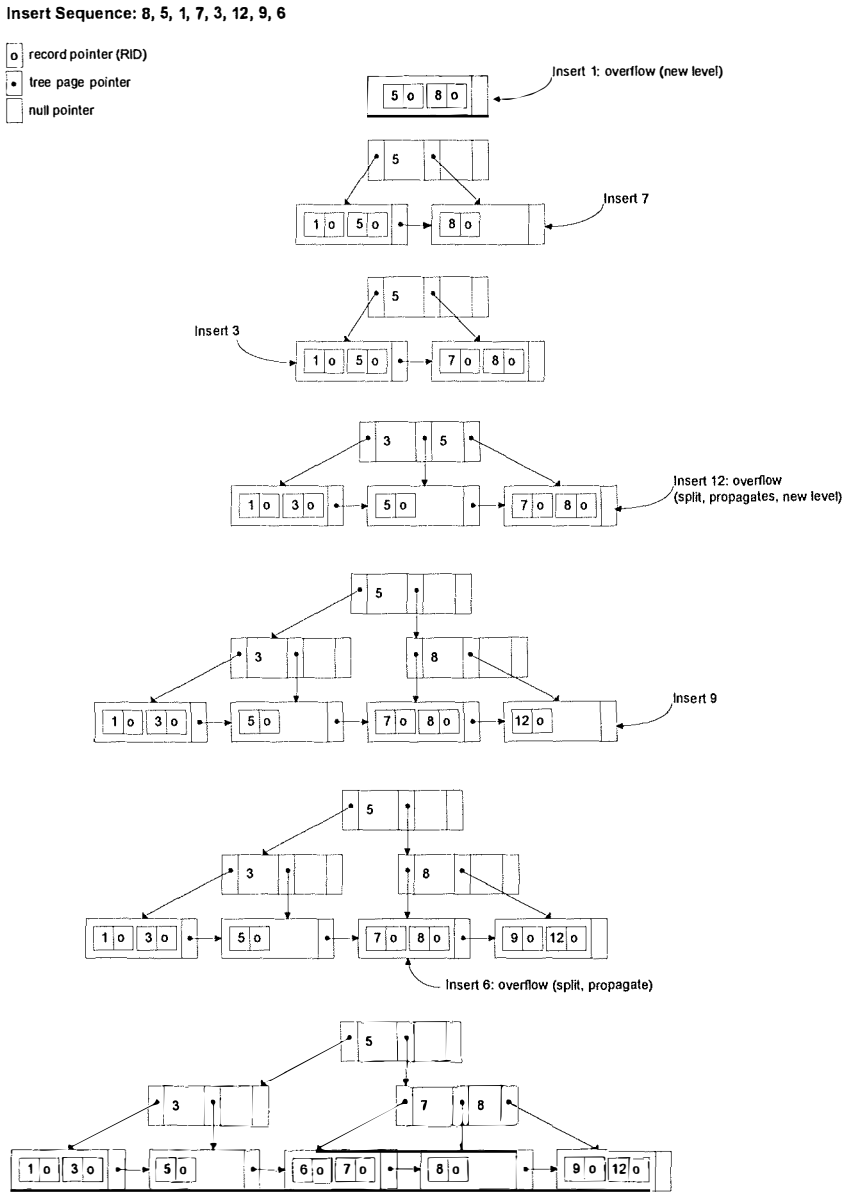


figure 3.8.: Inserts in a B-Tree [40]

Now, what happens when entries are ***deleted*** from the B-Tree in response to rows being deleted or attribute values updated in the indexed table? Figure 3.9. [O'Neil 1994] illustrates deletion from a B-Tree.

[40] [O'Neill 1994]

When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal page, it must also be removed from there. In the latter case, the value to its left within the leaf page must replace it in the internal page. Because that page is now the right most entry in the sub-tree. Deletion may cause *underflow* by reducing the number of entries in the leaf page to below the minimum required. In this case the system tries to find a *qualified* leaf page, a leaf page directly to the left or the right of the page with underflow. And redistribute the entries among the page and its sibling page so that at least both are hall full; otherwise, the page is merged with its sibling and the number of leaf pages is reduced. A common method is to try redistributing with the left sibling; if this is not possible, an attempt to redistribute with the right sibling is made. If this is not possible either, the three pages are merged into two leaf pages. In such a case, underflow may propagate to internal pages because one fewer tree pointer and search value are needed. This can propagate and reduce the tree depth.

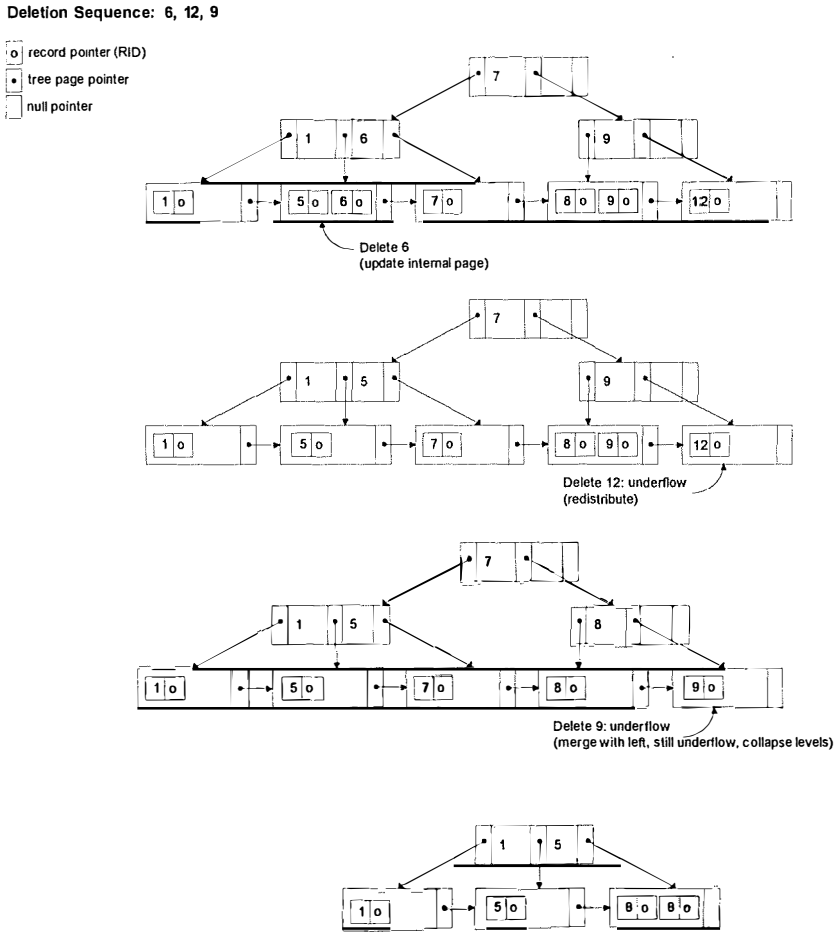


figure 3.9.: Deletion from a B-Tree[41]

[41] [O'Neill 1994]

An algorithm that performs the actions just outlined when entries are deleted from a B-Tree, could be used to keep a shrinking B-Tree balanced and all pages higher than the root at least half full. However, very few commercial database systems implement this algorithm, because they use to much disk I/Os. The logic to merge pages is somewhat more complicated, and requires extra I/O to keep pages well populated. Since disk space is cheap, most systems architects have decided to allow pages to become *depopulated without automatic reorganization*. The major reason for concern with a sparsely populated index is not the disk space, but the extra disk I/Os entailed in a range search for some number of entries that are spread on an unusually large number of leaf nodes. To respond to such inefficacy, the various database systems provide utilities to reorganize B-Trees. Such utilities duplicate the work of the original index creation and result in a clean new copy of it, with efficient disk utilization.

3.2.3. Clusters

Throughout this section we are going to describe the aspect of table clusters based upon [ORACLE 7.0]. ORACLE 7.0 offers the possibility of clustering tables, it is an optional method of storing table data. In fact a cluster can be seen as a group of tables that share the same data pages, because they share common attributes and are often used together. For example, the EMP and DEPT table share the DEPTNO attribute. As we define a cluster on the EMP and DEPT table (see figure 3.10.), all records for each department from both EMP and DEPT tables are physically stored in the same data page.

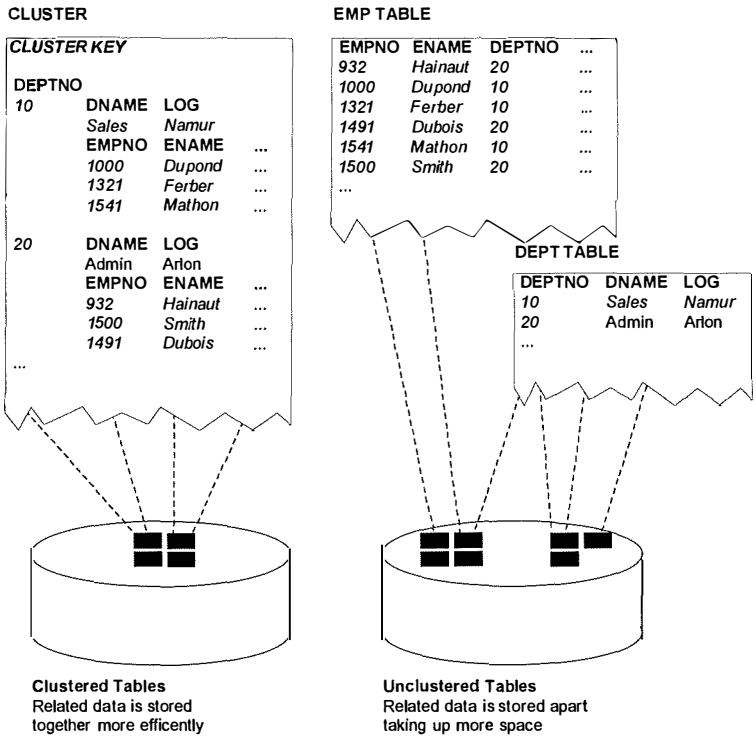


figure 3.10.: Clustered Tables[42]

[42] [ORACLE 7.0]

Two primary benefits can be brought to light when storing related records of different tables together in the same data page:

- *Disk I/O costs are reduced and access time improves* for joins of clustered tables.
- In a cluster, there is a key cluster value, which is the value of the cluster key attributes for a particular record. Each cluster key value is stored only once, each in the cluster and the cluster index, no matter how many records of different tables contain the value. Thus, *less storage might be required to store related tables* than in a non-clustered table format. For example, notice that each cluster key (each DEPTNO in figure 3.10.) is stored just once for multiple records that contain the same value in the EMP and DEPT tables.

We already revealed two primary benefits of table clusters, but there are more performance considerations as we will see.

It is obvious that clusters can reduce the performance of update statements (INSERT, DELETE, UPDATE) as compared to storing a table apart with its own index. As multiple table have data in each page, more pages are used to store a clustered table than if that table where stored non-clustered. Thus the performance disadvantage is related to the use of more space and the higher number of pages that have to be accessed to scan a table.

To identify cluster tables, the designer must look for tables that are related via referential integrity constraints and tables that are often accessed together in a SELECT statement, which joins two or more tables. If we cluster tables on the join attributes, we reduce the number of data pages that must be accessed when execution the query, as all records needed for the join lie on the same page. Thus performance for joins is improved. Similarly, it might be useful to cluster an individual table as we are going to see later on. For example, the EMP table can be clustered on the DEPTNO to group the records for employees of a same department. This is obviously an advantage if data operation commonly process records department by department.

For more details about Cluster we refer the reader to [ORACLE 7.0], however we give some guidelines for creating clusters.

Choose Appropriate Tables to Cluster: Use clusters to store table that are primarily queried (not predominantly INSERT, DELETE, UPDATE), and for which queries frequently join data of many tables in the cluster or retrieve related data from a single table.

Choose Appropriate Attributes for the Cluster Key: A good cluster key has enough unique values so that the group of qualified records to each value fills approximately one page. Not enough records per cluster value can waste space and result in bad performances. Cluster keys that are so specific, that only few records share a common values, can cause wasted space in pages, unless a small page size was specified at cluster creation time. Similarly, too many records per value can cause extra page accesses to qualify records for the given key. Cluster keys on values that have not enough cardinality (for example: male and female) result in excessive searching and can result in worse I/O costs than with non-clustered.

3.2.3.1. Clustered and Non-Clustered Indexes (Primary / Secondary)

As we already mentioned, a cluster can be defined on a single table. During this section we are going to see the implications of a clustered and non-clustered index defined upon a single table.

Placing records on disk ordered by some common index key value, is called *clustering*. An index with referenced records in the same order as its key values, is known as a *clustered index*, in some writings it is also referred to as a *clustering index*. A slightly more general concept is a *primary index* for a table, which determines the placement of the records, not necessarily the ordering, since there is no ordering in a hash index. The advantage of a clustered index is that certain query types are more efficiently answered, when the qualified records lie close to one each other. We think here about Range, Multipoint, Prefix match, Ordering, Grouping and Join queries.

In case where records with common index key values are clustered together and we read in the page containing one of the records with the given value, other records with the same value are likely to reside on the same page. Thus in accessing those records, we don't have to repeat the disk I/Os required to access the next record. Even if our database access method is relatively simple and accesses records only one at a time, on the basis of their RID values, we find that the second and successive records reside on the same page, already in the buffer.

Clustering indexes are *sparse* in some systems; like SYBASE, and *dense* in others, like DB2 and ORACLE. In other systems, like INGRES, the clustering index is sparse if based on an ISAM or hash structure and dense if based on B-Tree [O'Neil 1994]. Sparse clustering indexes hold pointers at leaf level for every page of the table being indexed. Whereas dense indexes hold RID pointers for every distinct attribute value. So sparse clustering indexes will often have fewer levels than dense ones. According to [Shasha 1992] sparse clustering indexes can reduce respond time by a factor of two or more.

Notice, that because a clustering index implies a certain physical file organization and the data file can only be organized one way at a time, there can be at most one clustering index per table.

A ***non-clustering index***, sometimes called *secondary index*, is an index on an attribute (or set of attributes) that puts *no constraint* on the table organization. They do not have the advantages of data proximity like the clustered once. Successive entries of a non-clustered index reference records on disk pages that are likely to be far apart, so there is no saving from one record to another.

Let us make sure, we have well understood the distinction between both index organizations. In a library, books may be clustered by access number, for example the ISBN number. The access number is the book's address in the library. Books with close related access numbers tend to be physically close one to another. In addition, there may be several nonclustering indexes represented by use of card catalogues. These are non-clustering, because two books with the same index entry may be physically far apart. For example, the same author may write a book about database tuning and another about a mathematical detective, both of the books will most probably not be placed near each other.

3.2.3.2. Evaluation of Clustered Indexes

We have seen the structure of the clustering index, know it is time to examine the advantages and disadvantages of clustered indexes in relation with the query types.

A clustered index offers the following ***benefits*** compared with non-clustering one:

- If the clustered index is sparse, then it will store fewer pointers than a dense index. Note that a non-clustering index is always dense. This *can save disk access* as more index entries can fit in a leaf page.
- A clustered index is ***good for Multipoint queries***. For example, equality predicate on a non-key attribute. A clustering index is useful for looking up names in a telephone book, because all people with the same last name are on consecutive pages. By contrast, a non-clustering index on the first three digits of subscribers phone number would be worse than useless for Multipoint queries. A query to find all subscribers in the 497 exchange might require an access to nearly every page.
  For the same reason, a clustering index will help perform an equality join on an attribute with few distinct values. For example, consider the equality join query on first names:

  ```
  SELECT Employee.ssnum, Student.course
  FROM Employee, Student
  WHERE Employee.firstname = Student.firstname
  ```

  If the table Employee has a clustered index on firstname, then for each Student record, all corresponding Employee records will be packed onto consecutive pages.
  If the Employee and Student tables both have clustering index in firstname, then the database system will often use a processing strategy called a *merge-join*. Such a strategy reads both relations in sorted order, thus minimizing the number of disk accesses required to perform the query[43]. This will also work if both tables have clustered index on first name, based on a hash structure that uses the same hash function.
- A clustered index based on B-Tree ***supports range, prefix match, and ordering queries well***. The pages of a telephone book provide a good example of the performance benefits. All names that begin with 'Ge%' will be on successive pages. The clustering index can also eliminate the need to perform the sort in an ORDER BY query on the indexed attribute.
- A clustered index on attribute or set of attributes can *reduce lock contention* in two ways.
  - A retrieval of several records with the same value, a prefix match query, or a range query will access and lock only a few consecutive pages of the table. If the table is non-clustered or clustered on some other attribute(s) then such queries may access many more pages. It can happen that we have a different page for each record. *The fewer pages accessed, the fewer pages are locked in the systems that use page level locking.*

[43] Each page of each table will be read in once.

- A clustering index can *eliminate a concurrency control* hot spot in intensive insert environments. Such environments make a hot spot of the last page of the heap organization. A clustering index based on a hash structure will always eliminate this hot spot on the table. A clustering index based on a B-Tree will also eliminate the hot spot provided the *key is not sequential*.

The main ***disadvantage*** of the clustering index is that its benefits can diminish if there are a large number of overflow data pages. The reason is that accessing such pages will usually entail extra disk I/Os. Overflow pages can result from two kinds of updates.

- Inserts may cause data pages to overflow, and they have to split with all known consequential effects on the index entries and/or levels.
- Record replacements that increase the size of the record. For example, the replacement of a NULL value by a long character string, or that change the indexed key value will also cause overflows and/or underflow of data pages with effects on the index entries and/or levels.

3.2.3.3. Evaluation of Non-Clustered Indexes

Because non-clustered indexes on a table do not impose any constraints on the ordering, there can be *many non-clustering indexes on a given table*.

A non-clustering index can eliminate the need to access the table. For example, consider that there is a non-clustering index on attribute A1, A2 and A3 of table T1. Then the following query can be answered completely within the index, without accessing the data pages.

```
SELECT A2, A3
FROM T1
WHERE A1 = 5;
```

If your system takes advantage of this possibility, non-clustering indexes will give better performance than sparse clustering ones. Of course, updates would need to access the data pages of table T1.

Suppose the query must touch the table T1 through a non-clustering index based on A1. Let Nr be the number of records retrieved and Np be the number of pages for T1. If Nr < Np, then approximately Nr pages of T1 will be logically read. The reason is simple, it is likely that each record will be on a different page. If Nr > Np, then more than Np pages may be retrieved if the buffer pool is smaller than the table size.

Thus, *non-clustering indexes are good if each query retrieves significantly fewer records than there are pages in the file,* in other words if the query has a good selectivity. For the moment we still use the word 'significant', but later on we see how the number of record to be retrieved behaves according to access performances. However the word is well chosen, because a table scan can often save time by reading many pages at a time, provided, the table is stored contiguously on tracks. For example, INGRES normally reads 8 pages at a time on a scan. Therefore, according to [Shasha 1992], even if the scan and the index both read all the pages of the table, the scan may complete by a factor of 2 to 10 times faster.

Consider a table T1 with 50 byte records and pages of 4Kb long. Assume further that attribute A1 takes 20 different values, which are *evenly distributed* among the records. The question is to know if a clustering index on A1 a help or a hindrance?

Each Multipoint query on attribute A1 will retrieve approximately 1/20 of the records. Because each page contains approximately 80 records, nearly every page will have a record for nearly every value of A1. So, using the index will give worse performance than scanning the entire table.

Consider the same situation, except that each record is 2Kb long. In this case, a Multipoint query on the non-clustering index will touch only every tenth page on the average, so the index helps at least a little.

We can draw three guidelines from these examples.

- A non-clustering index serves you best if it avoids touching a data page. This is possible for certain types of queries some Point or Multipoint queries, as well as Count, Join and Existence queries that depend on the key attributes of the non-clustering index.
- A non-clustering index is always useful for point queries.
- For Multipoint queries, a non-clustering index may or may not help, depending on the selectivity. However a good rule of thumb is to use the non-clustering index whenever the following holds:

  number of distinct key values > c * number of records per page,

  where *c* is the number of pages that can be prefetched in one disk read.
  This inequality implies that the use of the non-clustering index would entail fewer disk accesses than scanning all the pages of the table.

There are two situations when you should *never use* a non-clustering index.

- When the activation frequency of update operations compared to the activation frequency of data access operations is high. As a rule of thumb [Shasha 1992], *at least one-third as frequent*. A update operation is either an insertion, deletion or update to one of the values of the index attributes. The reasons are, like we have seen, that modifications entail at least two disk accesses when not more, because of the dynamic index reorganization.
- When lock escalation occurs. In some systems, a table with many non-clustered indexes may cause a transaction that does a substantial number of updates to escalate to table-level locking. Escalation occurs when a transaction acquires more than a predetermined threshold of locks [Shasha 1992].

3.2.4. Hash index

As already mentioned earlier in this section, the hash index, same as the clustered index, determines the record placement on disk. We say that the hash index is a *primary index*, because the record placement in the table is computed out of the index values, as we will see.

A *hash primary index* for a table provides look up by a method that is entirely different from the B-Tree structure. Note that hashing techniques can be used to construct the leaf level in an index, however we will not consider this structure in this paper. With a hash primary index, records inserted in a table are placed in a pseudo-random data page determined by a *hash function* applied to the key value, and retrieved the same way, usually with a single I/O operation. There is no key value directory (like the leaf level in B-Tree), so look up depends on proceeding directly to appropriate pseudo-random data page slots by hashing the key value.

There is *no order by key value possible* in such a structure. Normally in hashing, we can only ask for a specific key value, not for the sequent key values, as it is possible in a B-Tree structure. We would expect two records with successive key values to be located on entirely uncorrelated data pages, depending on the caprice of the hash function. As a result, Multipoint query is not well served by a hash index, because a brute force scan of the table is needed. This is a serious limitation, but the compensating value of hash indexes lies in rapid access to records in large tables by point queries. The ability to access a desired record with single I/O is a great advantage when compared with B-Tree, which requires extra I/O for directory search in large tables (in numerous cases 2-3 disk accesses)

3.2.4.1. Hash Function and Collisions

As we noted earlier in this chapter, hash indexes exist in INGRES, and an alternative hash feature exists in ORACLE version 7, but hashing is not possible in DB2 at this time[44]. We will illustrated the discussion on hash indexes by referencing the INGRES hash structures tables.

Hashing, or sometimes called hash-addressing is a technique for providing fast direct access to specific stored records on the basis of a given value for some field. The field in question is usually, but not necessarily, the primary key of a table. In outlines the technique works as follows.

Each stored record is placed in the table at a location, a *slot*, whose address (RID, or perhaps just page number) is computed as some function, the *hash function*, of some set of attributes of that record. So, hashing consists off applying a hash function **H** to the key[45] values of records and determine this way the location of the record in the database. A common class of hash function can be called "division/remainder". For reasons that are beyond the scope of this paper, the divisor of such a hash function is usually chosen to be a prime number.

---

[44] According to [O'Neill 1994]

[45] As with B-tree, we do not insist on unique key values unless the keyword *unique* is explicitely used.

To retrieve the record subsequently given the key value, the database system performs the same computation as before and fetches the record at the computed position.

Figure 3.10. illustrates the insertion of data records[46] by using a hash function. Note that in the figure we only represent the key values and no the whole records, this simplifies the figure. Slots for a given key value might be determined in a number of different ways. One way is to hash a given key value to a given disk page and then find an empty slot somewhere on the page where the record can be stored. However, we assume in what follows that the precise slot on any page of the table is being determined by the hash function. If this slot is already in use, a situation known as **hash collision** occurs, and **collision resolution** is performed in some deterministic way to locate an empty slot.

Hash collisions occur when the hash field value of the new added record hashes to an address that already contains a different record. In this situation we must insert the new record in some other slot position, as its hash address is occupied. The process of finding another position is called **collision resolution**. There are numerous methods for collision resolution, including the following:

- *Open addressing:* simply consist of looking at all other slots on the same page in some predetermined order until an empty one is found. In open addressing, *every attempt is made to find a new slot on the same page*, to minimize the number of I/Os needed to search through a chain collision. Only if we run out of space in a page, then a slot on a succeeding page is used[47].
- *Chaining:* various overflow locations are kept, usually by extending the initial number of pages with a number of overflow pages and their slots. The collision is resolved by placing the new added record in an unused overflow slot and setting a pointer, of the occupied slot, to the address of the overflowing slot. A linked list of overflow records for each hashed slot is maintained.
- *Multiple hashing:* the system applies a second hash function if the first results in a collision. If another collision results, the system uses open addressing or applies a third hash function.

Each of the above collision resolution methods requires its own algorithms for inserts, retrieval and deletion of records. The algorithms for chaining are the simplest. Deletion algorithms for open addressing are rather tricky, because the system does not remove the record physically from its position but only lists it as deleted.

In figure 3.10. we illustrate the insertion of the record with the given key value 55 in the slot that has been determined by hash function, H, and then rehashed. We see, that the record with key value 55 has collided in slot 66 with a record already stored there, and then has been rehashed to slot 69.

