Part 15: Basics of Physical DB Design

Elmasri/Navathe:

- Chapter 6, “Index Structures for Files”
- Section 16.3, “Physical Database Design in Relational Databases”

I have also used the following sources:

- Silberschatz/Korth/Sudarshan: Database System Concepts, 3rd Edition, McGraw-Hill, 1999. Chap. 11: “Indexing and Hashing” (only parts of Sections 11.1–3, 11.8)
- Kemper/Eickler: Datenbanksysteme (in German), 2nd Ed., Ch. 7, Oldenbourg, 1997.
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Oracle8 Concepts, Release 8.0, Oracle Corporation, 1997, Part No. A58227-01.

Objectives

- You should know the notion of a disk block.
- You should know that there is a buffer (cache) for disk blocks in main memory.
- You should know what an index is and which queries can be evaluated faster using an index.
- You should also know disadvantages of indexes.
- You should have some general impression on the issues involved in physical database design.

All this is treated in more detail in INFSCI 2711.

Overview

1. General Remarks
2. Indexes
3. Other Physical Design Issues

Physical Database Design (1)

- The purpose of physical database design is to ensure that the database system meets the performance requirements.
- So you need to know at least estimates for:
  - How big the tables will be. How the data is distributed.
  - Which application programs exist and which queries and updates they execute.
  - How often each application program is executed.
  - Performance requirements.

    Many interactive programs need fast answers, some seldomly executed programs could run a little longer, and reports may be generated over night.

Physical Database Design (2)

- Since these parameters are difficult to estimate and change over time, you must be prepared to repeat the physical design step from time to time.

  Creating a new index is simple in relational systems. However, if you have to buy entirely new hardware because performance criteria are not met, you have a problem. So you must think about realistic system loads during the design. There are tools for simulating given loads. Don't start using the system before you are sure that it will work.
- Physical design depends on the chosen DBMS.

  You need to know the DBMS, how it works, and its performance tuning parameters very well.

Disks

- Usually, the time spent for disk accesses is much longer than all the computation in main memory.
- Thus, one often counts only how many blocks are read/written.
- A block is a unit of e.g. 2KB which the DBMS always transfers as a whole between main memory and the disk.

  Disks do not allow to access every single byte. To change a byte, the entire block must be read into main memory, the byte changed there, and the block written back to disk.
- Also, reading blocks sequentially from the disk is faster than jumping to far away blocks.

  The disk head must move to another cylinder/track and wait there until the needed sector comes in sight.

Buffering

- Since reading blocks from the disk is much slower than main memory accesses, a copy of the most recently read blocks is kept in main memory (buffer, cache).
- Not all blocks can be kept in main memory, since normally the database is much bigger than the main memory.

  E.g. the DB is several GB, the main memory 256MB.

  The buffer size is an important tuning parameter.
- This makes performance tests difficult: The first time a query is executed it takes much longer than the second time.

  For small databases, soon the entire database will be in main memory, so that disk accesses are only needed to make updates durable (but not for queries).

Overview

1. General Remarks
2. Indexes
3. Other Physical Design Issues

Motivation

- Suppose we have a table with information about customers:

```
Customer(CustNo, First_Name, Last_Name,
         Street, City, State, ZIP, Phone)
```

- Let us assume further that this table is relatively big.
  E.g. 500 000 customers, about 45 MB.
- Now consider the query for the data of a specific customer:

```
SELECT *
FROM   Customer
WHERE  CustNo = 7635
```

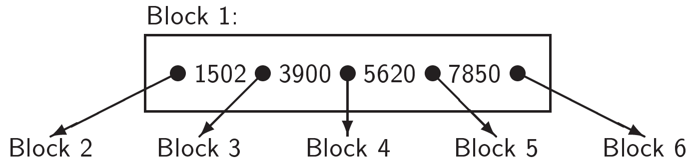
- Normally, the system has to read all rows of the table and check whether CustNo = 7635 ("Full Table Scan").
  A disk will need e.g. 5 sec. for reading 45 MB sequentially.

B$^+$-Trees (1)

- However, DBMS offer a variety of special access structures, which allow to find the requested record without reading the entire file. The most common one is the B$^+$-tree.
  Every modern DBMS contains some variant of B-trees. In addition it may support other specialized index structures.
- B-trees are named after their inventor, Rudolf Bayer.
  Bayer/McCreight: Organization and Maintenance of Large Ordered Indices. Acta Informatica 1(3), 173–189, 1972.
- B*-trees and B$^+$-trees are (normally) synonyms.
- In general, B-trees work like standard binary search trees.
- However, since we anyway need to read a whole block from external memory, the branching factor is much higher than 2.

B$^+$-Trees (2)