---

[46] Note that we used the same values than those in figure 3.7. for the B-tree

[47] [O'Neill 1994]

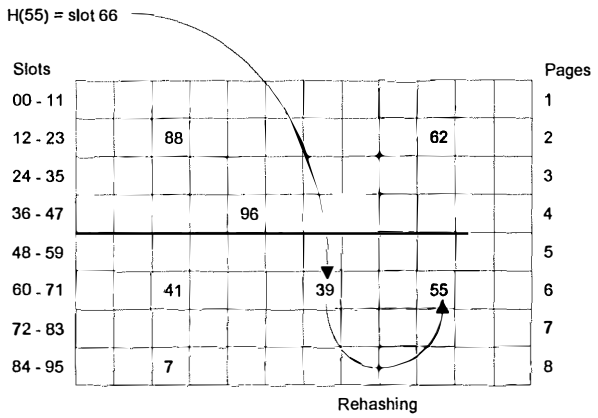


figure 3.10.: Hash Structure Table. Record Insertion 55 with Collision[48].

The goal of a good hashing function is to distribute the records uniformly over the slots so as to minimize collisions, while not leaving many unused slots. Simulations and analysis have shown that it is usually best to keep a hash table between 70% and 90% full so that the number of collisions remains low and that space is not wasted. Hence, let us see how the database system determines number of slots and data pages necessary for a given table. At the time the hash structure is created[49], a certain number of slots, Ns, on a known sequence of pages is set aside for usage. Consider that there are Nr records in the table, and that we defined a fill rate of fr (an integer from 1 to 100), then the number of slots set aside by the system can be expressed by the following relation:

$$Ns = \lceil (Nr * (100/fr)) \rceil$$

For example, if we have 100.000 records in the table and a fill rate of 70%, we would set aside $Ns = \lceil 100.000 * (100/70) \rceil = 142.858$ record slots.

Next let us calculate the number of pages Np required for the estimated number of slots. We assume that we can guarantee placing at least Nrp records per page. The number of pages used will be given by the following relation:

$$Np = \lceil (Ns / Nrp) \rceil$$

In the above given example with 143.858 slots, consider that 20 records can fit on a page, then the number of pages required is equal to $Np = \lceil 142.858/20 \rceil = 7.143$ pages.

---

[48] [O'Neill 1994]

[49] INGRES used a Modify command to give a table a hash structure.

```
MODIFY tablename | indexname
        TO HASH [UNIQUE] [on columnname {, columnname}]
        [WITH    [LOCATION = . . . ]
                 [MINPAGES = n] [, MAXPAGES = n]
                 [FILLFACTOR = n] ];
```

3.2.4.2. Fixed Number of Slots

It is important to realize that once the number of slots Ns specified, it cannot be enlarged, as it can with B-Tree node splitting. The reason for this limitation is that the hash function is a two phases calculation. The first phase generates a pseudo-random number based on the key value, x = r(keyvalue), where x might be a floating point number, uniformly distributed in the range of 0 < x < 1. Throughout the second phase the slot number H(keyvalue) resulting from the hash function can be generated, for example with the following relation:

$$H(keyvalue) = \lfloor Ns * r(keyvalue) \rfloor$$

The relation results in a random slot number from sequence 0, 1, ..., Ns-1. We consider this two phases approach because in this way the generic function r can easily lead to a uniform distribution of integers ranging from 0 to Ns-1, for any given value of Ns. However, if the total number of slots was changed, say to Ns', we would find that the hash function

$$H'(keyvalue) = \lfloor Ns' * r(keyvalue) \rfloor$$

might give the same slot number for all slot placements previously calculated.

For example, if r(keyvalue) is 0,33334, and Ns is 2, then

$$H(keyvalue) = \lfloor 2 * 0{,}33334 \rfloor = 0$$

However, if Ns is 3 we would have

$$H'(keyvalue) = \lfloor 3 * 0{,}33334 \rfloor = 1 .$$

This is the reason why we cannot enlarge the number of pages within a hash organization. It is helpful and even necessary to reorganize the table completely when the average length of collision chains begins to enlarge, because performance decreases rapidly as the pages fill up. Most database systems offer statistics on collision chains length to aid the DBA in tuning.

3.2.4.3. Collision Chain Length and Page Overflow

The major advantage of the hash structure is that it is usually possible to go directly to the qualified page, where the qualified records are located. This is likely if we can use a lot of disk space to specify a very low fill rate, but there is an enormous waste of space. In such a situation the occupancy of the pages tends to be low, and the collision chains short. Therefore, the probability to find each record on its originally hashed page it high.

Recall that there exists various techniques to solve the problem of hash collisions. One of the techniques is the known as open addressing, a second is known as chaining and a third is known as multiple hashing. Figures 3.11 and 3.12 [Hainaut 1994] show the evolution of the average I/O cost (number of physical access) depending on the page's fill rate, fr, and the number of records to be stored, thus on the number of records within a data page, Nrp, and the number of pages, Np.

Note that these figures confirm the fact that the more a page fills up, the more the fill rate increases, the more there are hash collisions, thus the higher the I/O cost is. However, we can also observe that for a small fill rate the more a page can hold records the better the I/O cost is, but the more space is wasted.

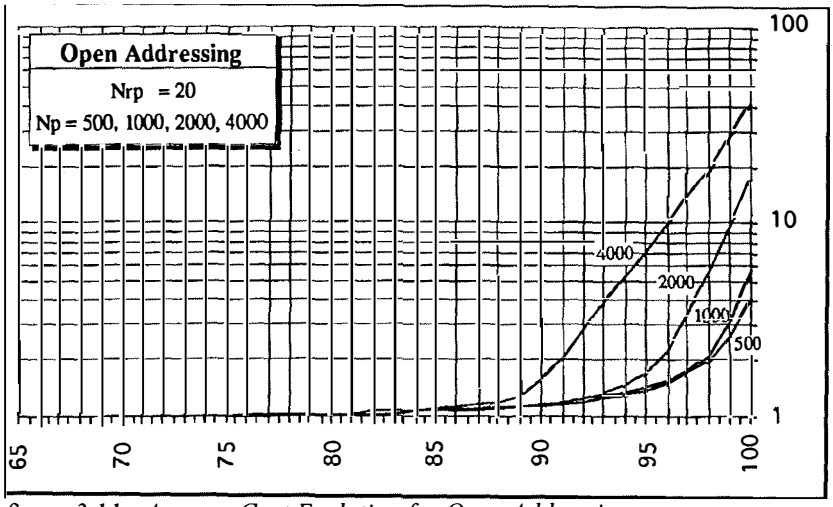


figure 3.11.: Average Cost Evolution for Open Addressing.

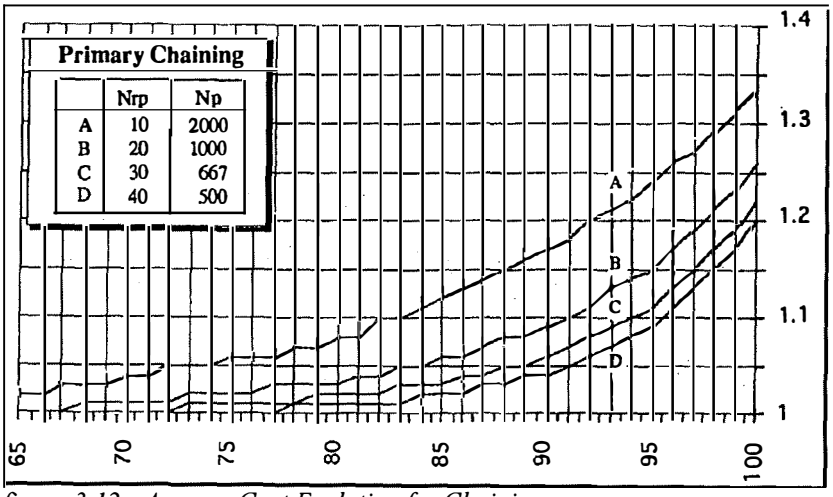


figure 3.12.: Average Cost Evolution for Chaining.

Throughout the next lines we will try to determine the average collision length for the open addressing, given a fill rate [O'Neil 1994].

First we want to estimate the probability, P, that the last hashed record, as it is added, encounters a filled slot in its first probe. Consider a given fill rate, fr. The likelihood that any given hash page is occupied is P = (fr/100), assuming that the records are randomly distributed among the slots. This means that the probability that the slot is empty is 1-P, this is also the probability that we will be able to place the new added record in the first position we come to. We call this *'a collision chain of length 1'*, because only one probe is necessary, and write it as:

Pr(collision chain length 1) = (1 - P)

On the other hand, in order to have a collision chain of length 2, the first position hashed to must be full, with probability P, and the second position that we reach in the rehash sequence must be empty, with probability 1-P. Using the principle of multiplication by which we calculate the probability of two or more independent events happening together:

$$Pr(\text{collision chain length } 2) = (1 - P) * P$$

Now for a collision chain of 3, we must start with full slots in the first two positions we reach, with probability $P * P = P^2$, and then an empty slot in the third with probability P. Simple extension of this argument gives:

$$Pr(\text{collision chain length } 2) = (1 - P) * P^2$$

$$Pr(\text{collision chain length } 2) = (1 - P) * P^3$$

...

$$Pr(\text{collision chain length } 2) = (1 - P) * P^{K-1}$$

Now the expected length of the collision chain, E(L), is given by the sum of all these probabilities times the associated lengths:

$$E(L) = (1 - P) + (1 - P) * P + (1 - P) * P^2 + \ldots + (1 - P) * P^{K-1}$$

or factoring:

$$E(L) = (1 - P) * (1 + 2P + 3P^2 + \ldots + KP^{K-1})$$

where the sum extends to some large number, K, of terms, proportional to the maximum number of possible collision in the table. Now we would like to be able to give a simple formula for this sum. To see how to do, start by considering the function f(x) given by the infinite series:

$$f(x) = x + x^2 + x^3 + x^4 + \ldots$$

This is the well known infinite geometric progression, $a + ar + ar^2 + ar^3 + \ldots$, with a and r forming x. The formula for the sum is known from algebra, a/(1 - r), so we can give a closed firm solution for the infinite series f(x):

$$f(x) = x + x^2 + x^3 + x^4 + \ldots = x/(1 - x)$$

Now, taking the derivative of all terms in the equations, we get:

$$f'(x) = 1 + 2x + 3x^2 + 4x^3 + \ldots = 1/(1 - x)^2$$

Rewriting the relation for the expected length E(L) of collision chain, we see that we can represent the infinite sum on the right with f'(P), replacing x with P in the left hand equality of equation f'(x):

$$E(L) = (1 - P) * (1 + 2P + 3P^2 + 4P^3 + \ldots) = (1 - P) * (f'(P))$$

Now using the right hand equality of equation f'(x), we can replace f'(P) with $1/(1 - p)^2$, to get:

$$E(L) = (1 - P) * (f'(P)) = (1 - P) * (1/(1 - p)^2) = 1/(1-P)$$

Thus we see that the expected length of collision chain is the reciprocal of (1 - P). Recall that P = (f/100) and consider a few examples. If the hash structure is 50% full then P = 0,5, and E(L) = 1/0,5 = 2. If the table is 90% full, then P = 0,9 and E(L) = 10. The graph of this relationship is given in figure 3.12..

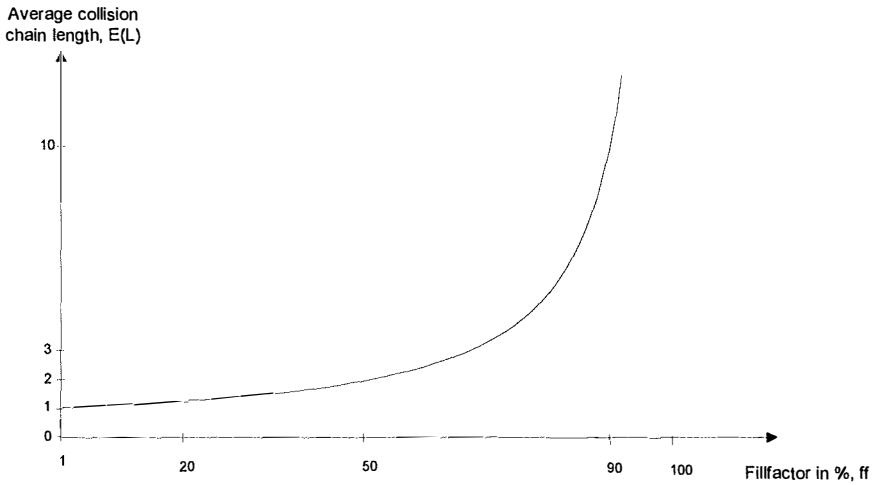


figure 3.11.: Relationship between E(L) and the fill rate, fr.

As we would expect, the more full we set the table, the more there will be collisions and the longer the average collision chain will be. What might be surprising, is how quickly the chain increases in length once the fill rate comes close to 100.

Assume that we can fit 20 records on a page and set and load a table half full, fr = 50%, it seems quite unlikely that a collision chain of average length 2 will grow long enough (L=21) to continue to a successive page. However, if the fill rate is 95%, the average length of a chain is 20, so about half the collision chains will continue to a new page, and cause a supplementary page access during retrieval. The point of all this is to show how important it is to keep the fill rate small relative to 100. On the average, we will be able to find a record associated with a *unique key value half-way through a collision chain*, which entails only a *single disk access*. Significant overhead, and supplementary disk accesses, start to occur when a significant number of entries start to hash to position 21 or later of the chain.

A hash table containing *duplicate key values* tends to have *longer collision chains*, since equal value records are certain to collide. In our derivation, we assumed independent random positions for separate hash entries, which is obviously not the case when duplicate entries exist.

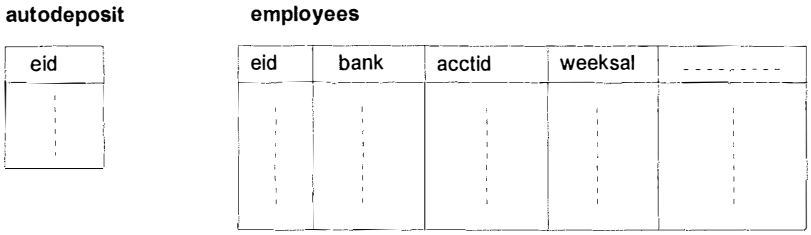
3.2.4.4. Evaluation of Hash Primary Index

To begin with, it should be clear that hash primary index *is extremely efficient with equal predicates in key attributes* (example id = 12345), such as Point queries. All other query types, like Range, Prefix Match, Extreme, Grouping, Ordering etc... queries, can not be solved efficacy by a hash index. The system falls back on another access method, such as table scans or B-Tree, to solve these queries. Assume that the hash index is the only access structure present for a given attribute, and the range predicate is the only predicate present in the Select statement, then the table scan seems the appropriate access method. Since slots of a hash table are rather sparsely filled to restrain collisions, a table scan on a hashed table entails more I/Os than usual in a sequential structure.

Another disadvantage of the hash primary index is that we have to leave room for expansion in the initial layout of the table, rather than depending on incremental expansion for later insert. Given normal uncertainty about table expansion, we usually tend to overestimate the extra space needed, and therefore waste disk space.

A third point is that if we use non key values (Multipoint queries) in the predicate, the retrieval routine will need to go through the entire collision chain for a given value. It should also be clear that if we sometimes have a large number of duplicate values or a poor selectivity there could be a very long collision chain, which detracts seriously from the efficacy of the structure. Note that 'NULL' values count as duplicate values.

Let us see throughout a small example how consideration arise in selecting a hash index structure or a clustered B-Tree structure. Consider the following two tables



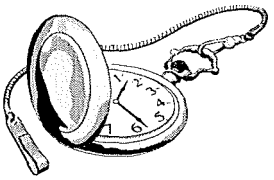
The autodeposit table, is a single-column list of employees (eid) for individuals who have asked that their weekly checks be automatically deposited. The table employees is constituted of the employee's number (eid), the deposit bank (bank), the account number (acctid) and the weekly salary for each employee (weeksal).

A common type of application program, performed once a week, would read the eid values to access the records of the employees table and make the desired automatic deposit in the appropriate account.

This appears to be a long sequence of indexed access to the employees table throughout an index on eid. We might decide to use a primary hash index on eid for the employees table, since this will give us the most efficient I/O access to these records. However, this approach would be a mistake. If instead we give the employees table a clustered B-Tree index on eid, the resulting table would be clustered by eid value. Now if we also cluster the records in autodeposit table by eid values, we would find that the application, looping through the autodeposit records, makes all accesses to the employees table in order by the clustered key. In general, successive accesses to the employees table pass down through index nodes, that always remain in memory buffer from prior access, and then to a record on a page that is already in memory buffer.

The total number of employee pages involved in I/O is equal to the number of pages in the table. If we assume that a large portion of employees receive autodeposit, so most employee table data pages will be involved. On the other hand, in the hash case, as each new record is accessed it lies on a random page, unrelated to the previous one, even if it was accessed by contiguous key. If there are, say, an average of 10 records on each hash page, that desire direct deposit, there will be 20 times as many I/Os in the hash primary index case than in the B-Tree clustered index case. We are between accesses, and that the B-Trees index page are few in number compared with the data pages, both common assumptions.

Thus we see an example where the hash structure is not as I/O efficient for record access as another ordered structure. Of course we stacked the deck, since the eid references are not actually random. On the other hand, this is a common situation, and one should be aware of how the record order within the table can improve resource saving in accessing the table.



Chapter 4. I/O Cost Estimations

In chapter 2 we examined the data operations and classified them into a set of query types, whereas in chapter 3 we described some of the different data structures that allow us to structure and to access data. We arrived at a point where we have to combine both features and find out how we can estimate I/O cost estimations for different types of query, according to different access structures and search techniques. In this chapter we will see if we can shed some light on the I/O cost estimations. We will try to estimate the I/O cost, by using some simplified relations. Note that if we use pessimistic estimations we can assume that in real life, access performances are generally better than the one we estimate. The fact that real life performances will surely be better then the one we estimate does not influence our search for the optimal index solution.

Given the multitude of parameters that influence I/O cost estimations, first it is helpful to understand the influence of the different parameters upon the cost estimation, second it is helpful to compare parameters between on each other. Throughout this section we will see that the filter factor plays an important role during cost estimation, for this reason we will only compare the filter factor to other parameters, such as the fill rate, the page size, the number of records and the fan out.

4.1. Brute Force I/O Cost

A brute force table scan is an algorithmic step, where all records in the table are scanned and only the records qualified by the select_filter, WHERE clause, are retrieved. In many database architectures there are situations in which distinct record types can be mixed on common extents of a tablespace[50]. However, in what follows we assume that all pages referenced in a tablespace contain only one type of record type.

Recall the relation, from chapter 3, for estimating the number of pages needed to store all records:

$$Np = \left\lceil \frac{Nr}{\left\lfloor \frac{(Ps - Hs) * fr}{Rs} \right\rfloor} \right\rceil$$

where
Np: Number of data pages
Nr: Number of records to store
Ps: Page size
Hs: Page header size
fr: fill rate
Rs: Record size

Assume that we are given an Employee table with Nr = 200.000 records, each record is of fixed length Rs = 200 bytes, and data pages have a fill rate fr = 70%. We assume that each 2-Kb data page uses roughly Hs = 48 bytes of overhead, leaving 2000 bytes, and with data pages loaded 70% we have 1400 available bytes. So a data page can hold up to a maximum of Nrp = 7 records. Thus, the total number of data pages needed for storing 200.000 records is equal to Np = 28.572 pages.

Consider the following point query (predicate on a key attribute):

```
SELECT eid, ename
FROM Employee
WHERE socsecno = 123456789;
```

where we search for the identification number, eid, and the name, ename, of an employee with a given social security number, socsecno.

Assuming that we do not have an index on attribute socsecno, our only technique to retrieve the requested employee, is to look at all Np data pages to find the record that satisfies the predicate. We are calling this technique the *brute force table scan* or *the sequential table scan.* Note, that we might know, at priory, that there is only one record satisfying the predicate, however the optimizer probably will not, since socsecno has no index, and statistics usually do not include such details for all attributes.

For our example, as there are 28.578 data pages that have to be scanned. The I/O cost is equal to $COST_{I/O}(PLAN)$ = 28.578 random I/Os. Recall that we do not try to estimate the time needed by the CPU to execute the query, $COST_{CPU}(PLAN)$, however we assume that the total I/O cost is, normally, proportional to the I/O cost.

[50] Because of this feature we should speak about tablespace scan and not about table scan.

4.1.1. Brute Force I/O Cost Estimation

Following the above example, the $COST_{I/O}(PLAN)$ for a *brute* force table scan, based on random page accesses, can be estimated by the following relation:

$COST_{I/O}(PLAN) = Np * R$

where
Np: number of data pages
R: fraction of time necessary to perform a random read (generally 1/40 seconds)

relation 4.1.: Cost estimation for Brute Force Table Scan

However, as already mentioned, in chapter 3., the table scan has some tricks to offer, like the *mutli-block access I/O*. Until now, we assumed random access and estimated that 28.578 accesses are needed. Considering that they are performed one after the other, they require 28.578 times as long as a single random access.

The idea behind the multi-block prefetch access is that the system specifies a large number of data pages, in sequence, to be accessed, most commonly 32 pages. This sequence of requests is communicated to the disk controller in a manner that allows the controller to read successive pages on a track. As a result, the database system is able to perform 32 page reads in sequence at full rotational transfer rate of the disk. Therefore at a much lower cost in terms of elapsed time during which the disk arm is employed. Figure 4.1. [O'Neil 1994] gives a comparison between a the multi-block access of 32 disk pages in sequence and 32 random accesses. Assuming that a random access needs 0,025 second.

	Random Access (in seconds)	*Multi-Block Access (in seconds)*
Seek time	0,016	0,016
Rotational latency	0,008	0,008
Transfer time	0,001	0,048 (32 pages)
Total access time	0,025	0,072
Total for 32 pages	0,800	0,072

figure 4.1.: Time Comparison between Mutli-Block and Random Access time of 32 pages

The value of 0,800 sec. for 32 random reads is approximately ten times larger than the value of 0,072 sec. for a multi-block access of 32 pages. In general, we can use the rule of thumb that multi-block access proceeds ten times faster than single random accesses, defining a rate of 1/400 sec. per page access.

We estimated for the above Select statement an I/O cost of 28.572 random reads, which needed 28.572/40 = 714,3 seconds. If we assume that we can use the technique of the multi-block access to access the 28.572 data pages, then the query requires 28.572/400 = 71,43 seconds

Given the technique of the multi-block or mutli-block access, we can define the following relation for a brute force table scan:

$$COST_{I/O}(PLAN) = Np * S$$

where
S: fraction of time necessary to perform a mutli-block access (1/400 sec)

relation 4.2.: Cost estimation for Brute Force Table Scan using Mutli-block access

There exists also another kind of prefetch, known as the *list prefetch*, which we already described in section 3.1.2.. Recall that the list prefetch provides a list of pages (usually 32), to the disk controller, that need to be retrieved from disk. It is obvious that the list prefetch is more efficient than the random I/O request, because the disk arm is programmed to retrieve all pages in the most efficient way. However, the list prefetch is not as efficient as the mutli-block prefetch, which is an unachievable optimum. The speed of the list prefetch is determined by how far apart the pages are on disk, but we can consider as a rule of thumb [O'Neil 1994] that the list prefetch proceeds at 100 I/Os per second. It is obvious that the list prefetch is not an I/O technique which is used with the brute force table scan, it is more accurate to use it in an *index scan* to retrieve that qualified data pages.

If we use the list prefetch to retrieve the qualified pages the I/O cost results in 28.572 list prefetch reads. Based on list prefetch total access time, the query requires 28.572/100 = 258.72 seconds.

Given the technique of list prefetch, the following relation gives us the I/O cost, in terms of seconds, to retrieve a set of qualified data pages:

$$COST_{I/O}(PLAN) = Np * L$$

where
L: fraction of time necessary to perform a list prefetch read (1/100 sec)

relation 4.3.: Cost estimation for a List Prefetch Read of Np pages

The I/O cost estimations, we just explained do not consider the predicate filter factor. Recall that the *filter factor,* of a select_filter is defined as the product of all predicate filter factors that compose the select_filter. The reader finds formulas to estimate the filter factor in section 2.1.5.. However, recall that the average *number of records to be retrieved*, k, by an SQL statement is inferred by the filter factor, as shown by relation 4.4.

$$k = ff * Card(T)$$

where
ff: Filter Factor for predicate P
CARD(T): Number of records in table T

relation 4.4.: Number of qualified records of a given select_filter

Same as, the number of records to be retrieved, the *selectivity,* s, is inferred by the filter factor, as shown by relation 4.5.. The selectivity defines the percentage of records that are *not qualified* by the select_filter.

$$s = (1 - f(P))$$

where
ff: filter factor of predicate P

relation 4.5.: Selectivity of a select_filter

Throughout all our examples and descriptions, we assumed that the k records, are *randomly distributed* among a set of Np pages. Making this assumption, we are able to foresee the number of pages that contain the k records. If we also assume, that the probability that a record is located within a page is independent of the probability that another record lies within the same page, then the probability that no record is located in a page is equal to $(1\text{-}ff)^{Nrp}$. The probability that a page contains one of the qualified records is equal to $(1 - (1\text{-}ff)^{Nrp})$. Multiplying this probability by the number of pages and rounding it up to the next higher integer we get, per se, the number of pages that contain all the qualified records.

Consider the above probability and a data page of Nrp records. Relation 4.6. gives a rude approximation of the number of pages, K, that hold the k records:

$$K = \lceil (1 - ff^{Nrp}) * Np \rceil$$

where
Nrp: number of records per data page
Np: number of data pages
ff: filter rate

relation 4.6.: Number of Pages that contain the qualified records

Although we know, per se, the number of pages that hold the qualified records, the DBMS execution plan does not know, per se, which pages contain which records, qualified or non-qualified ones. Therefore we can not induce, for a brute tables scan, that the I/O cost is equal to the number of pages that hold the qualified records.

Thus, we should use relation 4.7. to estimate I/O cost in case of a brute force scan. Relation 4.7. defines a pessimistic cost estimation, where all data pages have to be consulted to retrieve all qualified records. In the next section, where we bring the relation face to face with the query types, we will see that this estimation is too pessimistic.

$$COST_{I/O}(\text{BRUTE FORCE}) = Np * \beta$$

where
β: time fraction R, L or S, depending on whether we use Random, Multi-block prefetch or Sequential reads

relation 4.7.: Pessimistic Cost Estimation for Brute Force Table Scan

4.1.2. Brute Force I/O Cost Estimation and Query Types

First, let us see what happens to the I/O cost estimation when a *point query* is committed and we use a brute force table scan. Recall that a point query *returns one and only one* record from the table. We can say, without no great harm, that not all data pages are scanned to find the qualified record, but as a rule of thumb we can say that on the average half of the pages are scanned. The following relation holds for a brute force scan related to a point query.

$$COST_{I/O}(\text{BRUTE FORCE}) = Np/2 * \beta$$

where
Np: Number of data pages
β: time fraction R, S or L, depending on whether we use Random, Multi-block prefetch or Sequential reads

relation 4.8.: Cost Estimation for Brute Force Scan with Point Query

All other types of queries, like Multipoint, Range, Prefix Match, Extremal, Ordering, Group by and Join queries, involve more than one record in the list of their qualified records. Let us group the Multipoint, Range, Prefix Match, Extremal, Ordering, Group by and Join queries into one single type of queries, named ***mutlirecord queries***. As mutlirecord queries retrieve more than one record the whole set of table pages has to be scanned for query evaluation. Relation 4.9. give a pessimistic estimation of I/O costs for multirecord queries.

$$COST_{I/O}(\text{BRUTE FORCE}) = Np * \beta$$

where
Np: Number of data pages
β : time fraction R, S or L, depending on whether we use Random, Multi-block prefetch or Sequential reads

relation 4.9.: Cost Estimation for Brute Force Scan with Multirecord Queries

By making the distinction between point queries and multirecord queries we defined to relation for estimating the I/O cost in case of a brute force tables scan. However, according to different writings we can say that on the average the DBMS accesses half of the pages to retrieve the qualified data. So relation 4.8. can be used as a general form for brute force I/O cost estimation.

We did not group Join queries with the Multirecord queries, as Join queries, in general, involve more than one table. Recall that the most used techniques to join two tables are the nested loop join and the merge join. As already defined in section 2.18., the *nested loop join* has the following relation to estimate I/O costs,

$$COST_{I/O}(\text{NESTED LOOP JOIN}) = COST_{I/O}(\text{BRUTE FORCE OUTER TABLE}) + (\text{NUMBER OF QUALIFYING RECORDS IN OUTER TABLE} * COST_{I/O}(\text{BRUTE FORCE INNER TABLE}))$$

relation 4.10.: Cost Estimation for Nested Loop Join using Brute Force Scans

whereas, the *merge join* has the following I/O cost estimation

$$COST_{I/O}(\text{MERGE JOIN}) = COST_{I/O}(\text{BRUTE FORCE OUTER TABLE}) + COST_{I/O}(\text{BRUTE FORCE INNER TABLE})$$

relation 4.11.: Cost Estimation for Merge Loop Join using Brute Force Scans

To get better understanding of these two relations, we refer the reader to section 2.8.4., where we give an explicit description of joining techniques and examples of their cost estimations.

4.2. Index I/O Cost

In the preceding section we recalled the relation for estimating the number of pages needed to hold the data records, Nrp, the notion of filter factor, ff. We also defined relations for brute force table scan I/O cost estimation related to the different query types. Knowing, that indexes are used to speed up data retrieval for a given set of queries, we first have to see how we can estimate I/O costs for the different kind of indexes, before we can decide if an index speeds up retrieval for a given set of queries. We will give rude approximations for clustered and non-clustered B-Tree I/O costs. We will also give a cost relation for Hash index.

4.2.1. B-Tree I/O Cost Estimations

Recall that the structure of a B-Tree index is made of three parts. First, the tree structure grouping the *index internal pages* into a structure, which will guide the search for the qualified records to be retrieved. Second, the *index leaf pages structure*, also known as the index dictionary, which holds the references, RID pointers, to the qualified records and/or data pages. Recall that the index dictionary can be dense or non-dense. As a rule of thumb, we note that the non-clustered B-Tree index has a dense index dictionary, whereas in a clustered index the dictionary is non-dense. The third part in the index structure is known as the *data pages*, which hold the data records and is organized as in a sequential file. Figure 4.2. gives us a rude graphical representation of the B-Tree index structure, where we can figure out the three different parts which make up I/O cost estimation in a B-Tree index.

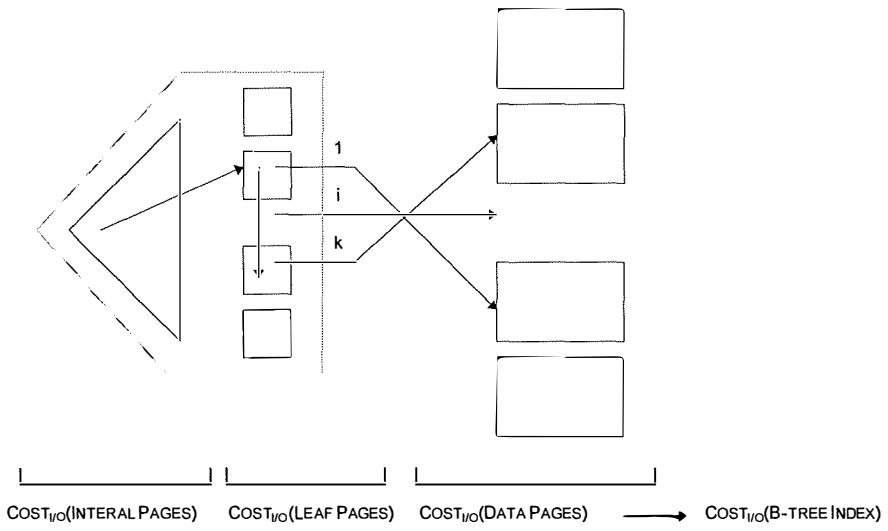


figure 4.2.: Cost Structure for B-Tree index access

The I/O cost estimation for of B-Tree indexes is given by the following relation:

$$\text{COST}_{I/O}(\text{B-TREE INDEX}) = \text{COST}_{I/O}(\text{INTERNAL PAGES}) + \text{COST}_{I/O}(\text{LEAF PAGES}) + \text{COST}_{I/O}(\text{DATA PAGES})$$

relation 4.12.: Cost estimation for an B-Tree index scan

In our cost estimation model we assume that the cost estimation for traversing internal pages, $COST_{I/O}$(INTERNAL PAGES), does not include the access to the first page of the index dictionary.

Before we give relations to estimate the I/O cost for the three components of the B-Tree index scan, we recall the notion of *fanout* and *depth* of a B-Tree. The fanout is the expected number of records that appear in an index (nodes) page. In section 3.2..2., we already made the assumption that the leaf pages and the internal pages will share the same structure, therefore they hold the same number of index entities. The fanout of a B-Tree index is given by the following relation:

$$fo = \left\lfloor \frac{(Ps - Hs) * fr}{As + Pps} \right\rfloor$$

where
Ps: Page size
Hs: Header size
Pps: Page pointers size
fr: Page fullness rate
As: Indexed attribute size

relation 4.13.: Fanout of an index node page

Knowing the index page fanout and the number of values to be indexed, the following relation can be used to estimate the depth of a B-Tree index, in other words the number of index levels needed for a given number of records. Looking back at the definition of the B-Tree index structure, we assumed that at leaf page level only holds as much records as their are number of distinct values, Ndv, for the indexed attribute.

$$d = \lceil \log_{fo}(Ndv) \rceil$$

where
fo: Fanout of index node pages
Ndv: Number of distinct records for the indexed attribute

relation 4.14.: Depth of a B-Tree

Recall that the difference between a non-dense (non-clustered index) and a dense (clustered) index, at leaf page level, is that the non-dense index does not hold all distinct values for the indexed attribute, but holds only a table reference, RID pointer, per table page. Thus, the number of distinct attribute values, Ndv, has to be replaced by the number of data pages, Np.

4.2.1.1. Internal Page I/O Cost Estimation

Making the assumption that in *the root level is held in most cases within the buffer*. Knowing that each traversed index level represents a disk I/O. Recall that the cost for internal page navigation does not include the access to the index dictionary. Relation 4.15. estimates the number of access need for navigating through the internal pages.

$$COST_{I/O}(\text{INTERNAL PAGES}) = (\lceil Log_{fo}(Ndv) \rceil - 2) * R$$

where
fo: Fanout of index node pages
Ndv: Number of distinct records for the indexed attribute
R: time fraction of time necessary to perform a random read (1/40 sec)

relation 4.15.: Cost estimation for navigating though the Internal Index Levels

4.2.1.2. Leaf Page I/O Cost Estimation

The estimation of the I/O costs for navigating within the leaf pages depends on the number of *distinct records values to be retrieved*, k_{Ndv}, and the *fanout,* fo, of an index page. Within the leaf page navigation, we have to differentiate between two types of leaf page navigation. The *first type* is where the index is defined on a *key attribute*, in other words the attribute values are unique. In different writings the key index is also referenced as being the primary key. Considering the filter factor and the fanout, the next relation estimates the I/O cost for this *first type* of leaf page navigation.

$$COST_{I/O}(\text{LEAF PAGES KEY ATTRIBUTE}) = 1 * R + (\lfloor k_{Ndv} / fo \rfloor) * S$$

where
fo: Fanout of index node pages
k_{Ndv}: Number of distinct records values to be retrieved
R: fraction of time necessary to perform a random read (1/40 sec)
S: fraction of time necessary to perform a mutli-block access read (1/400 sec)

relation 4.16.: Cost estimation for navigating though the Leaf Pages of a Key Index

The *second type* is where the index is defined on a *non-unique attribute, non-key attribute*. In different writings the non-key index is also referenced as being the secondary key. The multi-valued character of the attribute can be materialized by an extra level of indirection (section 32.2.4.). Recall, that the leaf page records refers to a page which contains all qualified RID pointers, Nrid, for a given attribute value. *It is common to consider, that a page of indirection can hold up to 254 RID pointers* [O'Neil 1994]. Each page of indirection, which has to be consulted, needs a disk access. Given these considerations and the fact that k_{Ndv} record values are to be retrieved, relation 4.17. defines the cost estimation for the secondary key leaf page navigation.

Note that we will use relation 4.17. as a general form of cost estimation for navigating through index leaf pages, though estimations for primary key indexes is wrong. Note, that for primary key indexes cost estimation is incorrect up to, a maximum of 1 random read, due to the second member of the sum, which is maximized by 1 in case of a key attribute ($\lceil k / 254 \rceil = \lceil 1 / 254 \rceil = 1$).

$$COST_{I/O}(\text{LEAF PAGES NON-KEY ATTRIBUTE}) = 1 * R + \lfloor k_{Ndv} / fo \rfloor * S + \lceil k / Nrid \rceil * R$$
$$= (1 + \lceil k / Nrid \rceil) * R + \lfloor k_{Ndv} / fo \rfloor * S$$

where
Nrid: number of RID pointers hold in a page of indirection (rude approximation 254)
fo: fanout of index node pages
k: number of records to be retrieved
k_{Ndv}: number of distinct records values to be retrieved
R: fraction of time necessary to perform a random read (1/40 sec)
S: fraction of time necessary to perform a mutli-block access read (1/400 sec)

relation 4.17.: Cost estimation for navigating though the Leaf Pages of a Non-Key Index

4.2.1.3. Data Page I/O Cost Estimation

Until now, we defined relations to estimate I/O cost which are related to the index navigation, but we still did not estimated the cost for retrieving data from the table. Thus, we have estimate the number of accesses needed to retrieve the data records, in other words the number of data pages to be consulted. Recall that relation 4.6. gives us an approximation of the number of pages which probably have to be retrieved from the table. This estimation is based upon probabilities. Let us abstract from the notion of probability that a data page contains at least one of the qualified records, and assume that the records are randomly distributed over the set of data pages, the *maximum number of pages* to be retrieved is the following.

$$^{MAX}COST_{I/O}(\text{DATA PAGES}) = k * \beta$$

where
k: number of records to be retrieved
β: time fraction R or L depending on whether we use Random or List prefetch

relation 4.18.: Pessimistic Cost Estimation for Data Pages Access

Considering the above named probability, we are able to define a more optimistic cost relation. Note however, that the best I/O cost is attained, when all qualified records are collocated on consecutive pages, because at this moment we are able to perform a multi-block access. The I/O cost, for such situations, is equal to $\lceil k/Nrp \rceil$, where k is the number of records retrieved and Nrp defines the number of records per data page (section 3.1.5). Situations like these are not very common in real world production, thus we define a more realistic relation, using the above named record page probability, to *estimate the best I/O cost*.

$$^{MIN}COST_{I/O}(\text{DATA PAGES}) = K * \beta$$

with

$$K = \lceil (1 - (1-ff)^{Nrp}) * Np \rceil$$

where
Nrp: number of records per data page
Np: number of pages for the table
ff: selectivity
k: number of records to be retrieved
β: time fraction R or L depending on whether we use Random or List prefetch

relation 4.19.: Optimistic Cost Estimation for Data Pages Access

Note that for B-Tree cost estimation we defined a *pessimistic* or *worst I/O cost estimation*, using relation 4.18, and an o*ptimistic* or *best I/O cost estimation*, using relation 4.19.

4.2.1.4. Global B-Tree I/O Cost Estimation

Grouping all three parts of the B-Tree cost estimation, we deduce the two following relations, one for pessimistic cost estimation and one for optimistic cost estimation of a non-clustered B-Tree index (B-TREE_{NC}). Relation 4.20. and 4.21. will help us introducing and analyzing the concept of ***index indecision***. The index indecision problem can be seen as being the zone, during index placement, where we are not able to state that an index improves performance over a brute force table scan.

$$^{MAX}\text{COST}_{I/O}(\text{B-TREE}_{NC}) = (\lceil \text{Log}_{fo}(Ndv) \rceil - 2) * R + (1 + \lceil k / Nrid \rceil) * R + \lfloor k_{Ndv} / fo \rfloor * S + k * \beta$$

$$= (\lceil (\text{Log}_{fo}(Ndv) - 1) + (k / Nrid) \rceil) * R + \lfloor k_{Ndv} / fo \rfloor * S + k * \beta$$

$$^{MIN}\text{COST}_{I/O}(\text{B-TREE}_{NC}) = (\lceil \text{Log}_{fo}(Ndv) \rceil - 2) * R + (1 + \lceil k / Nrid \rceil) * R + \lfloor k_{Ndv} / fo \rfloor * S + K * \beta$$

$$= (\lceil (\text{Log}_{fo}(Nrdv - 1) + (k / Nrid) \rceil) * R + \lfloor k_{Ndv} / fo \rfloor * S + K * \beta$$

where

fo:	Fanout of index node pages
Ndv:	Number of distinct records for the indexed attribute
Nrid:	number of RID pointer hold in a page of indirection (rude approximation 254)
k:	number of records to be retrieved
k_{Ndv}:	number of distinct records values to be retrieved
R:	fraction of time necessary to perform a random read (1/40 sec)
S:	fraction of time necessary to perform a mutli-block access read (1/400 sec)
β :	time fraction R or L depending on whether we use Random or List prefetch

relation 4.20.: Worst and Best I/O Cost estimation using Non-clustered B-Tree

Until here, we considered the situation of a non-clustering index, which implies a dense index dictionary. During the next lines we will consider the situation where the index is *clustered* (B-TREE_{c}). Recall that a clustered index is said to be non-dense or spare and that it forces a physical order upon the records location within the data pages. We can look at that data pages as being similar to the index dictionary on a non-clustered index, all qualified records are collocated within consecutive pages in the table file. This enables us to use mutli-block access for data page retrieval. As the index dictionary is non-dense, each leaf page includes only one index record per data page. This implies that the number of index levels is probably smaller than the number of index levels in a non-clustered index, but not always the case, because it depends on the number of data pages and the number of distinct values to be indexed. However, we can consider, without no great harm, that the access using a secondary index, costs more than the access using a primary index, because of the extra level of indirection. Knowing that a clustered index has a similar structure than a non-clustered and that the number of distinct records values, Ndv, in relation 4.20., is replaced by the number of data pages, Np, we defined relation 4.21. which defines an optimistic and pessimistic cost estimation for clustered B-Tree indexes.

$$COST_{I/O}(B\text{-}TREE_C) = (\lceil Log_{fo}(Np) \rceil - 2 + 1) * R + 1 * R + \lfloor k / Nrp \rfloor * S$$

$$= \lceil Log_{fo}(Np) \rceil * R + \lfloor k / Nrp \rfloor * S$$

where
fo: fanout of index node pages
k: number of records to be retrieved
Np: number of data pages
Nrp: number of records per data page
R: fraction of time necessary to perform a random read (1/40 sec)
S: fraction of time necessary to perform a mutli-block access read (1/400 sec)

relation 4.21.: I/O Cost estimation using Clustered B-Tree

Note, that in case of a clustered index, we no more make the difference between an optimistic and pessimistic cost estimation, as the random or list prefetch access to the data pages is replaced by a mutli-block access to the data.

4.2.2. B-Tree I/O Cost Estimation and Query Types

Note that we defined relation 4.20 and 4.21 by abstracting them to all kind of query types. However, we made a distinction on the physical ordering of the data records by defining a relation for a clustered and a non-clustered access structure. As we apply different query types to these relation, we will see that they simplify.

First, let us se how the relations simplify when they are related to a *point query*. Recall that a point query qualifies one and only one record, so $k = 1$, most of the time this happens when the index is defined on unique values, key attribute(s). As we assumed that a single record can not split over data page, the number of pages that hold the single qualified records is equal to $K = 1$. Note that there is no more a need to differentiate between an optimistic and pessimistic cost estimation in relation 4.20. The fact that only one record is retrieved by the query simplifies some of the terms of relation 4.20; the mutli-block access at index leaf page level becomes $\lfloor k_{Ndv} / fo \rfloor = 0$ and the fact that most of the time the point query is based on the uniqueness of the index values, implies that no extra level of indirection is needed to guide the query, $\lceil k / Nrid \rceil = 0$. The relation 4.20 becomes simplified as follows for a non-clustered index related to a point query:

$$COST_{I/O}(B\text{-}TREE_{NC}) = (\lceil Log_{fo}(Ndv) \rceil - 2) * R + 1 * R + 0 * S + 1 * R$$

$$= \lceil Log_{fo}(Ndv) \rceil * R$$

where
fo: fanout of index node pages
Ndv: number of distinct records for the indexed attribute
S: fraction of time necessary to perform a mutli-block access read (1/400 sec)
R: fraction of time necessary to perform a random read (1/40 sec)

relation 4.22.a.: I/O Cost estimation for Non-clustered B-Tree related to a Point Query

For a clustered B-Tree index the above consideration also remains true, except that the mutli-block access takes place at data page level and not at index leaf page level, so expression $\lfloor k / Nrp \rfloor$ becomes equal to zero, and relation 4.21 can be simplified as follows when related to a point query:

$$COST_{I/O}(B\text{-}TREE_c) = (\lceil Log_{fo}(Np) \rceil - 2 + 1) * R + 1 * R + 0 * S$$

$$= \lceil Log_{fo}(Np) \rceil * R$$

where
fo: fanout of index node pages
Np: number of data pages
S: fraction of time necessary to perform a mutli-block access read (1/400 sec)
R: fraction of time necessary to perform a random read (1/40 sec)

relation 4.22.b.: I/O Cost estimation using Clustered B-Tree related to a Point Query

Second, let us see how relation 4.20. and 4.21. simplify when related to *a multi-record queries*. Recall that a multi-record query qualifies more than one record in the set of data records. The fact that more than one record is qualified by the query includes the case where the index is defined on non unique values, non-key attribute(s) and that the qualified records are spread over more than one page.

In case where the index is defined on *non-key attribute(s),* the index navigation needs an extra level of indirection to access the qualified data pages. The fact that more than one page can hold the qualified records involves that we fall back in the situation of an optimistic and pessimistic cost estimation. When putting all these things together, relation 4.20. gives us simplified estimations for multi-record retrieval.

$$^{MAX}COST_{I/O}(B\text{-}TREE_{NC}) = (\lceil (Log_{fo}(Ndv) - 1) + (k / Nrid) \rceil) * R + \lfloor k_{Ndv} / fo \rfloor * S + k * \beta$$

$$^{MIN}COST_{I/O}(B\text{-}TREE_{NC}) = (\lceil (Log_{fo}(Ndv) - 1) + (k / Nrid) \rceil) * R + \lfloor k_{Ndv} / fo \rfloor * S + K * \beta$$

where
fo: Fanout of index node pages
Ndv: Number of distinct records for the indexed attribute
Nrid: number of RID pointer hold in a page of indirection (rude approximation 254)
k: number of records to be retrieved
k_{Ndv}: number of distinct record values to be retrieved
R: fraction of time necessary to perform a random read (1/40 sec)
S: fraction of time necessary to perform a mutli-block access read (1/400 sec)
β : fraction R or L, depending on whether we use random or list prefetch reads

relation 4.23.a.: I/O Cost estimation usingr Non-clustered B-Tree related to a Multirecord Query on a secondary index

In case where the index is defined on a *key attribute(s),* the navigation through the extra level of indirection falls apart, $\lceil k / Nrid \rceil = 0$. Again the fact that more than one page can hold the qualified records involves that the situation of an optimistic and pessimistic cost estimation. Relation 4.20. in accordance with a Multirecord query and a primary index simplifies as follows:

$$^{MAX}Cost_{I/O}(B\text{-}Tree_{NC}) = (\lceil(Log_{fo}(Ndv) - 1)\rceil) * R + \lfloor k_{Ndv} / fo \rfloor * S + k * \beta$$

$$^{MIN}Cost_{I/O}(B\text{-}Tree_{NC}) = (\lceil(Log_{fo}(Ndv) - 1)\rceil) * R + \lfloor k_{Ndv} / fo \rfloor * S + K * \beta$$

where
fo: Fanout of index node pages
Ndv: Number of distinct records for the indexed attribute
k: number of records to be retrieved
k_{Ndv}: number of distinct records values to be retrieved
R: fraction of time necessary to perform a random read (1/40 sec)
S: fraction of time necessary to perform a mutli-block access read (1/400 sec)
β : fraction R or L, depending on whether we use random or list prefetch reads

relation 4.23.b.: I/O Cost estimation using Non-clustered B-Tree related to a Multirecord Query on a primary index

In case of a clustered index, relation 4.21. does not simplify when related to a multi-record query. Note that it does not matter if we consider a primary or secondary index key type.

$$Cost_{I/O}(B\text{-}Tree_{c}) = \lceil Log_{fo}(Np) \rceil * R + \lfloor k / Nrp \rfloor * S$$

where
fo: fanout of index node pages
k: number of records to be retrieved
Np: number of data pages
Nrp: number of records per data page
R: fraction of time necessary to perform a random read (1/40 sec)
S: fraction of time necessary to perform a mutli-block access read (1/400 sec)

relation 4.23.c.: I/O Cost estimation using Clustered B-Tree related to a Multi-record query

Third, let us see what happens during *join queries.* To evaluate the I/O cost we have to go back and look at section 2.1.8. and section 4.1., where the cost estimation is done for the outer and inner table based on one of the above explained relations.

To resume the cost estimations for a query using B-Tree indexes as access structure we establish the following table resume.

	Non-Clustered B-Tree	**Clustered B-Tree**
Point query or Key attribute(s)	$\lceil Log_{fo}(Nr) \rceil * R$	$\lceil Log_{fo}(Np) \rceil * R$
Multi-record query on Non-Key and Key attribute(s)	***Pessimistic estimation*** *Non-Key attribute* $\lceil(Log_{fo}(Ndv) - 1 + (k / Nrid)\rceil * R$ $+ \lfloor k_{Ndv} / fo \rfloor * S + K * \beta$ *Key attribute* $\lceil(Log_{fo}(Ndv) - 1)\rceil * R + \lfloor k_{Ndv} / fo \rfloor * S + K * \beta$ ***Optimistic Estimation*** *Non-Key attribute* $\lceil(Log_{fo}(Ndv) - 1 + (k / Nrid)\rceil * R$	$\lceil Log_{fo}(Np) \rceil * R + \lfloor k / Nrp \rfloor * S$

	$+\lfloor k_{Ndv} / fo \rfloor * S + k * \beta$ *Key attribute* $\lceil (Log_{fo} (Ndv) - 1) \rceil * R + \lfloor k_{Ndv} / fo \rfloor * S + k * \beta$	

4.2.3. Hash index I/O Cost estimations

Based upon a made by [Hainaut 1994] we are able to give a rude approximation for hashed access costs. As we already seen in section 3.2. the hash index is extremely beneficiary for point queries, for example queries involving equality predicates, as in most cases the retrieval does not include the search within the collision chain. It is even more efficient when it is used with a key attribute (unique values). Thus throughout this section we will not make cost estimations for the different query types.

Recall that in section 3.2.4 we already described the notion of hash indexes. We have seen, that there exists multiple techniques to solve the problem of hash collisions. One of the techniques is the known as open addressing, a second as chaining and a third as multiple hashing. We also stated that access costs increase the more a page fills up, as the collision chains get longer. We might be surprising, is how quickly the collision chains increases in length once the page fill rate comes close to its maximum.

Within this section we do not give estimations for all collision chain techniques, but we will try to define a relation and to illustrate technique of chaining [Hainaut 1994]. Note that in [Hainaut 1994] this technique is referenced as independent zone chaining. This means, that there exists two types of chaining, the primary zone chaining and the independent zone chaining. The difference is that the primary zone chaining tries first to store the new added record within the referenced page, whereas the independent zone chaining directly goes on a secondary page.

Assume that the records are randomly distributed among the set of data pages, and that the DBMS uses the technique of independent chaining to solve the problem of hash collisions.

The probability that a given page holds k records can be *Binomial* or *Poisson.*

Binomial: $P(k) = C_{Nr}^{k} * \left(\frac{1}{Np}\right)^{k} * \left(1 - \frac{1}{Np}\right)^{Nr-k}$ note: $C_N^n = \frac{N!}{n!(N-n)!}$

Poisson: $P(k) = Nrp^{k} * \frac{1}{k!} * e^{-Nrp}$

with
k : Number of qualified records
Nr: Number of records in the table
Np: Number of pages
Nrp: Number of records per page

According to these probabilities, we define the number of pages that hold the k records, n(k), the number of pages that are empty n(0), as well as the number of pages that are not empty, n(>1):

$$n(k) = Np * P(k)$$

$$n(0) = Np * P(0)$$

$$n(> 1) = Np * (1 - P(0))$$

The probability that a page overflows, P_{over}, is equal to:

$$P_{over} = \sum_{k=Nrp+1}^{Nr} P(k)$$

Given the probability that a page overflows, we define the number of pages that have overflows, Np_{over}, as well as the number of pages that have no overflow, $Np_{no\text{-}over}$:

$$Np_{over} = Np * P_{over}$$

$$Np_{no-over} = Np * (1 - P_{over})$$

To estimate the I/O cost for a hash access, we have to estimate the number of records within primary pages, Nr_{prim}, and the number of records within independent pages, Nr_{indep},.

$$Nr_{prim} = Np * m_{prim}$$

$$Nr_{indep} = Nr - Nr_{prim}$$

with

$$m_{prim} = \sum_{k=1}^{Nrp} k * P(k) + Nrp * \sum_{k=Nrp+1}^{Nr} P(k) : \text{average number of records per primary page}$$

The probability that a record is within independent pages is equal to

$$P_{indep} = \frac{Nr_{indep}}{Nr},$$

and the average length of a the collision chain is given by,

$$E(L) = \frac{Nr_{indep}}{Np_{over}} = \frac{Nrp - Nr_{prim}}{p_{over}}.$$

The average cost for accessing records, with success, has to be splited into two terms, one for the access costs within primary pages and a second for access costs within independent pages:

primary pages: $Cost^{succ}(Hash_{prim}) = \frac{Nr_{prim}}{Nr} * 1R$

independent pages: $Cost^{succ}(Hash_{indep}) = \frac{Nr_{indep}}{Nr} * (1 + \frac{1 + E(L)}{2})R$

All together the average I/O cost for accessing records using a hash index is estimated by the following relation:

$$Cost^{succ}(Hash) = Cost^{succ}(Hash_{prim}) + Cost^{succ}(Hash_{indep})$$

$$Cost^{succ}(Hash) = \left(\frac{Nr_{prim}}{Nr} + \frac{Nr_{indep}}{Nr} * \left(1 + \frac{1+E(L)}{2}\right)\right)R = \left(1 + P_{indep} * \frac{1+E(L)}{2}\right)R$$

where

Nr_{prim}:	number of records in primary pages
Nr_{indep}:	number of records in independent pages
Nr:	number of records
E(L):	average length of the collision chain
P_{indep}:	probability that a page overflows
R:	fraction of time necessary to perform a random read (1/40 sec)

relation 4.24.: I/O Cost estimation, with success, for a Hash index using Independent Chaining.

Same as for the access cost, with success, we define the access cost in case were no record is qualified by the predicate.

primary pages: $Cost^{no-succ}(Hash_{prim}) = (1 - P_{over}) * 1R$

independent pages: $Cost^{no-succ}(Hash_{indep}) = P_{over} * (1 + E(L))R$

All together, the average I/O cost for accessing records using a hash index is estimated by the following relation:

$$Cost^{no-succ}(Hash) = Cost^{no-succ}(Hash_{prim}) + Cost^{no-succ}(Hash_{indep})$$

$$Cost^{no-succ}(Hash) = (1 + P_{indep} * E(L))R$$

where

E(L):	average length of the collision chain
P_{indep}:	probability that a page overflows
R:	fraction of time necessary to perform a random read (1/40 sec)

relation 4.25..: I/O Cost estimation, with no success, for a Hash index using Independent Chaining.

To illustrate the relations for hash index cost estimation, let us consider a small example.

Consider the following:

- Number of records NR: 500.000 records
- Record size, RS: 200 bytes
- Page size, PS: 2000 bytes (note that header space is not included)
- Fill rate, fr: 90%
- Number of qualified records, k: 10 records

Assume that the probability that a given page holds the k records is of Poisson. Assume also that the number of records is fix, thus the fill rate can be considered as fix.

According to this input we can determine the number of records that can hold within on page, Nrp, and the number of pages, Np:

$$Nrp = \left\lfloor \frac{2000 * 0.9}{200} \right\rfloor = 9 \text{ records/page} \qquad Np = \left\lceil \frac{500000}{9} \right\rceil = 55.556 \text{ pages}$$

Knowing the number of records per page and the number of records qualified, we determine the probability that the k records are within one page.

$$P(k) = 9^{10} * \frac{1}{10!} * e^{-9} = 0.1185$$

Assume that the average number of records per primary page, m_{prim}, tends to 8. Now we can determine the number of records in primary pages, Nr_{prim}, and the number of records within independent pages, Nr_{indep}, are equal to:

$Nr_{prim} = 55.556 * 8 = 444.448$ records $\qquad Nr_{indep} = 500.000 - 444.448 = 55.552$ records

Thus, the probability that a record is within independent pages is equal to:

$$P_{indep} = \frac{55.552}{500.000} = 0.11$$

We still need to determine the average length the collision chain, to compute an approximation of the access cost.

The probability that a page overflows tends to:

$P_{over} = 0.41$

Given this probability we determine the number of pages that have overflows, n_{over}, as well as the number of pages that have no overflow, $n_{no\text{-}over}$:

$Np_{over} = 55.556 * 0.41 = 22.778$ records $\qquad Np_{no-over} = 55.556 * (1 - 0.41) = 32..778$ records

Knowing the number of pages with overflow we determine the average length of the collisions chain.

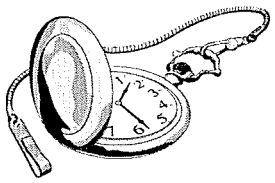
$$E(L) = \frac{55.552}{22.778} = 2{,}43$$

Thus, the access cost for a success, using a hash index, in terms of I/O is equal to:

$$Cost^{succ}(Hash) = (1 + P_{indep} * \frac{1 + E(L)}{2}) = (1 + 0.11 * \frac{1 + 2.43}{2}) = 1.18 = 2 \text{ accesses}$$

If we consider that the accesses are random, we can determine the following time used to access data:

$Cost^{succ}(Hash) = 2 / 40 = 0.05$ sec



Chapter 5. Index Selection

In this chapter, we describe a methodology of index selection based on a study made by [Finkelstein 1988]. Methodologies for index selection are based on models of data retrieval and updates, as the one we defined in preceding chapters. Some solve the problem in an analytic way. Whereas, others use heuristic searches to find a quasi optimal solution. Since we assumed that the database management system uses an optimizer to choose an access strategy. It makes sense to use the optimizer itself to provide the estimated I/O costs given SQL statement, when the database already exists. However, if we are only at the phase of database conception, then we have to use the cost relations defined in chapter 4. The estimations use general forms of cost estimations, independent relational DBMSs, however, it is always a good thing to certify our index solution by using optimizer cost estimation when the database is implemented. The optimizer examines the set of access structures that exist and computes the best expected cost for a statement by evaluating different join orders, join methods, and access choices.

Using the optimizer, we might guarantee that any retained solution is one the optimizer might use to its full advantage. Working with an external model, might result in a solution that has good theoretical performance. However, when the optimizer is faced with the set of indexes, it may choose an execution plan different from the one predicated by the model, which may result in poor performance.

To describe the methodology for index selection, we reference to the design tool, DBDSGN, defined by [Finkelstein 1988]. This design tool has five principal steps. Figure 5.1. represents the steps and the tools major interactions with the database designer and the DBMS.

The design tool as well as the DBA can interact with the DBMS to collect information without physically running a statement by using the SQL EXPLAIN facility. EXPLAIN causes the optimizer to choose an execution plan, including indexes, for a given statement and stores the information into an internal tables. These tables can be accessed and abstracted using ordinary queries.

The EXPLAIN command has four options:

- EXPLAIN REFERENCE. Identifies the statement type (Query, Update, Delete, Insert). The tables and the attributes referenced in the statement in ways that influence their plausibility for indexing.
- EXPLAIN STRUCTURE. Identifies the structure of the subquery tree in the statement. The estimated number of record qualified by the statement and its subqueries. And the estimated number of time the statement and its subqueries are executed.

- EXPLAIN COST. Indicates the estimated cost of execution of the statement and its subqueries in the plan chosen by the optimizer.
- EXPLAIN PLAN. Describes aspects of the access plan chosen by the optimizer. Including the order in which tables are accessed for executing the statement, the indexes used, the methods used to perform joins (nested loop, merge or hybrid), and the sorts performed.

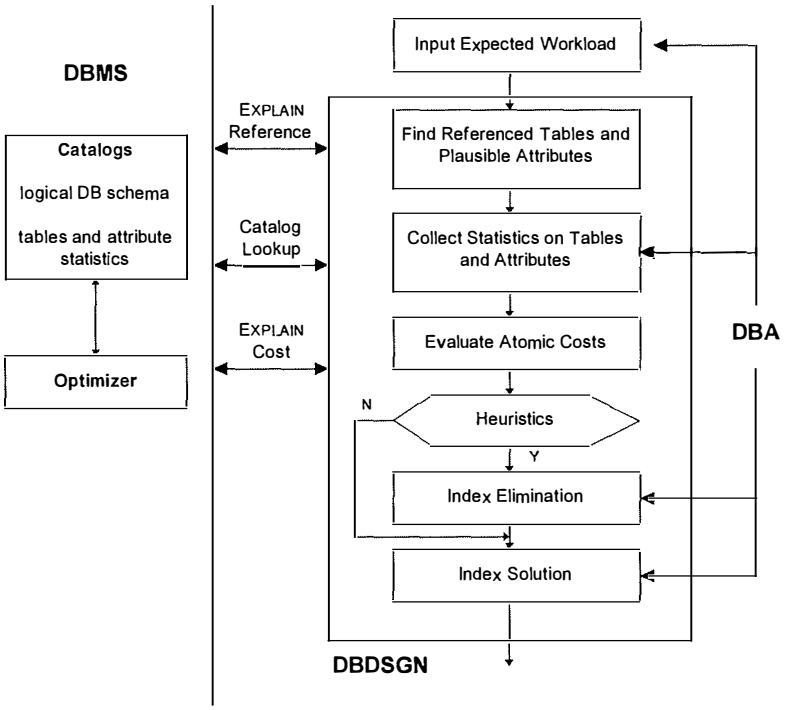


figure 5.1.: Architecture of [Finkelstein 1988]s DBDSGN Tool

- The first step of the methodology involves the *identification of referenced tables and plausible attributes.* Based on an analysis of the structure of the input statements, we should only allow attributes that are *'plausible for indexing'* to enter the index solution set.
- Second step consists in *collecting statistics on tables and attributes.* Statistics are either provided by the database designer or extracted from database register.
- During the third step *I/O costs are evaluated* using relations defined in chapter 4 and/or the Explain facility.
- Fourth step is called the *index elimination.* If the problem is large, a heuristic-based dominance criterion can be invoked to eliminate indexes and to reduce the search for the solution during last step.

- Fifth step consists in *generating an index solution*. A controlled search of the set of intermediate index configurations leads to the discovery of a good index solution. The database designer uses its knowledge to control and to guide the search.

5.1. Setting up a Cost Model

5.1.1. Workload Model

When a database designer is asked to supply an index design solution for a given database, he must determine the workload that is expected for that system over a specified time period. The expected workload during that time period is characterized by a set of pairs

$W = \{(q_i, w_i), I = 1, 2, .., q\}$,

where
each q_i is an SQL statement and
w_i is its assigned weight.

The SQL statements are queries on single tables and multi-table joins as well as updates, inserts and deletes. The q_i are statements that the designer expects to be relatively important during database implementation and running. The statements in the workload W may initiate from different sources:

- predictable ad hoc statements that will be issued from terminals,
- old application, or
- new application programs that will be executed during the database live or a given time period

The weight associated to each of the statements is a function of

- the *frequency of execution* of the statement during a given period, and/or
- the *system load* when the statement is executed. Statements that can be run off-line may be given a smaller weight than critical statements which require fast response time.

Different statements that are treated identically by the optimizer could be combined, although this requires deep knowledge of the optimizers internal architecture. For example, consider the query with predicate CLIENTID = 123, we are able to combine this query with the one that uses predicate CLIENTID = 456 since they have the same filter factor, ff. Either query might be included in the workload, with the sum of the original weights specified. However, a query with predicate CLIENTID BETWEEN 123 AND 456, could not be combined with one requesting CLIENTID BETWEEN 10 AND 20, since they associate different filter factors.

For application programs, the search for the weights is quiet a difficult problem. In general, frequencies must be approximated. Designers may perhaps know how often an application will be run, but may find it difficult to predict the frequency of execution of a statement due to the complexity of the program logic.

5.1.2. Atomic Costs

This section will give us another glance at the behavior of the database optimizer. We should be aware that some principles are to be verified to use and/or to predict the optimizer behavior. It is not our aim in this section to describe how the optimizer makes its decisions, but to help understand the principles vehiculated by the optimizer. The basic principles used by the optimizer in executing a given statement are as follows:

Optimizer principles

- **P1**: Exactly one index is used for each appearance of a table in the statement.
- **P2**: The costs of all combinations using one index per table appearance are computed, and the one with the minimal cost is chosen.

Principle, **P1**, would not be true of a system that used conjunction of indexes on a single table, such as RID intersection, which we do not assume to exist for our considerations. Principle, **P2**, might not be true for an optimizer that uses heuristics to limit its search for the plan with the smallest expected execution cost. We can slightly relax principle, **P2**, as it is not necessary for the optimizer to compute all possible costs, as long as it finds the plan with the smallest expected cost.

The cost of executing a statement can consist of three components: *record access cost, record maintenance, update cost and index maintenance cost*. Throughout the following lines we will only consider access costs, update costs will be aborted in a next section.

To illustrate the above principles, let us consider a statement on a single table that has n indexes. The optimizer computes at least n+1 access costs, n using each single index, and 1 using brute force scan, an chooses the index with the minimal cost. The access costs are computed independently, since the presence of a given index cannot influence the cost computation of accessing the records by using another index (principal, P1, only one index per table can be used).

Now consider a t-table join statement, q, with I_j a set of indexes on the j^{th} table. Let $C_q(\alpha_1, \alpha_2, ..., \alpha_t)$ be the optimizers best cost execution plan of q when the indexes $\alpha_1, \alpha_2, ..., \alpha_t$ are used, where α_i is either one of the indexes in I_j or the brute force scan ρ. The tables may be accessed in many different orders, and many join methods (section 2.1.8.) are possible even when the indexes are fixed. Considering the optimizer principles, we can think of the optimizer as if it computes each $C_q(\alpha_1, \alpha_2, ..., \alpha_t)$ independently. The choice it selects for execution is one with the minimal estimated I/O cost, so we define:

$$COST_q(I_1, I_2, ..., I_t) = \min_{\alpha j \in Ij \cup \{\rho\}} C_q(\alpha_1, \alpha_2, ..., \alpha_t)$$

$COST_q(I_1, I_2, ..., I_t)$ represents the minimum I/O cost estimation which can be computed. Let ISET be a collection of indexes that exist on a set of tables. For this index configuration ISET, we write $COST_q$[ISET] to represent $COST_q(I_1, I_2, ..., I_t)$, where I_j is the set of indexes in ISET that are on the j^{th} table. Indexes in ISET on tables not referenced in the statement do not affect $COST_q$[ISET]. For a single table statement against a table with n attributes, we can build $n2^{n-1} + 2^n$ different index configurations. There are n clustered choices, and for each of these, there are 2^{n-1} different non-clustered sets. If no clustered index is chosen, there are 2^n sets of non-clustered indexes. For a join query, the number of configurations is the product of the number of configurations for each table. Which is exponential in the total number of attributes in the table.

Configurations with at most one index per table are *called atomic configurations*, and their costs are called *atomic costs*, since costs for all other configurations can be computed from them[51]. Atomic configurations for a table(s), are atomic configurations where indexes are only on that (those) table(s). Atomic configurations for a given statement are configurations that are atomic for the tables in that statement.

Proposition 1:

The cost of a query, single table or join, for a configuration is the minimum of the costs for that query taken over the atomic configurations that are subsets of the configuration. More formally,

$$COST_q[ISET] = \min_{ASET \in ISET} COST_q[ASET]$$

where ASETs are atomic configurations

This proposition follows from the definition of $COST_q$. $COST_q$[ISET] is the minimum of the $C_q(\alpha_1, \alpha_2, ..., \alpha_t)$ values, where the α_s are indexes over appropriate tables, and any α can be the brute force scan. Similarly $COST_q$[ASET] replacing by its definition, $C_q(\alpha_1, \alpha_2, ..., \alpha_t)$ each appears in the right-hand side minimum at least once, and the C_q terms involving brute force scan appear more than once. Since both minimum are over the same set of C_q terms, they are equal, verifying the proposition.

Performing EXPLAIN COST only for atomic configurations significantly reduces the number of cost inquiries that have to be fulfilled. For a query on a table within attributes, there are 2n+1 atomic configurations, n with 1 clustered index, n with 1 non-clustered index, and the configuration with no index at all, so the number of cost estimations is reduced from exponential to linear in the number of attributes. For a t-table join, recall that the number of configurations is exponential in the total number of attributes in the joined tables. The number of atomic configurations for a join equals the product of the number of atomic configurations for each single table. That is, if we let n_j be the number of attributes in the j^{th} table of the join, there are $\Pi_{j=1}^{t} (2n_j + 1)$ atomic configurations for the join. Despite this significant reduction, the computation of all atomic costs may still be impractical for large n_j and t. Later on we describe methods to reduce the number of plausible indexes.

---

[51] Configurations with more than one index per table are admitted to evaluate statements with self-joins, when a table is joined on itself, but for simplicity we omit discussion of this case.

5.1.3. Update, Maintenance Costs

In section 2.2., we already discussed the problems of I/O cost estimations for update operations. In here we will not go in much more details, however we will see how we can integrate the cost estimations in the cost model.

Let us assume that update operations involve only one single table at the time. However, update statements may have subqueries, but the DBMS handles them separately from the root of the subquery tree on, just as it does when the root is a query on its own. During execution the update statements follow tree steps:

- Using some access path(s), the records acted upon are found or the locations for inserted records are found.
- The records are updated, deleted or inserted.
- If necessary, table indexes are updated.

The cost of index updates may be substantial, so we have to care about these costs when evaluating a physical design. Furthermore, as mentioned by [Finkelstein 1988], the update cost can not be considered as a constant for every index, because of two things.

- The update cost depends highly on the form of the statement, such as the predicates in the Where clause, and/or the contents of the Set clause for the update operations.
- Another distinction in cost estimation, must be made based on the way the records and indexes to be modified are accessed. In particular, the access path determines the order in which the records in the data pages are scanned. Different formulas apply based on whether or not these objects are scanned in the same order they are stored.

We separate the costs for update operations into two components:

- The cost of accessing and modifying records, and
- the cost of maintaining indexes on attributes that are affected by the statement.

The notion of atomic cost holds also for update statements. We will make a difference between the *atomic access costs*, which are defined as the sum of the costs of accessing and updating the qualified records. And the *atomic index update costs*. Fortunately, a small set of atomic index update costs determine the costs of update operations, for any set of indexes in the database. We must estimate the cost of updating any index, no matter what index is used to access the records. The important distinction is not which index is used, but whether the access index and updated index are ordered in the same way. When they are, we call this an *ordered scan*, however, when they are not , we call it an *unordered scan.* For example, for a clustered index the scan is ordered if the index is either the same, which often occurs for inserts and/or deletes, or brute force scan[52]; in this case, the modifications follow the order in which the RIDs are stored at index dictionary level. If the clustered index is updated following a scan on a non-clustered index instead, the RIDs may be hit in an unordered way, incurring a higher cost. For updating a non-clustered index, the only ordered scan is the index itself.

Let us assume that ISET is a set of indexes and q an update operation (Insert, Delete, Update). Since q can involve only one table, although subqueries can mention other tables, we assume without loss of generality that ISET is only on the modified table. Because of the optimizer principles, defined in section 5.1.2., the optimizer costs estimated for executing update operation q in configuration ISET is:

$$Cost_q[ISET] = \min_{\alpha \in ISET \cup \{\rho\}} \left[C_q(\alpha) + \Sigma_{\beta \in ISET}\ U_q(\beta,\alpha)\right]$$

where C_q is the I/O cost of accessing and modifying records using index α, and $U_q(\beta,\alpha)$ is the I/O cost of updating index β if index α is used as an access path to the table.

As with queries, indexes in ISET on tables not referenced in an update operation do not affect, $Cost_q[ISET]$, I/O cost. The definition of $Cost_q$ in the relation above here is consistent with the definition of $Cost_q$ in section 5.1.2. for single-table queries.

Let q be a statement on a single table, including updates, deletes, and inserts, as well as queries on a single table, and let $AP_q(ASET)$ be the index chosen by the optimizer to process statement q in atomic configuration ASET, which is either the brute force scan ρ or the one index in ASET that is on the referenced table. The following proposition decomposes the I/O cost of q for configuration ISET into the I/O costs C_q and U_q for atomic configurations ASET included in ISET.

Proposition 2:

The cost of a query, single table or join, for a configuration is the minimum of the costs for that query taken over the atomic configurations that are subsets of the configuration. More formally,

$$Cost'_q[ISET] = \min_{ASET \in ISET} \left[Cost_q[ISET] + \Sigma_{\beta \in ISET}\ U_q(\beta, AP_q(ASET))\right]$$

---

[52] Recall, when an indexed attribute in a given record is updeted, the RID pointer associated with the record is deleted from the index dictionnary following the old key value and inserted in the dictionnary of new key values. Accessing the tuples through an index that is currently updated may lead to hitting the same RID more than once.

where ASETs are atomic configurations. Then,

$$\text{COST}'_q[\text{ISET}] = \text{COST}_q[\text{ISET}]$$

The above proposition is proved by the following.

Let the n indexes in ISET be $\alpha_1, \alpha_2, \ldots, \alpha_n$. By definition, $\text{COST}_q[\text{ISET}]$ is the minimum of the following I/O costs:

$$c_0 = C_q(\rho) + \Sigma^n_{i=1} U_q(\alpha_i, \rho),$$

$$c_1 = C_q(\alpha_1) + \Sigma^n_{i=1} U_q(\alpha_i, \alpha_1),$$

. . .

$$c_n = C_q(\alpha_n) + \Sigma^n_{i=1} U_q(\alpha_i, \alpha_n).$$

And $\text{COST}'_q[\text{ISET}]$ is the minimum of the following I/O costs

$$c'_0 = \text{COST}_q[\{\rho\}] + \Sigma_{\beta \in \text{ISET}} U_q(\beta, \rho) = c_0$$

$$c'_1 = \text{COST}_q[\{\alpha_1\}] + \Sigma_{\beta \in \text{ISET} - \{\alpha 1\}} U_q(\beta, AP_q(\{\alpha_1\})) = \min(c_0, c_1),$$

. . .

$$c'_n = \text{COST}_q[\{\alpha_n\}] + \Sigma_{\beta \in \text{ISET} - \{\alpha n\}} U_q(\beta, AP_q(\{\alpha_n\})) = \min(c_0, c_n).$$

Hence, $\text{COST}'_q[\text{ISET}] = \min(c_0, c_1, \ldots, c_n)$, demonstrating the proposition.

As we mentioned earlier in this section, for an index β the maintenance cost $U_q(\beta, \alpha)$ depends on whether the access to β is an ordered or an unordered scan and is otherwise independent of α's attributes. This remains true even if α is ρ. Thus, there are only two costs to be computed for β. Let $U'_q(\beta)$ be the cost of updating β if α determines an ordered scan of β, and let $U''_q(\beta)$ be the cost of updating β if α determines an unordered scan of β. Then $U_q(\beta, \alpha)$ is either $U'_q(\beta)$ or $U''_q(\beta)$.

Performing I/O cost estimations, EXPLAIN COST, for atomic configurations, we collect the maintenance cost of a given index for both ordered and unordered scans. For example, assume q being an UPDATE statement. We want to evaluate the maintenance cost for index β. The atomic configurations with β that are of interest for q, depend on whether β is clustered or not. Performing EXPLAIN COST statement for q with β clustered, we get an I/O cost $C_q(\beta)$, for access and record maintenance, and an I/O cost $U_q(\beta, \rho)$, for updating the index. The only possible index for the optimizer is the brute force scan ρ, so $U_q(\beta, \rho) = U'_q(\beta)$ is the cost of maintaining the index β following an ordered scan. Similarly the configuration with β being a non-clustered index, gives us the cost $U''_q(\beta)$ of the unordered scan[53]. Similar considerations can be applied for DELETE and INSERT statements.

5.1.4. Plausible Attributes for Index Solution

[53] We assume here that the I/O cost of updating an index following an unordered scan is always the same, no matter what other index is chosen. In reality this is not always true, but we think it a reasonable approximation.

Estimating and/or performing optimizer cost calculations for atomic configurations reduces the set of cost estimations to entail. The following section describes an additional technique to reduce the set of cost estimations.

In general, the number of index candidates on a table equals twice the number of attributes on that table, because indexes may be clustered or non-clustered. However, not all attributes are plausible candidates for indexing. Attributes that appear in statements in ways that support the use of indexes are *called plausible attributes*, for the given statement. Other attributes belong to the set of *non-plausible attributes*. The consideration which allows us to determine the set of plausible attributes for each statement is optimizer dependent. The critical assumption is that, for the statement, non-plausible attributes must have the same costs for indexes, no matter what other indexes exist. For a relational system the considerations include the following:

- First, an attribute belong to the set of plausible attributes if there is a predicate on it and the system is able to use it to process the statement. This happens when the predicate, section 2.1.6.1., is ANDed to the rest of the WHERE clause, and it is usable as a search criterion to retrieve records through an indexed access. Let us recall a general form of the predicates that include plausible attributes. This happens when the predicate is of form attribute θ X, where θ is a comparison or range operator (>, >=, = ; =<, Between, In), and X is a constant, a program variable, or a referenced attribute in an other table.

  Let us illustrate this by using an example. We take the following table defining products:

  Prod (Prodno, Descrip, Suppno, Quality, Price, Qonord, Qonhand ...)

  an the following statement

  ```
  SELECT Prodno, Descrip
  FROM Prod
  WHERE Suppno = 274
  AND (Quality = 'High' or Price > 10000)
  AND Qonord = Qohand + 50
  ```

  For this statement, Suppno is a plausible attribute, whereas Quality and Price are not-plausible. Qonord is also non-plausible, because it is compared with the result of an expression.

- Second, an attribute that is not-plausible for indexing because of the selection predicate may however still be a plausible candidate for indexing for other reasons. For example, there may be a GROUPE BY or ORDER BY clause on that attribute. The optimizer could even decide that an attribute that does not appear at all within the statement is plausible. Moreover, an implausible attribute (index) might be a better access path than a brute force scan in certain cases. Since all indexes on implausible attributes have almost identical costs, a single implausible representative can be added to the plausible set of attributes.

- Third, if a table is not mentioned in a statement, all its attributes are not-plausible for that statement.

It remains, that plausible attributes may be unusable for a particular statement because of system constraints. For example, an attribute that is changed by an Update statement may not be usable even if it appears in an index-processable predicate. This does not mean that all plausible attributes, which are used in an Update statement, are to become not-plausible, we must weight the update frequency and the query frequency. If the query frequency gives us a higher advantage than the update cost than the attribute shall remain in the set of plausible attributes, otherwise it should be putted with the non-plausible attributes.

Limiting access-paths to the plausible access paths greatly reduces the number of cost estimations to be done. A configuration is plausible for a statement if all indexes in it are plausible for that statement. We will use the following criterion to limit the number of I/O cost estimations.

Costs are obtained for each statement only for plausible atomic configurations for a given statement.

The validity of this criterion is a consequence of proposition 1 and 2 of section 5.1.2. and 5.1.3..

Let us illustrate, the validity of plausibility in reducing the complexity of index-selection problem, by the following example. Consider, again, the product table, PROD, and an order table, ORDER, where each table has 10 attributes. Without the technique of plausibility we would have to consider $10*2^9 = 5.120$ configurations for each table with one index clustered and the others non-clustered, and $2^{10} = 1.024$ configurations with all indexes as being non-clustered, for a total of 6.144 configurations. For the two tables together, there are a total of $6.144^2 = 37.748.736$ configurations. Plausibility allows us to drastically reduce the number of configurations. Consider the following statement:

```
SELECT P.Prodno, P.Descrip
FROM PROD P, ORDER O
WHERE P.Prodno = O.Prodno
AND O.Suppno = 274
AND P.Quality = 'High'
AND P.Price BETWEEN 10.000 AND 40.000
```

Of the 20 attributes PROD and ORDER, only 5 are plausible for the given statement: Prodno, Quality and Price for PROD, and Prodno, Suppno for ORDERS. Hence, there are 160 plausible configurations on the two tables, and only 35 of them are atomic plausible configurations. Suppose that another statement in the same workload is defined:

```
SELECT *
FROM PROD P, ORDER O
WHERE P.Prodno = O.Prodno
AND O.Date = 19961506
AND P.Price < 30.000
```

All the attributes in PROD and ORDER, appear in the select list, but only four are plausible. For this statement there are 64 plausible configurations, of which 25 are atomic plausible configurations. 15 of those are also atomic plausible configurations for the preceding statement. The total number of different atomic plausible configurations for both statements is 45. In practical workloads many attributes in the database are not referenced, and some attributes are only referenced in the SELECT list and never in the WHERE clause. The plausible configurations for joins often intersect considerably; it is particularly common for several statements to have the same join attributes, because of hierarchical and network relationships that exist in the data tables. Furthermore, as we previously indicated, not all attributes referenced in the WHERE clause are plausible. Hence, performing index selection on the basis of the plausible configurations can be of help.

5.1.5. Atomic Costs Computation

In order to obtain solutions for index selection problem, the database designer needs to compute the costs of the statements for plausible atomic configurations.

We will say that an *index configuration is covered by another*, for a given statement q, when both configurations have the same indexes for all tables referenced in q. Same, a *set of configurations is said to covers another*, for statement q, when each configuration in the second set is covered for q by a configurations in the first set. A *set of configurations is minimal,* for a workload when it contains no configuration that is covered by the other configurations, for every statement in the workload. Since, the cost of a statement is independent of the indexes on tables that are not referenced by the statement. To determine all plausible atomic costs it is enough to simulate a (minimal) set of atomic configurations, that covers the set of plausible atomic configurations for the given workload.

Remember that we might get statistics and cost estimation out of database catalog tables. We must note, that during computation and simulations of index configurations the catalog updates can be very time consuming. Let us consider a statement on a single table. Since the same index may be plausible for more than one statement, the number of system catalog updates necessary to simulate the configurations equals the total number of different indexes that are plausible for at least one statement. Brute force scan must be counted once for each table. For a single table statement, catalog updates could be done efficiently on a table by table basis.

For a workload that includes joins, the number of catalog updates may be very high, since the number of atomic configurations to be simulated grows exponentially with the number of tables joined. We want to reduce the number of catalog updates by never simulating a configuration more than once, by simulating a minimal set of configurations for a workload, and by simulating configurations in sequence that reduces the number of catalog updates. In this section we briefly describe a simple procedure to enumerate, a cover for, the plausible atomic configurations so that the database designer obtains the plausible atomic costs for all statements in the workload.

For a statement q involving t_q tables, let NA_{iq} be the number of plausible indexes to the j^{th} table of the statement. The number of different atomic configurations to be simulated is

$$\prod_{I=1}^{t_q} NA_{iq}$$

Using Gray coding[54] we are able to enumerate atomic configurations, with table t as the highest order, the least frequently changing, attribute and table 1 as the lowest order, the most frequently changing, attribute. With regard to this the number of catalog updates becomes:

$$\psi_q = \sum_{j=1}^{t_q} \prod_{i=1}^{j} NA_{iq}$$

ψ_q is minimized by permuting the tables so that the NA_{iq} values are monotonically increased. Different tables may be used for different statements. $\psi = \Sigma_q \psi_q$ catalog updates are enough to compute costs for all statements. According to [Finkelstein 1988], in many cases the plausible configurations for joins intersect considerably with one each other. Thus performing the cost estimations independently for each join, risks to create identical configurations more than once. To avoid this situation, and hence reduce the number of catalog updates, whenever we simulate an atomic configuration, we compute the cost of each statement for which that configuration is plausible. More generally, the database designer estimates the cost for each statement such that the simulated configuration covers a plausible configuration. Ordering the statements, so that the ones with the largest number of tables are processed first, also reduces the number of possible configurations, since a join involving many tables may enumerate configurations needed by simpler statements.

Join cost computation rule

- The list of join statements is ordered in decreasing order, by the number of tables referenced;
- For each join q, all plausible atomic configurations are enumerated, using for example Gray coding[55], with the tables permuted so that the NA_{iq} values are increasing. A configuration is simulated only if the cost of q for that configuration has not yet been estimated.
- For each simulated configuration, a I/O cost estimation is performed for every join, after the current join in the join list, such that the simulated configuration covers an atomic configuration that is plausible for that join.

---

[54] According to [Baudoin-1984] a Gray code of order n is a permutation of 2^n words of n bits ordered a way that the i^{th} bit divers from the $i\text{-}1^{th}$ in one position only, the whole in a a cyclic way, taking i modulo n. The following is an example of order 3 Gray code:

000 001 011 010 110 111 101 100

[55] Any other enumeration technic generating each configuration once is acceptable.

5.2. Index Elimination

Above here we described the notion of plausible attributes for indexation. The plausibility of an attribute is based on its appearance in a statement, not on the I/O cost of the index as access path. Plausibility is a valid criterion for restricting the I/O cost evaluations. If we assume, that the solution generation procedure of section 5.3. is used, then all configurations are considered, and the optimal index configurations are found. Assuming that the estimated costs are the actual execution costs. Analyzing all possible atomic configurations, may be impractical when the workload includes joins on many tables, where a large number of attributes are plausible. As we already discussed in chapter 1, finding the optimal configuration is not required, as statistics provide an incomplete and approximate description of the database workload, as well as the cost estimations of chapter 4 and/or the optimizer cost computations provide us a rude approximation of real time I/O costs.

In what follows, we concentrate on some heuristics which will help us to decide on which plausible indexes are likely to be chosen as access paths by the optimizer when other indexes exist within the database. These criteria, based on access costs, can reduce the set of configurations. First we describe the problem of index indecision between index performance estimations and brute force performance estimation, then we describe an heuristic for index elimination on single table statements, after what we enlarge our considerations to multi table statements.

5.2.1. Index Indecision Problem

In section 4.2. we defined an optimistic and a pessimistic I/O cost estimation for B-Tree index accesses and we have seen the problem of the index indecision problem. In section 4.1. we defined a cost estimation for brute force table scans. During this section we will give a graphical interpretation (figure 5.2.) of the different cost relations and see how they led us to a heuristic on index selection.

First, recall the three cost relations for the B-Tree and the brute force table scan accesses. Note that, in this section, we abstract the relations from the time fractions, R, S, L, used to value the access cost in terms of time.

$$^{MAX}Cost_{I/O}(B\text{-}Tree_{NC}) = \lceil (Log_{fo}(Ndv) - 1) + (k / Nrid) \rceil + \lfloor k_{Ndv} / fo \rfloor + k$$

$$^{MIN}Cost_{I/O}(B\text{-}Tree_{NC}) = \lceil (Log_{fo}(Nrdv - 1) + (k / Nrid) \rceil + \lfloor k_{Ndv} / fo \rfloor + K$$

$$Cost_{I/O}(Brute\ Force) = Np/2$$

where
fo: Fanout of index node pages
Np: number of pages
Ndv: number of distinct records for the indexed attribute
Nrid: number of RID pointer hold in a page of indirection (rude approximation 254)
k: number of records to be retrieved
k_{Ndv}: number of distinct records values to be retrieved
K: number of pages to be accessed

Second, let us draw all three cost relations on a graphical figure, figure 5.2., and describe their intersections (See annexes 7.1.1.).

Recall that the parameters involved in our cost estimations are valued as it follows:

• number of records	Nr:	10000 records
• records size	Rs:	100 bytes
• indexed attribute size	As:	10 bytes *(page pointer size included)*
• page size	Ps:	2000 bytes *(header size already subtracted)*
• fill rate	fr:	70%
• number of records per page	Nrp:	14 records
• number of pages	Np:	714 pages
• fanout	fo:	70 records *(index page size and data page size are identical)*
• number of index levels	d:	3 levels
• number of RID pointers per indirection page	Nrid:	254 pointers

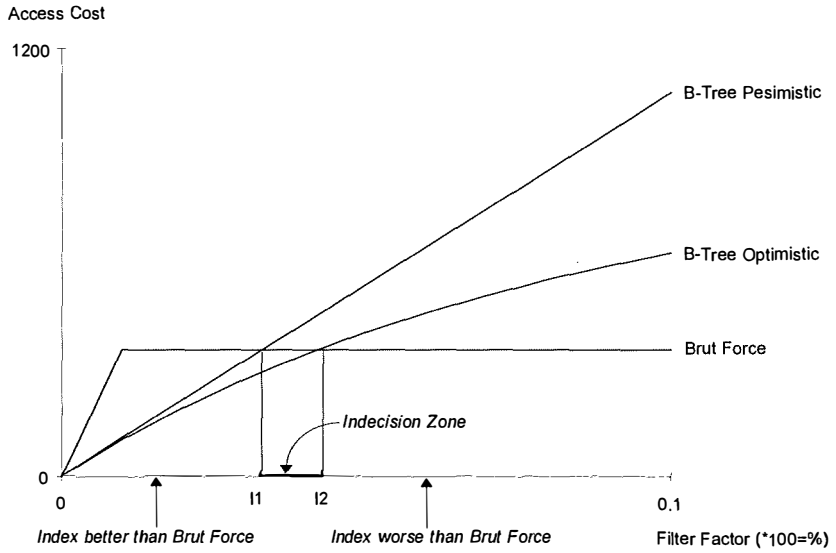


figure 5.2.: Index Indecision Problem Representation

Analyzing the graph, we figure out that the three cost relations cross within two points.

The first intersection takes place at when I1% of the records are qualified by the query. Meaning that when the query qualifies *less than I1*Nr records* the I/O cost using an index is *always better* than executing a brute force table scan.

The second intersection, arises when the query qualifies I2% of the records. Meaning that when the query predicate, qualified *more than I2*Nr records* the access costs using a B-Tree index is *always worse* than performing a brute force table scan.

The section between those two points, I1 and I2, is called the *Indecision Zone*. Meaning that when the predicate values a filter factor of ff ($I1 \leq ff \geq I2$), the designer might not be able to state that the I/O costs using an index are better or worse than executing a brute force table scan.

Above here we, defined three zones that guide the search for index placement, elimination, but what happens when the different input parameters vary in value. We can distinct between to kind of parameters; the one that are fix to the cost estimation and the one that are not.

Fixed parameters to the indecision problem, are for example the number of records, the records size and/or the indexed attribute size. They are said to be fix, because the designer has no control over their values. Given a query and data objects as well as their values, he has to decide whether the index placement makes sense or not.

Variable parameters to the indecision problem, are for example the page size and/or the fill rate. The designer might, choose according to the system and to its pre analyses to increase or decrease their values.

In annexes 7.1.2. and 7.1.3., we valued and represented variations of the page size and the fill rate. As we would expect, the higher we set the page size, the lower are the I/O costs for both index retrieval and brute force table scan. What might be surprising is that the higher we set the page size, the earlier comes the break point, in terms of filter rate, where the index search is always worse than the brute force table scan, and the smaller is the indecision zone. Same for variations of the fill rate, the higher we set the fill rate, the more records can be hold within a page, the less access have to be performed for data retrieval. What might be surprising, in this case, is that the higher we set the fill rate, the better becomes the brute force scan over the index. The fact is, when the fill rate varies the amplitudes in brute force scan costs are bigger than in B-Tree cost estimations, which influence highly the cost break point valuation.

Out of these variations we can learn that a lower page size or/and fill rates are beneficiary for indexes selection. However *it is folly to think* that a minimum page size and fill rate gives an optimal I/O cost for a given index. The designer should not think of tuning the page size and/or the fill rate given an index and a query. But to question himself, on the problem, if an index makes sense, or not, for a given page size, fill rate and query. The aim of this section was to show that an index can be more or less interesting for a given query depending on the pre-defined page size and/or fill rate.

5.2.2. Index Elimination for Single Table Statements

Assume that all considered statements involve one and only one table, hence, all atomic configurations have no more than one index. Index elimination is not usually necessary in this case, since the number of plausible atomic configurations is quiet small. However, to help motivate the technique used for index elimination in multi-table statement cases, we should begin with a single table statement case.

The database designer figures out all possible configurations, for clustered and non-clustered indexes, for all table attributes. Index elimination is carried out by comparing every index choice with every other index choice, as well as with the brute force table scan. A set of elimination criteria is valid if the criteria never eliminate an index that appears in the optimal solution. The elimination heuristics, describe later on, are only valid for single-table queries. When the workload also contains maintenance statements and joins, the heuristic may not be valid. The problems associated with the maintenance and join costs will be shortly inspected at the end of this section.

Let $COST_i(j)$ be the cost of query q_j associate with an index j. If $COST_i(k) < COST_i(j)$, then, if both indexes exist in the design, the optimizer will prefer k to j. We should also consider storage cost, s_j, for each index j, which is defined to be the number of pages needed for the index multiplied by a storage weight σ , supplied by the designer to trade off page costs versus execution costs in computing total configuration costs. If $COST_i(k) \leq COST_i(j)$, for all q_j, then the optimizer will never take j if k is in the database design. If this consideration holds, k is a better index choice than j, and we can eliminate j from consideration, unless the storage cost required for index k is higher that the one of index j, $s_k > s_j$.

The above consideration let us to the following definition.

Given two indexes j and k, if $s_k \leq s_j$ and, for all $q_j \in W$, $COST_i(k) \leq COST_i(j)$, then index j is dominated, as an index choice, by index k. If equality holds for all q_j and s, then j and k are said to be equivalent.

In case where indexes are equivalent, all but one can validly be eliminated. The configuration, with no indexes is represented by a vector $R(\rho)$ of costs $COST_i(\rho)$ that corresponds to the brute force table scan. In general, the optimizer never returns a cost $COST_i(j) > COST_i(\rho)$, so any index access cost equivalent to the brute force table scan cost can be validly eliminated.

Let us consider CLUST to be the set of plausible clustered indexes over all the queries, q_j, and NONCLUST to be the set of plausible non-clustered index over the same queries. The following four heuristics, based on the above definition, can be used to eliminate indexes from CLUST and NONCLUST. Note, that after that an index has been eliminated it cannot eliminate any other index.

H1: If j,k ∈ CLUST and index k dominates index j, then eliminate j from CLUST.

H2: If j ∈ CLUST is equivalent to k ∈ NONCLUST and index k and j are defined on the same table attribute(s), then eliminate j from CLUST.

H3: If j ∈ NONCLUST is equivalent to ρ, then eliminate j from NONCLUST.

H4: If j ,k ∈ NONCLUST and index k dominates index j, then eliminate j from NONCLUST.

Each of these heuristics is valid because we assumed earlier that the optimizer uses only one access path per table and there is only one table. Heuristic H1 and H2 should be applied in that fixed order before H3 and H4, since otherwise we may eliminate an index before it has the chance to eliminate others. Heuristic H3 and H4 may be applied in either order. Heuristic H1 and H4 eliminate dominated indexes and keep only one among equivalent indexes. Heuristic H2 eliminates any clustered index that is equivalent to the non-clustered index on the same attributes, because there is no advantage in keeping the records ordered on those attribute values. In H3, non-clustered indexes are compared to the configuration with no indexes and eliminated if equivalent. If the corresponding clustered indexes are equivalent to the brute force scan, ρ, they are eliminated by heuristic H2, since no non-clustered index can be better than a clustered index on the same attribute(s). After the application of the above heuristics, CLUST and NONCLUST contain only the indexes that are comparatively useful for at least one query, or have small storage costs.

	Clustered Indexes						Non-Clustered Indexes						No Index
	1c	*2c*	*3c*	*4c*	*5c*	*6c*	*1n*	*2n*	*3n*	*4n*	*5n*	*6n*	ρ
q_1	100	100	50	100	90	100	100	100	50	100	100	100	100
q_2	150	10	50	35	40	150	150	20	50	150	40	150	150
q_3	5	10	10	10	10	5	5	10	10	10	10	10	10
q_4	100	60	100	200	100	200	100	140	200	200	130	200	200

figure 5.2.: Cost Matrix for Index Elimination Problem

To illustrate these heuristics, let us consider a small example. Figure 5.2., gives us costs for a table T_i with a set of 6 plausible attributes for index selection and 4 queries. Normally, different attributes might be plausible for different queries. The single index atomic costs are arranged in a matrix with 4 rows, representing the queries, and 13 columns, 6 for costs of clustered indexes, 6 for the non-clustered ones and 1 for the brute force scan. Ignoring storage costs for simplicity, the results of index elimination is:

Results of heuristic index elimination:

- H1: 1c eliminates 6c, 2c eliminates 4c.
- H2: 1n eliminates 1c.
- H3: ρ eliminates 4n and 6n.
- H4: no elimination.

Further elimination heuristics may be applied to CLUST and NONCLUST if they still contain many elements. Other indexes can be eliminated if they are 'almost' dominated by some other index. If the strict domination criterion is used, indexes may survive the elimination process because they are slightly better than other for a small set of queries, even though they are worse for most queries. Therefore, heuristic elimination may be preferable to strict domination. Let the maximum advantage of k over j for all q_i be

$$Ma_{k,j} = \max\{w_i[C_i(j) - C_i(k)]\}$$

and let ε be an elimination coefficient specified between 0 and 1. Heuristics elimination can be based on the following domination definition:

An index k ***ε-dominates*** an index j if

$$Ma_{j,k} \leq \varepsilon Ma_{k,j}$$

and for storage costs

$$\sigma(s_k - s_j) \leq \varepsilon Ma_{k,j}$$

where σ is the storage weight supplied by the designer.

Index k ε-***dominates*** an index j if the maximum advantage of j over k, over both I/O cost estimations and storage costs, is less or equal to a fraction of the maximum advantage of k over j. Note that zero-domination is identical to the first domination definition, so index elimination is the same as index elimination with ε = 0. ε-***domination*** can be defined in other ways, for example by comparing total advantages or by comparing maximum advantage to total advantage. In here we prefer to compare maximum advantages, since this comparison means that eliminated indexes are comparatively unimportant.

Domination increases monotonously as ε increases; that is, if j ε_1-dominates k, then j ε_2-dominates k for $\varepsilon_1 < \varepsilon_2$. Assuming storage costs are equal, for any pair of indexes j and k there is a smallest ε between 0 and 1 such that one index ε-dominates the other, based on the ration of their maximum advantages. Note that if both maximum advantages are 0, the indexes are equivalent. ε-domination is not a transitive operand, so the order in which elimination is applied may change the set of eliminated indexes.

If the clustered index were chosen on a table, further index elimination could be done based on that choice. This motivates an additional elimination heuristic criterion. Fix a specific table. Let G_0 contain all surviving indexes on that table in NONCLUST. For each clustered index k on the given table that survived index elimination, let G_k contain clustered index k as well as the non-clustered survivors. Elimination is performed within each group G_k by applying a domination criterion using the clustered index within the group:

H5: *For each group G_k, if k ∈ CLUST ε-dominates an index j ∈ NONCLUST, if j and k are indexes on the same attribute(s), then eliminate j from G_k, but not from any other group.*

Heuristic, H5, eliminates the non-clustered index on the clustered attribute, which is always dominated by the corresponding clustered index, and eliminates other indexes that are dominated by the clustered index. Elimination using H5 *can be done only group by group and not globally* on NONCLUST, since non-clustered indexes dominated by some clustered choices may be useful for other clustered choices. The result of applying H5 to the groups are called by [Finkelstein 1988] the basic groups for the tables.

We previously showed index elimination for figure 5.2., which is the same as index elimination with ε = 0. Index elimination using the ε-domination definition for ε = $^1/_3$ yields the following results:

Results of heuristic index elimination with ε = $^1/_3$:

- H1: 1c eliminates 6c, 2c eliminates 4c, 3c eliminates 5c.
- H2: 1n eliminates 1c.
- H3: ρ eliminates 4n and 6n.
- H4: 2n eliminates 1n.
- H5: In basic group for 2c: 2c eliminates 2n and 5n.
- H5: In basic group for 3c: 3c eliminates 3n and 5n.

Basic groups after elimination with $\varepsilon = {}^1/_3$:

- (2c, 3n)
- (3c, 2n) and
- (2n, 3n, 5n)

For maintenance statements as well as queries, costs are compared only for atomic configurations. Since the rest of the solution is not determined, the designer cannot include the cost of maintaining other indexes in its cost comparisons during the index elimination phase. Hence, the elimination heuristics may not be valid when there are maintenance statements in the database workload.

5.2.3. Index Elimination for Multi-Table Statements

Most approaches to the index selection problem are restricted to single table statements. Approximate solutions are obtained by performing the index selection separately table by table. Most commercial systems have join methods performing table joins, using nested loop and/or merge scan as well as hybrid join methods. The optimizer chooses the sequence in which tables are joined, and the index used for accessing each table. For an n-way join, it can use two methods in any appropriated sequence of 2-way joins. In each join, the choice of table order, the join method, and the index on tables cannot be done independently.

According to [Finkelstein 1988] the single table heuristics from section 5.2.1. also hold, although not necessarily valid, for multi-table index elimination, when following assumptions are made:

A1: Indexes can only eliminate other indexes on the same table.

A2: Clustered indexes on join attributes can never be eliminated.

A3: Indexes on join attributes can never eliminate any other indexes.

Assumption, A1, arises because indexes are single table access paths. Assumption, A2, arises because merge scan is often a very efficient join method when both join attributes are clustered. This can seldom be detected from single index atomic costs. Consider again, for example, the two tables Prod and Order and the following SQL query:

```
SELECT O.Suppno, P.Qonord
FROM Prod P, Order O
WHERE O.ProdNo = P.ProdNo
AND O.SuppNo = 15
AND P.Qohand BETWEEN 100 AND 150
```

Assume that we made the decision to cluster the Prod table on attribute Descrip and that a non-clustering index on ProdNo exists for Prod. Given this, the best clustered index for Order is probably on SupNo. This allows a quick retrieval of the records from Order that have SupNo = 15. For each of these records, the corresponding records in Prod, having the same ProdNo, can be located using the index on ProdNo in a nested-loop join method. The best choice of clustered index for Order would be entirely different had the choice for clustered index on Prod been ProdNo. In that case, clustering Order on ProdNo enables an even faster processing of the above statement. The join predicate would be resolved by performing one pass over each table via the clustered index, by using the technique of the merge join. This example shows us, *that the selection of a clustered index cannot be done independently for each table.*

Assumption, A3, arises because some very good solutions would be ignored if indexes on join attributes were allowed to eliminate indexes on non-join attributes. Without considering assumption, A3, two negative results might occur. Again, suppose that our workload contains the above query. Now apply index elimination to non-clustered indexes on attributes SupNo and ProdNo of the Order table. Let us just consider the costs of the nested-loop join. As seen in section 2.1.8.1. one table must be the outer table, whereas the other is the inner table. For each qualifying record in the outer table, satisfying predicates on this table, matching records are found in the inner table, satisfying the join predicate and the other predicates on the inner table. Let k_X be the expected number of records that satisfy predicates on the outer table X, which gives us the number of times the inner table is scanned, let ρ_Y be the cost of the brute force scan of table Y, let $C(\alpha_j)$ be the cost of accessing the outer table using the index on attribute j, and let $C'(\alpha_j)$ be the access cost to retrieve records matching an outer record using the index on attribute j of the inner table. The optimizer cost estimation are as follows:

For the index on attribute O.SuppNo, the minimum of Acc1 and Acc2 are:

using Prod as the outer table:

$$Acc1 = \rho_{Prod} + k_{Prod}\, C'(O.SuppNo)$$

and using Order as the outer table:

$$Acc2 = C(O.SuppNo) + k_{Order}\, \rho_{Prod}$$

For the index on attribute O.ProdNo, the minimum of Acc3 and Acc4 are:

using Prod as the outer table:

$$Acc3 = \rho_{Prod} + k_{Prod}\, C'(O.PartNo)$$

and using Order as the outer table:

$$Acc4 = \rho_{Order} + k_{Order}\, \rho_{Prod}$$

Let us make the assumption that each index has I/O access costs that are less than the brute force scan and that Acc3 is less than both Acc1 and Acc2 (Acc3 < Acc1, Acc3 < Acc2). During index elimination we would eliminate the index on O.SuppNo. But, if we put an index on attribute P.Qonhand, the I/O cost of accessing Prod might bee significantly reduced. Let us define Acc1' and Acc3' by substituting C(P.Qonhand) to ρ_{Prod} in relation Acc1 and Acc3. Similarly we define Acc2' and Acc4' by replacing C'(P.Qonhand) to ρ_{Prod} in relation Acc2 and Acc4.

$Acc1' = C(O.Qonhand) + k_{Prod}\, C'(O.SuppNo)$

$Acc2' = C(O.SuppNo) + k_{Order}\, C'(O.Qonhand)$

$Acc3' = C(O.Qonhand) + k_{Prod}\, C'(O.PartNo)$

$Acc4' = \rho_{Order} + k_{Order}\, C'(O.Qonhand)$

The cost reduction we obtain from Acc2 to Acc2', is k_{Order} times greater than the cost reduction from Acc3 to Acc3'. If the value of k_{Order} is large, then the value for Acc2 will now be much less than the value of Acc3. Thus, the decision to eliminate the index on O.SuppNo was poor.

A second outcome that produces even worse results might occur if we do not consider assumption, A2. Again we will use the above statement as an example. The non-clustered index on O.ProdNo could eliminate all other non-clustered indexes on Order, since the cost Acc3 with Order, as inner table, is small. Similarly P.ProdNo could eliminate all other non-clustered indexes on Prod, because executing a nested loop join with Prod, as inner table, might be cheaper than the alternatives. These indexes would not be used for a nested-loop execution of the considered statement, since one of them would be on the outer table and there is no non-join predicate on either attribute. An index can be used to scan all the records in a table, but this is typically not profitable. Thus, the optimizer is forced to choose brute force scan on one of the two tables, and consequently one of the indexes will be useless for the considered statement.

However, the database designer is allowed to undertake some non-clustered index elimination on join attributes by other indexes. In case where Acc3 is bigger than Acc1 or Acc2. If we define an index on one of the attributes of Prod then Acc3 remains bigger than Acc1 or Acc2. This is true of Acc4 as well as Acc3, and non-clustered indexes are poor choices for merger-scan joins. Hence, the non-clustered index on join attribute O.ProdNo can safely be eliminated when it is dominated by another non-clustered index.

Our discussion of restrictions A2 and A3 shows that solutions for single-table cases do not extend, by combining all the individual solutions for each table, to the multi-table cases in a trivial way. According to [Finkelstein 1988] index elimination is a good heuristic when assumptions A1, A2 and A3 are followed.

5.3. Solution Generation

For the last step in the design process we will, explain how [Finkelstein 1988]s database design tool, DBDSGN deals with the generation of an index solution. The index solution step is a controlled search of the space of subsets of the survivor indexes in CLUST and NONCLUST to find good solutions to the index selection problem.

Solutions, which are index configurations, are annotated with the I/O costs, maintenance costs, and indexes used for each statement in the workload and the total cost. The total cost can also depend on the total storage and the storage weight σ if the designer wants to balance execution time versus the cost of storage. However we will ignore the storage cost in this section.

The indexes in CLUST and NONCLUST are stored in a list, the *survivor list*, where the clustered indexes proceed the non-clustered ones. The search in this list is done through a tree expansion that enumerates the configurations so that no configuration appears twice in the tree. The ordering of the survivor list is important, however it will be described later on in this section.

Before we start the tree expansion, we should give the tool whether there is an index storage limit or not, if there is, the designer has to supply the maximum number of pages available for an index configuration in the database.

Table	T_1	T_2
Survivors	2c 3c 2n 3n 5n	7c 9n
Basic Groups	2c 3n	7c 9n
	3c 2n 5n	9n
	2n 3n 5n	
Survivor List	2c 7c 3c 9n 2n 5n 3n	

figure 5.3.: Example of a Survivor's List and Basic Groups

The root of the tree represents the solution with no indexes at all. A node's children always have one additional index, so the nodes at level n have exactly n indexes. Adding a node's children to the tree is referred to as *expanding the node*. The tree expands according to the following rules:

Tree expansion rules

- The root is expanded with one child for each index on the survivor list. These nodes represent solutions having only one index in the database.
- For each node, expansion is done with indexes that appear later in the survivor list than any index already in the node.
- A node can be expanded only with indexes that belong to the basic groups of the clustered indexes already present in the solution represented by that node.
- If a node has no clustered index for a table, any clustered index on that table can be added, but only non-clustered indexes in the all-non-clustered basic solution for that table can be added to the node. Recall that clustered indexes precede non-clustered indexes in the survivor list.
- Any node that exceeds the index storage limit specified by the designer is pruned.

To explain how the tree grows let us take an example. Suppose that the design is for table T_1 and T_2, whose survivor list and basic groups are show in figure 5.3..

Figure 5.4. shows the first expansion of the tree with solutions having only one index on the database. In this figure the root has no index.

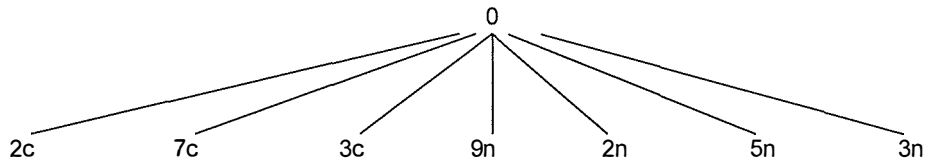


figure 5.4.: First Tree Expansion

In figure 5.5. the tree is expanded from the first level to the second, and the application of the expansion rules take place. For example, the solution represented by 7c is not expanded with 2c; the solution (2c, 7c) is already present and equivalent to (7c, 2c). No expansion takes place for the solution 3n, which is the last index in the survivor's list, but all possible combinations of 3n with other indexes appear elsewhere in the tree. Furthermore, 2c is not expanded with 3c or 5n because they do not belong to the same group, and 3c is not expanded with 3n.

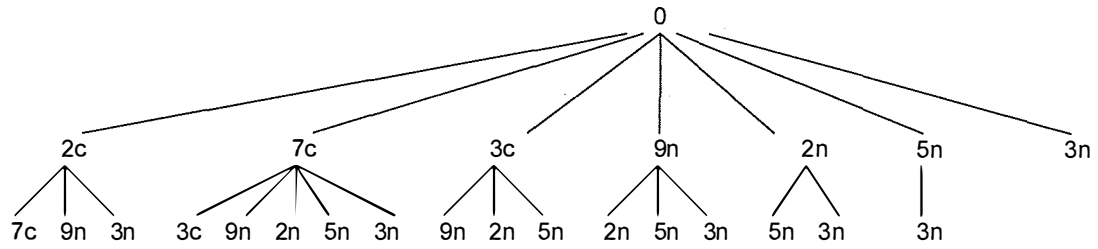


figure 5.5.: Second Tree Expansion

For each node the total cost is estimated during expansion. This total cost is the weighted sum of all costs of the statements, access and maintenance costs, when the database has the set of indexes represented by that node. Let ISET be the set of indexes in a given node. The total cost of the solution represented by the node is:

$$\text{TotalCost}[\text{ISET}] = \sum_q w_q \text{COST}[\text{ISET}]$$

where $\text{COST}_q[\text{ISET}]$ is the cost of statement q as defined in section 5.1.2. for the queries and in section 5.1.3. for maintenance statements.

The solution in a node can be worse than the parent solution. Assume we start from a node having ISET as a solution and add index α. The access advantage of a solution $\text{ISET}' = \text{ISET} \cup \{\alpha\}$ is

$$\text{Adv} = \text{TotalCost}[\text{ISET}] - \text{TotalCost}[\text{ISET}']$$

$$\text{Adv} = \sum_q w_q \text{COST}[\text{ISET}] - \sum_q w_q \text{COST}[\text{ISET}']$$

$\text{COST}_q[\text{ISET}']$ can be efficiently computed using atomic costs as the minimum of

- $\text{COST}_q[\text{ISET}]$ and
- the minimum value of $\text{COST}_q[\text{ASET}]$, taken over atomic subsets of ISET that contain α.

If α is used for q in configuration ISET', then the access paths for q correspond to those in some ASET containing α. Adv cannot be negative. If Adv is 0, no atomic cost including α is better than those without α, so α is not used as an access path in any statement for configuration ISET'. If Adv is positive, the disadvantage in maintenance cost must be considered:

$$D = \sum_q w_q \left[\sum_{\beta \in \text{SET}'} U_q(\beta, \text{AP}_q(\text{ISET}')) - \sum_{\beta \in \text{SET}} U_q(\beta, \text{AP}_q(\text{ISET})) \right]$$

The difference inside the square brackets, [], is not simply $\sum U_q(\beta, \text{AP}_q(\text{ISET}'))$, the additional maintenance cost for index α, because the maintenance costs of other indexes may have changed, based on the change of some index choices. Thus, to correctly evaluate maintenance costs, the tool has to keep track of actual access paths for each statement. ISET' has a better total cost than ISET only if Adv is greater than D.

Furthermore, knowing the actual indexes allow us to detect *wasteful* solutions. There are solutions that contain one or more indexes that are never taken in account. In the index elimination phase, we ensure that no index is dominated by any other single index. At level of solution generation we want to ensure that no index is *wasted* because it is overpowered by a set of other indexes. Wasteful solutions can arise in two ways:

The most recently added index α is not used in any statement. In this case ISET overpowers α.

When α is added to ISET, some index β in ISET is no longer in an access path for any statement. In this case $\text{ISET} \cup \{\alpha\} - \{\beta\}$ overpowers β.

If a wasted index is plausible for some join, future additions of indexes on different tables may make it useful for that join. If that index is not plausible for a join, or if all plausible indexes for the tables in the same join are already in the solution, then that index will always be wasted, no matter how the solution is expanded. In that case, the node can be pruned from the tree.

At the end of expansion, the tool displays the S solution having the smallest I/O cost, where S is a parameter specified by the designer. Wasteful solutions are never displayed, so they may be dropped as soon as they have been expanded. In figure 5.6. we show the full expansion of the tree. Recall that no index storage limit has been considered during expansion. The bordered nodes indicate the best solution (S=3). Some nodes are pruned because the tool detects wasteful solutions. When maintenance statements are present in the workload, the best solutions are not necessarily at leaves of the tree.

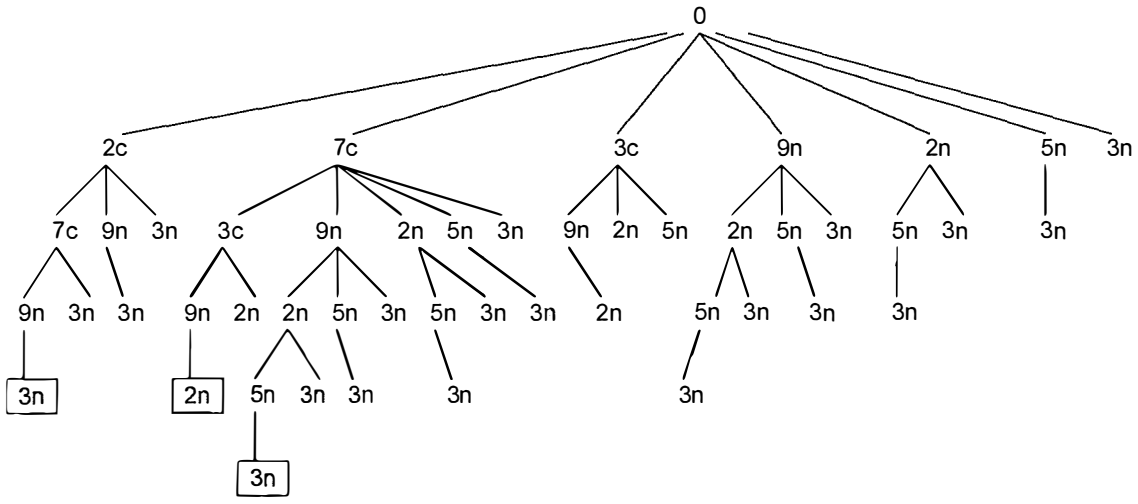


figure 5.6.: Full Tree Expansion

Exploring all the solution in the tree is very time consuming when there are many tables in the database with a large number of surviving indexes. Thus, we allow a controlled partial expansion of the tree, using breath-first search with heuristics pruning. The search is conducted by expanding all solutions with no more than L indexes and keeping only the best N, where N and L are parameters specified by the designer, with $N \geq S$. Each of these N solutions is then expanded by at most L additional indexes, assuming they were at the frontier of the previous expansion. This process continues until no further expansion is possible.

Because of this pruning rule, each different ordering of the survivor list determines a different tree, even though the set of solutions examined during the first L index expansion is always the same. This is because the survivor list imposes a discipline on a node's descendants, denying them certain indexes. We want 'influential' indexes to be early in the survivor list, so that other indexes can still be added, despite pruning. If the first index in the list was good only for some low-frequency statements, it might be pruned completely after the first L index expansions. The *order of the survivor list* is determined by two rules:

- The clustered indexes are stored before the non-clustered ones. Clustering indexes are, in general, the most influential. Thus, they should be considered before the non-clustered ones. They also allow the identification of the basic groups.

- The indexes in each set, clustered and non-clustered, are ordered according to their total costs, computed on the basis of their weighted total single index atomic cost: $\sum_q w_q \, Cost_q(\alpha)$. Indexes with higher total single costs are usually less influential than indexes with lower total single costs.

This ordering for the survivor list is likely to be one of the best among the possible permutations of indexes.

In figure 5.7. - 5.9., the expansion of the same tree as in figure 5.4. - 5.6., is shown with N = 3 and L = 1. We assume the order of the survivor list is the same as for figures 5.4. - 5.6.. The bordered nodes indicate the best solutions found at each level of expansion. The best solution obtained with pruning need not be the same as for full expansion. In this example, however, the number of nodes searched dropped from 48 to 27. If we set L = 1, we visit a number of nodes proportional to the number of surviving indexes. In general, the number of nodes grows exponentially in L. If we make L the number of surviving indexes, we visit the entire tree. The trade off, is that some of the low cost solutions appearing in the unrestricted tree may not appear in the restricted tree, so some of the best solutions may be missed. By controlling these two parameters, a designer can get a good set of choices rather quickly. The Tool allows the designer to pursue different choices of N, L, and the index storage limit in the same run. According to [Finkelstein 1988] choosing L = 1 usually allows the tool to find the best solution.

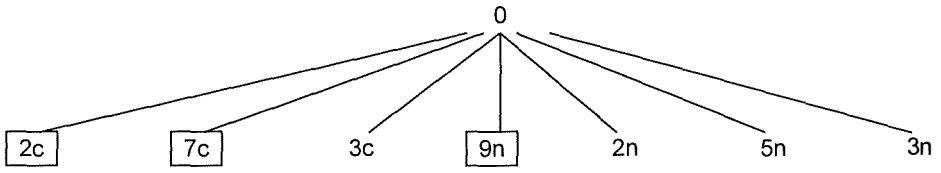


figure 5.7.: First Tree Expansion with N = 3 and L = 1.

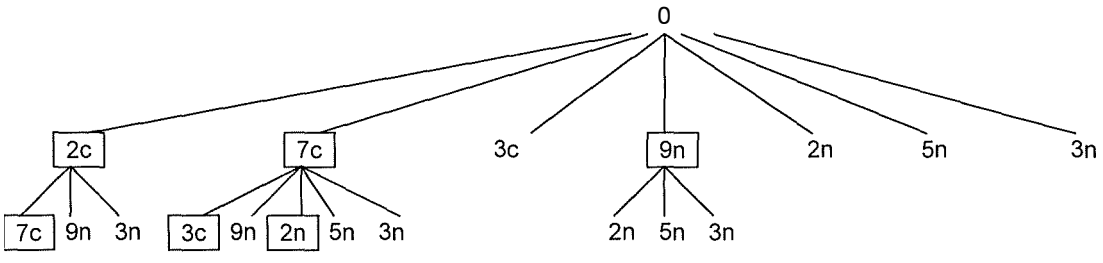


figure 5.8.: Second Tree Expansion with N = 3 and L = 1.

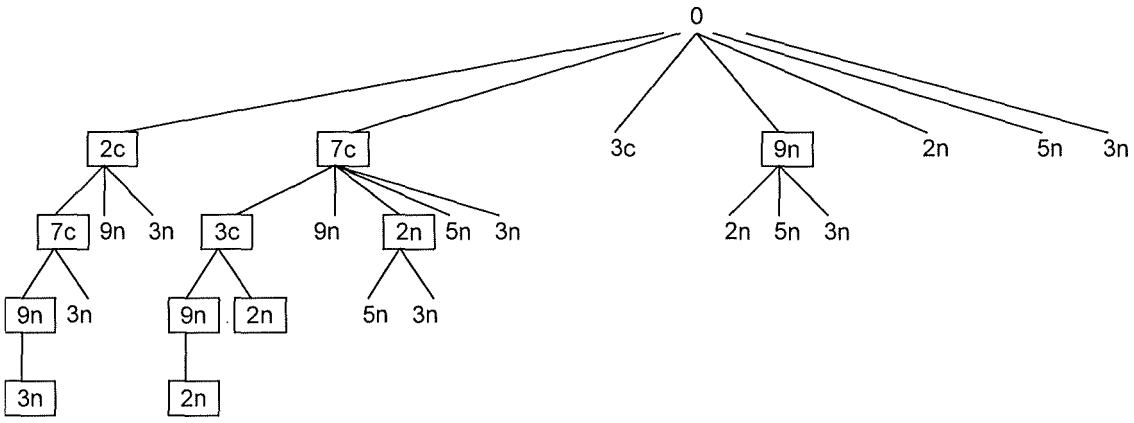
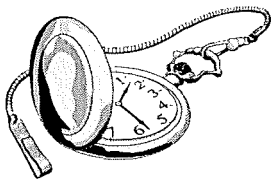


figure 5.6.: Full Tree Expansion with N = 3 and L = 1.



Chapter 6: Case Study

Throughout this document, we have seen many theoretical aspects and guidelines for physical database tuning. It is true that we concentrated on how indexes should be placed to gain maximum access performances. All this together can bee confusing. This chapter should bring some order into all this. Using as input a simplified version of a case study made by [Hainaut 1989], we will show how index tuning can take place in real world.

Before we start the phase of physical tuning, we consider that the logical database schema has been optimized according to the study made by [Mathon 1994]. Consider also that the queries are optimized according to the DBMSs optimizer considerations. Note that each DBMS documentation gives an explicit description of the query optimization.

In this chapter we will not consider the different aspects of buffer, page size, fill rate tuning, as they are strongly depending on database management system (ORACLE, SYBASE, INGRES, DB2, ...) and the operation system (UNIX, VMS, MVS, ...) under which the database will be implemented. We assume that the database designer completed the pre-job of tuning those parameters to their optimum.

6.1. Logical Schema

Figure 6.1. illustrates our case study logical schema. It represents the information stored within a small library. There are books having a set of keywords and being written by only one principal writer. Each book exists only one time within our library and can only be borrowed to one person. For each book the library stores the title, the publisher, the publishing year, the writer's identification, the borrower's identification, the date of borrowing and the foreseen date of return. For each writer the database holds information on its name and its nationality. Similarly, for each borrower the library stores the name and the address. To each book can bee associate 0 or 12 key words.

As already mentioned, the physical tuning uses as input the logical schema. For example the one in figure 6.1., representing our library database.

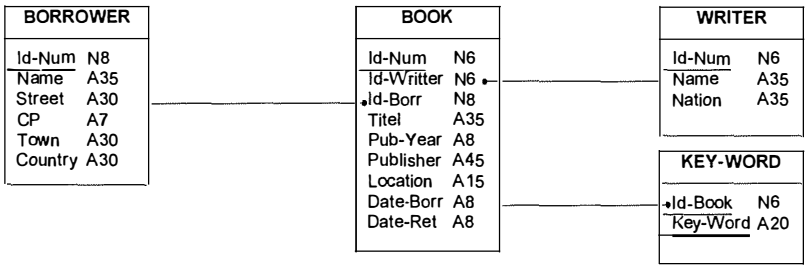


figure 6.1.: Logical Schema for Case Study

The schema would not be entire without the following referential constraints and integrity constraints.

- BOOK.Id-Writer in WRITTER.Id-Num
- BOOK.Id-Borr in BORROWER.Id-Num
- KEY-WORD.Id-Book in BOOK.Id-Num
- BOOK.Date-Ret ≥ BOOK.Date-Borr

6.2. Requirements Collection

6..2.1. Data Statistics

During requirements collection the following statistics have been gathered.

- The system uses data pages of 2Kb. For simplicity, we consider that the page size is equal to 2.000 bytes, reserving 48 bytes for the page header.
- The database is poor in insert and/or deletes and there are no record enlargements. The database designer determined a fill rate of 70% for the entire database.
- He estimated 10.000 persons that are allowed to borrow books.
- There are 50.000 books to be stored in the database.
- On the average the database designer identified 10 keywords per book.
- On the average a writer writes 3 books, thus there are 16.667 writers to be stored in the database.

Table Name	Nr	Rs	$Nrp = \lfloor \frac{Ps * fr}{Rs} \rfloor$	$Np = \lceil \frac{Nr}{Nrp} \rceil$
BORROWER	100.000 records	140 bytes	10 records/page	1.000 pages
BOOK	50.000 records	139 bytes	10 records/page	5.000 pages
WRITER	16.667 records	76 bytes	18 records/page	926 pages
KEY-WORD	500.000 records	26 bytes	53 records/page	9.434 pages

6.2.2. Queries

At the same moment, the database designer identifies a set of data operations. Already classified into query types (see section 2.1.8.).

Point queries:

Q1:
```
SELECT *
FROM BORROWER
WHERE Id-Num = #x
```

Q2:
```
SELECT Title, Pub-Year, Publisher, Location
FROM BOOK
WHERE Id-Num = #x AND Date-Borr NOT NULL
```

Multipoint queries:

Q3:
```
SELECT *
FROM BORROWER
WHERE Name = #y
```

Q4:
```
SELECT Id-Num, Title, Pub-Year, Publisher, Location
FROM BOOK
WHERE Date-Ret <= #today
```

Range queries:

Q5: SELECT Id-Num, Tittle, Pub-Year, Publisher, Location
FROM BOOK
WHERE Pub-Year BETWEEN #date1 AND #date2

Ordering queries:

Q6: SELECT Name, Nation
FROM WRITER
ORDER BY Nation, Name

Join queries:

Q7: SELECT B.Id-num, B.Title, B.Publisher, B.Pub-Year, W.Name
FROM BOOK B, WRITER W
WHERE BOOK.Id-Writer = WRITER.Id-Num

Q8: SELECT B1.Id-Num, B1.Name, B2.Title
FROM BORROWER B1, BOOK B2
WHERE B2.Id-Borr = B1.Id-num AND B2.Publisher = #x

Update operation:

U1: UPADTE BOOK
SET Date-Borr = NULL, Date-Ret = Null
WHERE Id-Borr = #y

U2: UPADTE BOOK
SET Date-Borr = #today, Date-Ret = #returnday
WHERE Id-Borr = #y

6.2.3. Query Statistics

After identifying the data operations, the designer identified the number of activation for each query and the number of records qualified during their activation.

- Query Q1 is requested on the average 5 times a day. As its predicate is defined upon a key attribute, it qualified only one record per activation. Thus the predicate has a good filter factor.
- Query Q2 is on the average activated 20 times a day. As one third of the books are borrowed and as the predicate Id-Num = #x involves a key attribute, the query qualifies on the average 1,5 records.
- Query Q3 is initiated 0.5 times a day. Consider that the predicate filter factor is equal to 2%, as there are on the average 20 persons with the same name. Thus the query retrieves 5.000 records.
- Query Q4 becomes active once a day. Selecting 2 records out of 1/3 of the records, as on the average 5 books out of the 30% of borrowed books are not returned in time.
- Query Q5 is on the average activated 0,5 times a day. Requirement collection identified that on the average 100 records are qualified.

- Query Q6 is initiated once a month, thus 0.05 times a day. It qualifies all, 16.667, records, as it establishes a report for all the writers, ordered by nation and name.
- Query Q7 is activated once a week, thus 0.2 times a day. As each book has a writer, it generates 50.000 records.
- Query Q8 becomes active once a day. Consider that there are 1.000 publishers and that the books are uniformly distributed among those publishers.
- Update operation U1 is initiated 100 times a day. Consider that a persons borrows on the average 2 books at the same time and that he returns them also on the same date. Then the operation qualifies 2 book records per day.
- Update operation U2 is activated 120 times a day. When a person borrows on the average 2 books at each visit, 2 records are concerned per day.

Let us consolidate the above valuation into a spreadsheet.

Data Operations	Number of activation per day (Na/d)	Number of records qualified for one activation (k)	Number of records qualified for one day (k/d)
Q1	5	1	5
Q2	20	1,5	30
Q3	0,5	5000	2.500
Q4	1	5	5
Q5	0,5	100	50
Q6	0,05	16.667	834
Q7	0,2	50.000	10.000
Q8	20	2	40
U1	100	2	200
U2	120	2	240

6.2.4. First Set of Plausible Indexes

As we identified the various data access operations upon the database, we can define a first set of plausible indexes for each table according to the following guidelines.

- *Include all the table identifiers.*
- *Include all attributes used in a reference constraint.*
- *Include all attributes used in the WHERE clause, in other words referenced by a select predicate.*
- *Include all attributes used in the ORDER BY clause.*
- *Include all attributes used in the GROUP BY clause.*
- *Include all attributes used in the HAVING clause.*

Table Name	Plausible indexes
BORROWER	Id-Num, Name
BOOK	Id-Num, Id-Borr, Id-Writer, Date-Borr, Date-Ret, Pub-Year, Publisher
WRITER	Id-Num, Nation, Name
KEY-WORD	Id-Book, Key-Word

Beside the definition of the first set of plausible indexes, it is helpful to see which table is concerned by which query.

Table Name	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	U1	U2
BORROWER	x		x					x		
BOOK		x		x	x		x	x	x	x
WRITER						x	x			
KEY-WORD										

6.2.5. Filter Factors

Let us determine the filter factors for the various predicates.

Q1: $ff_{Id\text{-}Num = \#x} = 1/100.000 = 0.00001$

Q2: $ff_{Id\text{-}Num = \#x} = 1/100.000 = 0.00001$

$ff_{Date\text{-}Borr\ NOT\ NULL} = 33.334/100.000 = 0.33$

Q3: $ff_{Name = \#x} = 20/100.000 = 0.0002$

Q4: $ff_{Date\text{-}Ret < \#today} = 5/16.667 = 0.0003$

Q5: $ff_{Pub\text{-}Date\text{-}Ret\ BETWEEN\ \#date1\ AND\ \#date2} = $ see later

Q6: $ff_{Q6} = 1$

Q7: $ff_{BOOK.Id\text{-}Writer = \#x} = $ see later

$ff_{WRITER.Id\text{-}Num = \#x} = $ see later

Q8: $ff_{B1.Id\text{-}Borr = \#x} = $ see later

$ff_{B2.Id\text{-}Num = \#x} = $ see later

$ff_{B2.Publisher = \#x} = 50/50.000 = 0.001$

6.2.6. Assumptions

Throughout our inquiry we will consider each table separately. For each table and each query we determine the set of plausible non-clustered and clustered indexes, using I/O costs from chapter 4.

Consider that an index page is 2Kb long and a record pointer is 8 bytes long. A page of indirection for an index defined on a non-key attribute, can hold up to 256 RID pointers. Same as for data pages we define a fill rate of 70%. We consider that the root level of the index is always located in the buffer.

6.3. Table KEY-WORD

Throughout the analysis of the requirement collection, we could not identify any data operation using table KEY-WORD. That does not mean that table KEY-WORD is never referenced. Suppose, that there exists a hide transaction, which lists all the book for a given keyword. Then it would be beneficiary to have a composite index defined upon attribute Key-Word and Id-Book. However, this index is not at all beneficiary for a transaction that lists all the keywords of a given book. In this case a composite index on Id-Book and Key-Word would be beneficiary.

6.4. Table BORROWER

Table BORROWER is accessed by data operations, Q1, Q3 and Q8. Note that query Q8 is a Join query and implies also table BOOK.

6.4.1. Query Q3: Index on Name?

Let us calculate the fanout for the index on attribute Name.

$$fo = \left\lfloor \frac{2.000 * 0.7}{35 + 8} \right\rfloor = 32$$

Knowing that on an average there are 20 persons with the same name, we deduce that there are 5.000 distinct values for attribute Name. Now let us calculate the depth of the tree.

For a non-clustered index: $d = \lceil \log_{32} 5.000 \rceil = 3$

For a clustered index: $d = \lceil \log_{32} 1.000 \rceil = 2$

Let us determine the filter factor of predicate Name = #y. We know that the query retrieves daily 5.000 records, thus the daily filter factor is equal to 0,05. Using relation 4.6., we estimate that the 5.000 records are located within:

$$K = \lceil (1 - (1 - 0.05)^{10}) * 1.000 \rceil = 401$$

Non-Clustered Index I/O Costs.

As attribute Name is a non-key attribute, we use relation 4.23.a. to determine the I/O costs incurred by a search throughout a non-clustered B-Tree. Knowing the daily activation, we deduce the average number of I/Os, the query processes per day.

$$COST_{B-Tree} = \left\lceil 2 + \left(\frac{5.000}{256} \right) \right\rceil + \left\lfloor \frac{1}{32} \right\rfloor + 401 = 423 \text{ I/Os per activation}$$

$\Leftrightarrow$ an average of 423*0,5 $\approx$ 212 I/Os per day

Clustered Index I/O Costs.

Similarly, using relation 4.23.b., the costs incurred, using a clustered B-Tree are equal to:

$$COST_{B-Tree} = 2 + \left\lfloor \frac{5.000}{10} \right\rfloor = 502 \text{ I/Os per activation}$$

$\Leftrightarrow$ an average of 502*0,5 $\approx$ 251 I/Os per day

Brute Force Table Scan I/O Costs.

Estimating costs for the brute force table scan, using relation 4.8.. We access half of the pages to answer the query.

$$COST_{BruteForce} = \frac{1.000}{2} = 500 \text{ I/Os I/Os per activation}$$

$\Leftrightarrow$ an average of 500*0,5 $\approx$ 250 I/Os per day

It is obvious, that an index on attribute Name speeds up access performances for query Q3. We also see that the non-clustered index gives better performances than a clustered index or a brute force scan. Nevertheless, as the table is involved in other queries, we cannot decide, at this point, if a primary or secondary index is best for the overall performance.

6.4.2. Query Q1: Index on Id-Num?

Let us calculate the fanout for the index on attribute Name.

$$fo = \left\lfloor \frac{2.000 * 0.7}{6+8} \right\rfloor = 100$$

We know that attribute Id-Num is a key attribute and that query Q1 is a point query, hence we can use a simplified version of the I/O cost determination.

For a non-clustered index: $d = \lceil \log_{100} 100.000 \rceil = 3$

For a clustered index: $d = \lceil \log_{100} 1.000 \rceil = 2$

Non-Clustered Index I/O Costs.

As attribute Id-Num is a key attribute, we use relation 4.22.a., to determine the I/O costs incurred by a search throughout a non-clustered index. Knowing the daily activation, we deduce the average number of I/Os the query processes per day.

$COST_{B-Tree} = 3$ I/Os per activation

$\Leftrightarrow$ an average of 3*5 = 15 I/Os per day

Clustered Index I/O Costs.

Similarly, using relation 4.22.b., the costs incurred using a clustered B-Tree are equal to:

$COST_{B-Tree} = 2$ I/Os per activation

$\Leftrightarrow$ an average of 2*5 = 10 I/Os per day

Brute Force Table Scan I/O Costs.

$$COST_{BruteForce} = \frac{1.000}{2} = 500 \text{ I/Os I/Os per activation}$$

$\Leftrightarrow$ an average of 500*5 = 2.500 I/Os per day

Comparing the various costs for query Q1, we see that an index on attribute Id-Num highly improves I/O costs. Hence, we retain attribute Id-Num as a plausible candidate for indexation.

6.4.3. Query Q8: Index on Id-Num?

Recall that query Q8 is a Join query. It joins table BOOK and BORROWER., on attribute Id-Num. Hence Id-Num is a plausible index choice for our configuration. However, we have to foreseen the evaluation plan of the optimizer to determine, whether Id-Num is a good index choice or not. Assume that the optimizer uses the Nested Loop Join technique to join both table. It is clear that there are other techniques to join a table, we therefore reference the reader to section 2.1.8..

The number of records in BOOK is smaller than the number of records in BORROWER. We can assume that the optimizer uses BOOK as being the outer table and BORROWER the inner table. Consider that predicate B2.Publisher = #x is not present. In this case, BORROWER is accessed 50.000 times throughout attribute Id-Num, hence a cost of 3 * 50.000 = 150.000 I/Os using a non-clustered index and a cost of 100.000 I/Os using a clustered index. Now, consider that predicate B2.Publisher = #x exists. Hence, the optimizer first accesses the BOOK throughout attribute Publisher, and reduces the number of records to lookup in table BORROWER to 50.

Thus the costs of accessing table BORROWER are:

Non-Clustered Index I/O Costs.

$COST_{B-Tree} = 50 * 3 = 150$ I/Os per activation

$\Leftrightarrow$ an average of 150*2 = 300 I/Os per day

Clustered Index I/O Costs.

$COST_{B-Tree} = 50 * 2 = 100$ I/Os per activation

$\Leftrightarrow$ an average of 100*2 = 200 I/Os per day

Brute Force Table Scan I/O Costs.

$$COST_{BruteForce} = 50 * \left(\frac{1.000}{2}\right) = 25.00 \text{ I/Os I/Os per activation}$$

$\Leftrightarrow$ an average of 25.000*2 = 50.000 I/Os per day

Note, that there is a high variation in performance between an index access and a brute force table scan. Thus it is recommended to define an index upon attribute Id-Num.

6.4.4. Clustered or Non-Clustered Indexes ?

For simplicity, we note, $C_{Id\text{-}Num}$, the clustered index on Id-Num, and, C_{Name}, the clustered index on Name. Similarly, we note, $I_{Id\text{-}Num}$, the non-clustered index on Id-Num, and, I_{Name}, the non-clustered index on Name. In our tabulation we represent the brute force table scan by the symbol, ρ.

	$C_{Id\text{-}Num}$	C_{Name}	$I_{Id\text{-}Num}$	I_{Name}	ρ
Q1	10		15		2.500
Q3		251		212	250
Q8	200		300		50.000

Using the heuristics described in section 5.2.2., we can say that $C_{Id\text{-}Num}$ dominates C_{Name}. Thus we define a clustered index on Id-Num. As we define a clustered index on Id-Num, we remove index $I_{Id\text{-}Num}$ from the set of non-clustered indexes. Leaving index I_{Name} as the only non-clustered index.

To resume index solution on table BORROWER, we define a clustered B-Tree index on attribute Id-Num and a non-clustered B-Tree index on attribute Name.

6.5. Table WRITER

We use a similar reasoning as the one for table BORROWER. Table WRITER is referenced by query Q6 and Q7. Query Q6 involves attribute Nation and Name, to create an ordered list of all the writers by nationality and name. However, query Q6 does not include any predicate. At a first glance, we would think of a clustered index on Nation and Name, to be beneficiary, as it orders the table physically in the requested sequence.

6.5.1. Query Q6: Index or not?

As already mentioned, we are likely to go for a cluster, as the query involves an ORDER BY clause (see section 3.2.3.). However, the query does not include a restricting predicate (no WHERE clause), therefore it qualifies all the records within the table. During the query evaluation the optimizer might encounter two situations. First, the table is not ordered by nationality and/or name, then the system accesses all pages an performs an internal sort, which is CPU consuming. Second, the table is ordered by nationality and name, in this case there is no need to perform a sort, thus the only cost encountered by the system is the sequential access cost of 926 pages. Note that in both situations we access all pages.

But, as the query is executed only once a month, it is not critical to performance. Moreover, the CPU time needed to perform the sort is relatively low compared to the costs encountered by inserting and/or deleting records in the table. On the other hand, there are only a few inserts and/or deletes upon table WRITER.

Let us go further in our reasoning and think of a new possible query. Consider that we add a predicate to the query, restricting the target list to a set of two given nationalities.

```
Q6a: SELECT Name, Nation
     FROM WRITER
     WHERE Nation = #x OR Nation = #y
     ORDER BY Nation, Name
```

Consider that there are 20 nationalities and that the writers are uniformly distributed among them. In this case, predicate Nation = #y qualified 16.667/20 = 834 records, thus a filter factor of 0,05. Moreover, consider that the query is initiated once a day.

1. Putting a clustered index on non-key attribute Nation speeds up retrieval. What might be surprising, is that accessing the table throughout a non-clustered index is more I/O consuming than using the brute table force scan.

   Non-Clustered index:

   $$COST_{B-Tree} = 2 * \left(\left\lceil 0 + \frac{834}{256} \right\rceil + \left\lfloor \frac{1}{32} \right\rfloor + 559 \right) = 1.126 \text{ I/Os per activation}$$

   $\Leftrightarrow$ an average of 1.1126 I/Os per day

Clustered index:

$$COST_{B-Tree} = 2 * \left(\left\lceil 2 + \frac{834}{18} \right\rceil \right) = 98 \text{ I/Os per activation}$$

$\Leftrightarrow$ an average of 98 I/Os per day

Brute Force Table Scan:

$$COST_{BruteForce} = 2 * \left(\frac{926}{2} \right) = 926 \text{ I/Os}$$

$\Leftrightarrow$ an average of 926 I/Os per day

2. Putting a non-clustered index on composite key attribute Nation, Name, is highly beneficiary as the index answers the query. This is somehow a similar situation then the clustered index on attribute Nation. Except that, we reproduce the records at index leaf level. We gain in performance, because all the retrieved records are already in sorted order. However, this situation is space consuming as we reproduce exactly the data already held in the table.

6.5.2. Query Q7: Index on Id-Num?

Again, we encounter the situation where a query references two tables. Query Q7 joins tables BOOK and WRITER on attribute Id-Num.

Same as for query Q8, we consider that the optimizer uses the Nester Loop Join to join the tables. As WRITER holds the fewest records, the optimizer might consider this table as the outer table. Hence, BOOK is the inner table.

If we consider this there is no need to put an index on attribute Id-Num, as all its records are accessed.

Note however, that defining an index on Id-Num allows the system to check quickly for duplicate keys, while records are added. Therefore, it might be beneficiary to define an index on attribute Id-Num.

6.6. Table BOOK

Observe that table BOOK is used by six data access operations, Q2, Q4, Q5, Q7 and Q8, as well as by two data update operations, U1 and U2. Throughout the preceding sections, we have already seen some aspects of the queries Q7 and Q8.

6.6.1. Query Q2: Index on Id-Num and/or Date-Borr?

In query Q2, we can identify two predicates, Id-Num = #x, and, Date-Borr NOT NULL.

Using the optimizer principles from section 5.1.2. and the fact that the optimizer is likely to use the predicate with the best filter factor first during query evaluation (see section .2.1.5.)

Hence, we define an index on attribute Id-Num and not on Date-Borr. As the filter factor of predicate Id-Num = #x is better than the filter factor of predicate Date-Borr NOT NULL.

We also see that operation U1 and U2 are executed quite often. Because they update the values of attribute Date-Borr, this might generate a lot of dynamic reorganization within the index. Therefore, it is not beneficiary to define an index on attribute Date-Borr.

Using the same reasoning as for the other tables we determine the following I/O costs for the index on key attribute Id-Num.

Non-Clustered:

$COST_{B-Tree}$ = 3 I/Os per activation

$\Leftrightarrow$ an average of 20*3 = 60 I/Os per day

Clustered index:

$COST_{B-Tree}$ = 2 I/Os per activation

$\Leftrightarrow$ an average of 20*2 = 40 I/Os per day

Brute Force Table Scan:

$$COST_{BruteForce} = \frac{5.000}{2} = 2.500 \text{ I/Os}$$

$\Leftrightarrow$ an average of 20*2.500 = 50.000 I/Os per day

6.6.2. Query Q4: Index on Date-Ret?

It is not a good solution to define an index on attribute **Date-Ret**, as it is used only twice a day, to retrieve the books that are not returned within time. Whereas, it is updated 120 times a day, making it volatile.

As we already mentioned in the preceding section it, is not beneficiary to define an index upon attribute, whose values are often updated. Hence, we eliminate the index from the set of plausible indexes.

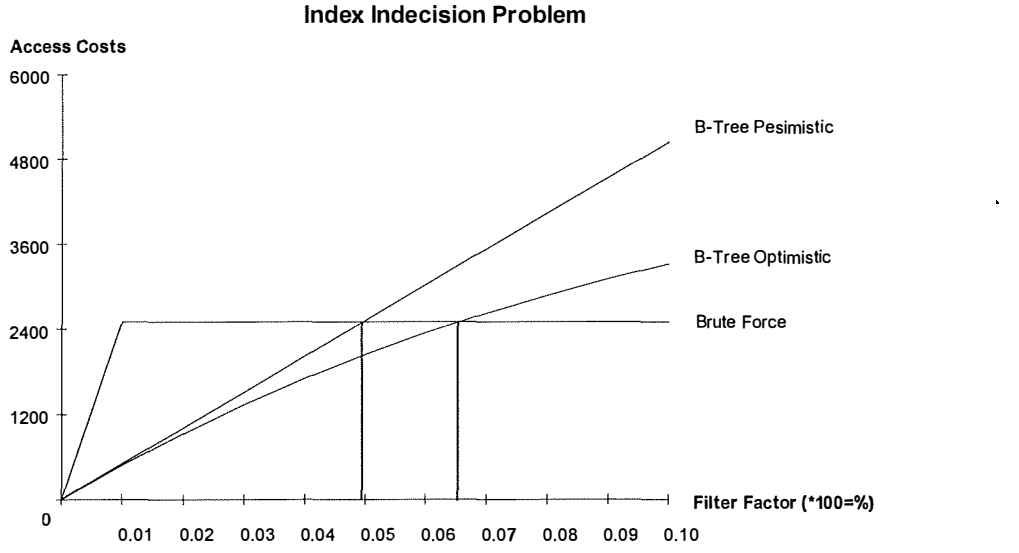
6.6.3. Query Q5: Index on Pub-Date.

For query Q5, we have not been able to a fix filter factor, since the range changes dynamically at query activation time. Let us see how we can use the filter factor growth and changes to determine index elimination.

The following sampling outlines the I/O access costs for a non-clustered index and a brute force table scan.

ff	k	K	Brut Force	B-Tree Pessimistic	B-Tree Optimistic
0	0	0	0	2	1
0.0100	500	481	2500	506	487
0.0200	1000	921	2500	1011	931
0.0300	1500	1321	2500	1516	1336
0.0400	2000	1686	2500	2020	1706
0.0500	2500	2017	2500	2525	2042
0.0600	3000	2319	2500	3030	2349
0.0700	3500	2593	2500	3535	2627
0.0800	4000	2841	2500	4040	2881
0.0900	4500	3066	2500	4544	3110
0.1000	5000	3270	2500	5049	3319

Using the graphical representation of the outlined costs, we may say, that an index on Pub-Date is beneficiary as long as the query retrieves less than 0.05*50.000 = 2.500 records. Similarly we may say, that the index is not at all beneficiary, when more than 0.065*50.000 = 3.250 records are qualified. For the quantity of qualified records which lies in between, we cannot decide on the advantages or disadvantages of an index over a brute force table scan evaluation.



Since the index values are not updated and since there are a small number of inserts and/or deletes upon BOOK, we might express that it is always beneficiary to define an index on Pub-Date, as long as the filter factor does not exceed 0.05.

Consider that on an average, the query retrieves 2.500 records uniformly distributed among 30 years. We determine the succeeding I/O costs for query Q5 using non-clustered, clustered indexes or a brute force table scan to retrieve the target records.

Non-Clustered:

$$COST_{B-Tree} = \left\lceil 2 + \frac{3.250}{256} \right\rceil + \left\lfloor \frac{30}{100} \right\rfloor + 2.447 = 2462 \text{ I/Os per activation}$$

$\Leftrightarrow$ an average of 0.5*2.46 = 1.231 I/Os per day

Clustered index:

$$COST_{B-Tree} = 2 + \left\lfloor \frac{3.250}{10} \right\rfloor = 327 \text{ I/Os per activation}$$

$\Leftrightarrow$ an average of 0.5*327 $\approx$ 164 I/Os per day

Brute Force Table Scan:

$$COST_{BruteForce} = \frac{5.000}{2} = 2.500 \text{ I/Os}$$

$\Leftrightarrow$ an average of 0.5*2.500 = 1.250 I/Os per day

It is obvious that we will put an index to attribute Id-Writer, as the index yields much better performance consideration than the brute force table scan.

6.6.4. Query Q7: Index on Id-Num?

Analyzing query Q7 for table WRITER, we considered that WRITER is the outer table and BOOK is the inner table of the Nested Loop Join. Therefore, we access table BOOK using attribute Id-Num, at least 16.667 times. Each access qualifies one and only one record, as Id-Num is a key attribute.

Non-Clustered:

$$COST_{B-Tree} = 16.667 * 2 = 33.334 \text{ I/Os per activation}$$

$\Leftrightarrow$ an average of 0.2*33.334 $\approx$ 6.667 I/Os per day

Clustered index:

$$COST_{B-Tree} = 16.667 * 2 = 33..334 \text{ I/Os per activation}$$

$\Leftrightarrow$ an average of 6.667 I/Os per day

Brute Force Table Scan:

$$COST_{BruteForce} = 16.667 * \left(\frac{5.000}{2}\right) = 41666.500 \text{ I/Os}$$

$\Leftrightarrow$ an average of 0,2*41.666.500 = 8.333.500 I/Os per day

Comparing the cost, it is obvious to see that an index improves access performances.

6.6.5. Query Q8: Index on Id-Borr or Publisher?

As already considered for table BORROWER, we consider that the optimizer performs a Nested Loop Join. It uses BOOK as the outer table and BORROWER as the inner table. Further, we considered that the optimizer uses first attribute Publisher to qualify a restricted set of records, since it has a good filter factor. Defining an index on attribute Publisher is beneficiary for record lookup.

Non-Clustered:

$$COST_{B-Tree} = \left\lceil 2 + \frac{50}{256} \right\rceil + \left\lfloor \frac{1}{27} \right\rfloor + 248 = 248 \text{ I/Os per activation}$$

$\Leftrightarrow$ an average of 2*248 = 496 I/Os per day

Clustered index:

$$COST_{B-Tree} = 2 + \left\lfloor \frac{50}{10} \right\rfloor = 7 \text{ I/Os per activation}$$

$\Leftrightarrow$ an average of 2*7 = 14 I/Os per day

Brute Force Table Scan:

$$COST_{BruteForce} = \frac{5.000}{2} = 2.500 \text{ I/Os}$$

$\Leftrightarrow$ an average of 2*5.000 = 10.000 I/Os per day

Comparing the I/O costs, we deduce that it is beneficiary to define an index upon attribute Publisher.

As we use BOOK, as the outer table, and as we use Publisher to restrict the set of records to qualify records in BORROWER, we do not need to define an index on attribute Id-Borr.

6.6.6. Clustered of Non-Clustered Indexes ?

As only one single clustered index can exist for one table, we must still determine which of the plausible indexes to cluster. According to what we have seen throughout the preceding lines, we are able to define the following set of plausible indexes.

{Id-Num, Pub-Date, Publisher}

Let us represent the clustered and non-clustered indexes as follows:

$C_{Id\text{-}Num}$ the clustered index on Id-Num
$C_{Pub\text{-}date}$ the clustered index on Pub-Date
$C_{Publisher}$ the clustered index on Publisher
$I_{Id\text{-}Num}$ the non-clustered index on Id-Num
$I_{Pub\text{-}date}$ the non-clustered index on Pub-Date
$I_{Publisher}$ the non-clustered index on Publisher

In our tabulation we represent the brute force table scan by the symbol, ρ.

	$C_{Id\text{-}Num}$	$C_{Pub\text{-}date}$	$C_{Publisher}$	$I_{Id\text{-}Num}$	$I_{Pub\text{-}date}$	$I_{Publisher}$	ρ
Q2	40			60			2.500
Q5		164			1231		1250
Q7	6.667			6.667		496	8.33.500
Q8			14				1.250

Using the heuristics described in section 5.2.2., we can say that $C_{Publisher}$ dominates $C_{Pub\text{-}date}$, and $C_{Id\text{-}Num}$, therefore we define a clustered index on Publisher. Thus removing $I_{Publisher}$ from the set of non-clustered indexes. All other indexes are better in access than brute force table scan, hence we define indexes on all attributes referenced in the set of non-clustered indexes.

6.7. Index Solution

Let us abstract, the above lines into a synthesis of the indexes to define, to accomplish an overall good access performance.

For table BORROWER, we retained the following indexes:

. a clustered index on attribute Name, and

. a non-clustered index on attribute Id-Num.

For table BOOK, we registered the following indexes:

. a clustered index on attribute Publisher,

. a non-clustered index on attribute Id-Num, and

. a non-clustered index on attribute Pub-Date.

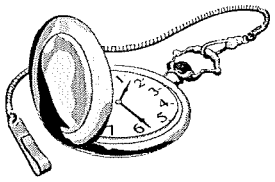
For table WRITER, we registered the following indexes:

. a clustered index on attribute Nation.

For table KEY-WORD, we retained no index at all.

Note, that due to our restricted set of queries, the solution excludes all plausible indexes on foreign key attributes, as there is no query which makes, per se, use of it. However, it is a good reasoning to define indexes on foreign key attributes, as normally there are queries using the foreign key attributes to retrieve data.

Further, our solution excludes nearly all plausible indexes on key attributes. In real world this is generally not the case, as there are various possibilities of inserts and/or deletes upon the database. Defining an index on a key attribute, enables quick verification of duplicates.



Chapter 7: Annexes

7.1. Index Indecision Examples

7.1.1. Example 1: Basic Data

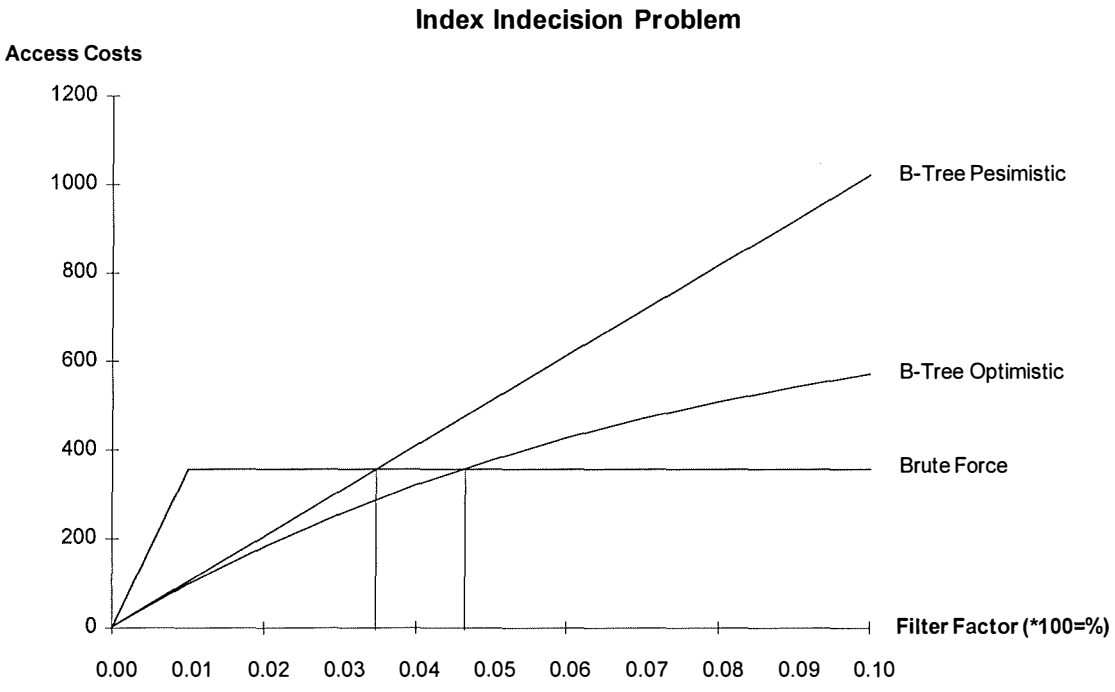
7.1.1.1. Input Parameters

Nr	**10000**	**Rs**	**100**
As	**20**	**Ps**	**2000**
fr	**0.70**	**Nrp**	**14**
Np	**714**	**fo**	**70**
d	**3**	**Nrid**	**254**

7.1.1.2. Cost Estimations

ff	s	k	K	Brute Force	B-Tree Pessimistic	B-Tree Optimistic
0	1.00	0	0	0	2	2
0.0100	0.99	100	94	357	104	98
0.0200	0.98	200	176	357	206	182
0.03	0.97	300	248	357	308	256
0.0400	***0.96***	***400***	***311***	***357***	***410***	***321***
0.0500	***0.95***	***500***	***366***	***357***	***512***	***378***
0.06	0.94	600	414	357	613	427
0.0700	0.93	700	456	357	715	471
0.0800	0.92	800	492	357	817	509
0.09	0.91	900	524	357	919	542
0.1	0.90	1000	551	357	1021	572

7.1.1.3. Graphical Representation



7.1.2. Example 2: Varying pages size

7.1.2.1. Input Parameters

a)

Nr	10000	Rs	100
As	20	**Ps**	**4000**
fr	0.70	Nrp	14
Np	714	fo	70
d	3	Nrid	254

b)

Nr	10000	Rs	100
As	20	**Ps**	**1000**
fr	0.70	Nrp	14
Np	714	fo	70
d	3	Nrid	254

7.1.2.2. Cost Estimations

a)

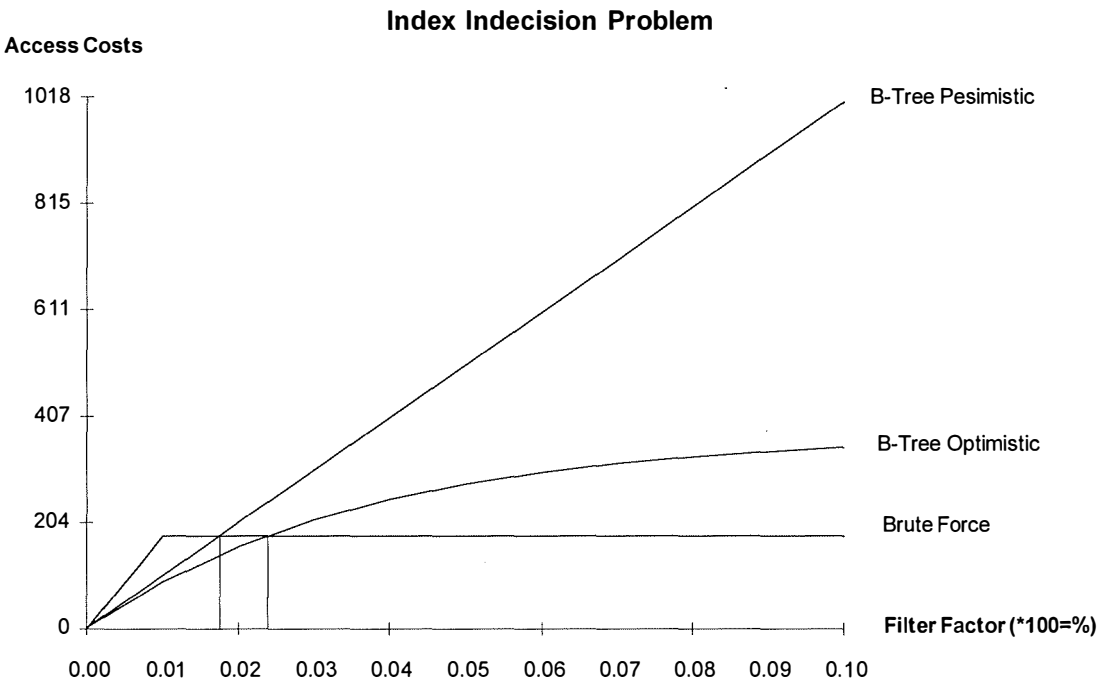
ff	s	k	K	Brute Force	B-Tree Pessimistic	B-Tree Optimistic
0.00	1.00	0	0	0	2	2
0.01	0.99	100	88	179	103	90
0.02	*0.98*	*200*	*154*	*179*	*203*	*158*
0.03	*0.97*	*300*	*205*	*179*	*304*	*209*
0.04	0.96	400	243	179	405	248
0.05	0.95	500	272	179	506	278
0.06	0.94	600	294	179	606	300
0.07	0.93	700	310	179	707	317
0.08	0.92	800	323	179	808	330
0.09	0.91	900	332	179	909	340
0.10	0.90	1000	338	179	1009	348

b)

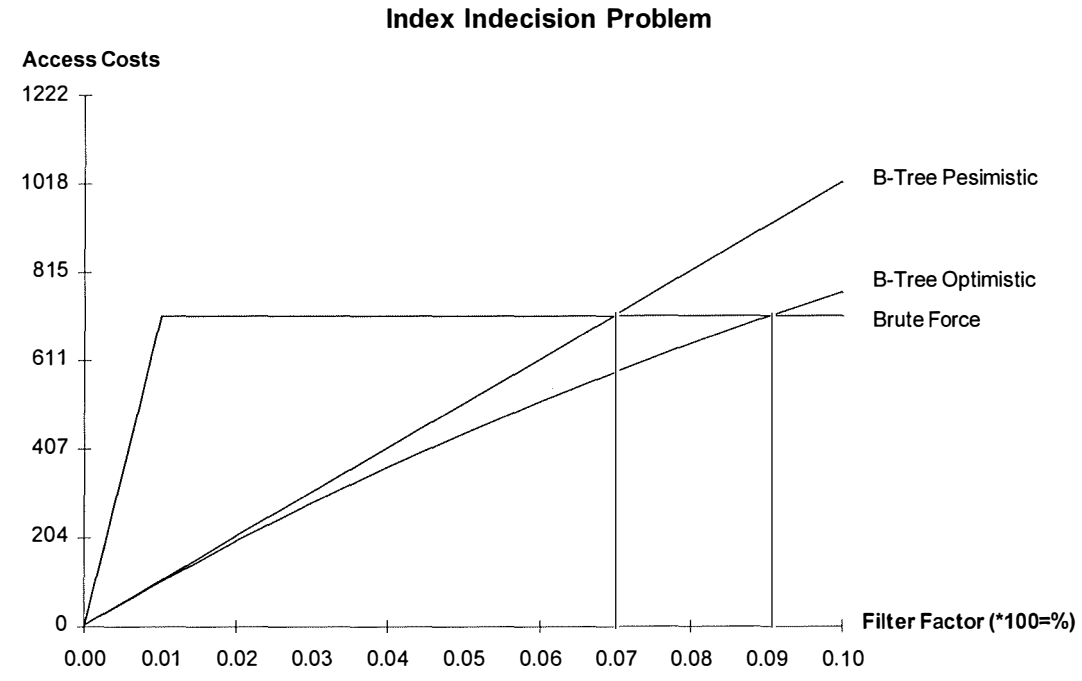
ff	s	k	K	Brute Force	B-Tree Pessimistic	B-Tree Optimistic
0.00	1.00	0	0	0	4	4
0.01	0.99	100	97	714	106	103
0.02	0.98	200	188	714	207	196
0.03	0.97	300	274	714	309	284
0.04	0.96	400	355	714	411	366
0.05	0.95	500	431	714	513	444
0.06	0.94	600	502	714	615	517
0.07	*0.93*	*700*	*569*	*714*	*716*	*585*
0.08	0.92	800	632	714	818	650
0.09	*0.91*	*900*	*690*	*714*	*920*	*714*
0.10	0.90	1000	745	714	1022	767

7.1.2.3. Graphical Representation

a)



b)



7.1.3. Example 3: Varying Fill Rate

7.1.3.1. Input Parameters

a)

Nr	10000	Rs	100
As	20	Ps	2000
fr	***0.50***	Nrp	14
Np	714	fo	70
d	3	Nrid	254

b)

Nr	10000	Rs	100
As	20	Ps	1000
fr	***0.95***	Nrp	14
Np	714	fo	70
d	3	Nrid	254

7.1.3.2. Cost Estimations

a)

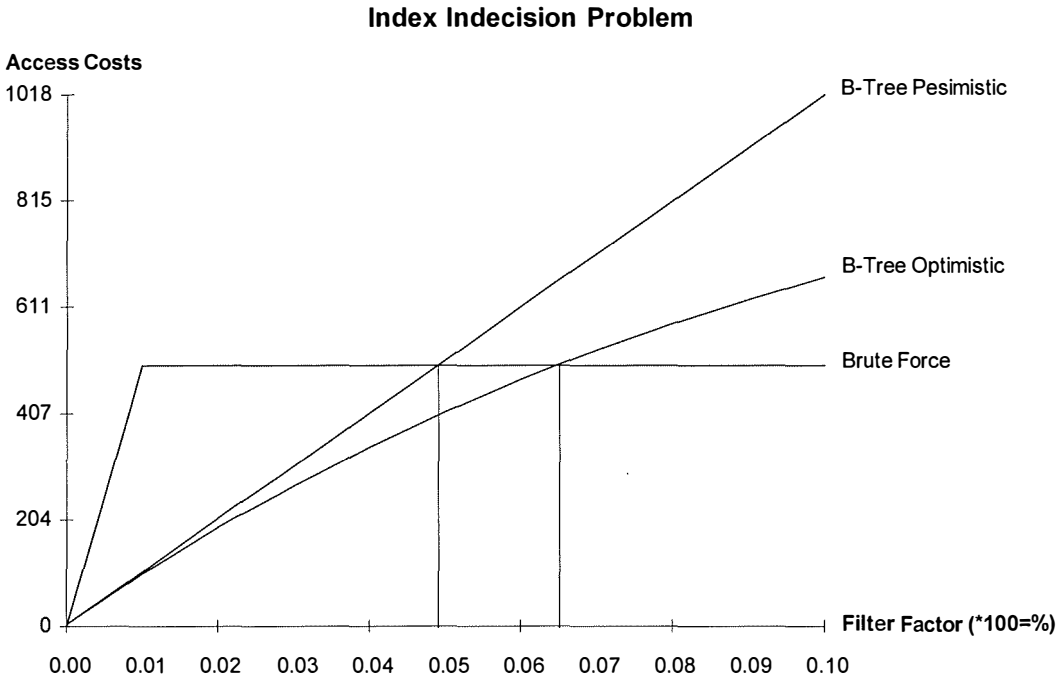
ff	s	k	K	Brute Force	B-Tree Pessimistic	B-Tree Optimistic
0.00	1.00	0	0	0	3	3
0.01	0.99	100	96	500	104	100
0.02	0.98	200	183	500	206	189
0.03	0.97	300	263	500	307	270
0.04	0.96	400	335	500	409	344
0.05	0.95	500	401	500	510	411
0.06	0.94	600	461	500	611	473
0.07	0.93	700	516	500	713	529
0.08	0.92	800	566	500	814	580
0.09	0.91	900	611	500	916	626
0.10	0.90	1000	651	500	1017	668

b)

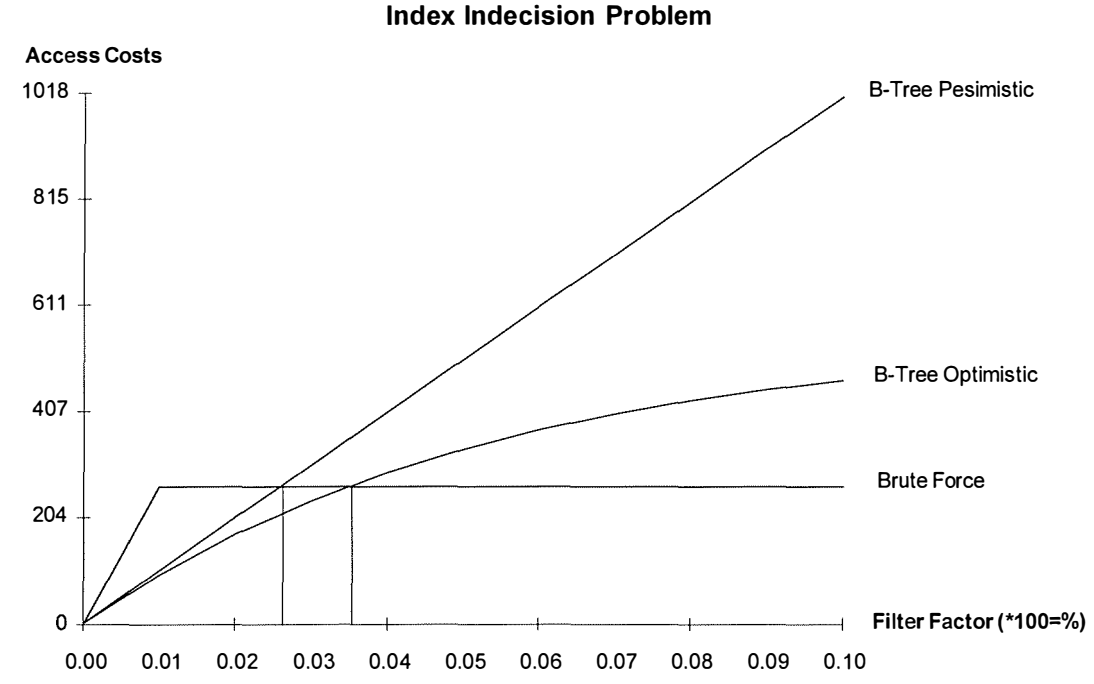
ff	s	k	K	Brute Force	B-Tree Pessimistic	B-Tree Optimistic
0.00	1.00	0	0	0	2	2
0.01	0.99	100	91	263	103	95
0.02	0.98	200	168	263	204	172
0.03	0.97	300	231	263	305	236
0.04	0.96	400	284	263	406	290
0.05	0.95	500	328	263	507	334
0.06	0.94	600	364	263	608	372
0.07	0.93	700	394	263	709	402
0.08	0.92	800	418	263	809	428
0.09	0.91	900	439	263	910	449
0.10	0.90	1000	455	263	1011	467

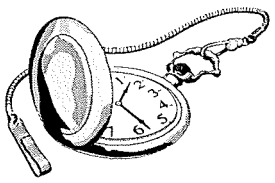
7.1.3.3. Graphical Representation

a)



b)





Chapter 8: References

8.1. Figures

8.2. Relations and Algorithms

8.3. Bibliography

[Baudoin-Meyer 1984] Méthodes de programmation
B. Meyer & C. Baudoin 1984
Eyrolles ISSN 0399-4198

[Bodart 1989] Conception Assisté des systèmes d'Information
(Méthode, Modèles, Outils) 2^{e} édition
F. Bodart & Y. Pigneur 1989
Masson ISBN 2-225-81807-x

[Date 1990] An Introduction To Database Systems
Volumle I (Fifth Edition)
C.J. Date 1990
Addison Wesley ISBN 0-201-52878-9

[Elmasri 1994] Fundamentals of Database Systems
Elmasri & Navathe 1994
Benjamin Cummings ISBN 0-8053-1748-1

[Finkelstein 1988] Physical Database Design for Relational Databases
S. Finkelstein & M. Schkolnick & P. Tiberio
ACM Transactions on Database Systems
Vol. 13 No. 1 March 1988 Pages 91-128

[Hainaut 1986] Conception Assistée des Applications Informatiques
J.L. Hainaut 1986
Masson ISBN 2-225-80730-2

[Hainaut 1994] Base de Données - Clés d'Accès Principales
J.L. Hainaut 2/11/1994
FUNDP BD/9-1 à 9-57

[Mathon 1994] Database Tuning
J.N. Mathon 1994
Memoire Faculté des Sciences Informatiques

[Shasha 1992] Database Tuning (A Principled Approach)
Dennis E. Shasha 1992
Prentice Hall ISBN 0-13-205246-6

[ORACLE 7.0] Oracle 7 Server Documentation

[O'Neil 1994] Database - Principles, Programming, Performace
Partick O'Neil 1994
Morgan Kaufmann ISBN 1-55860-219-4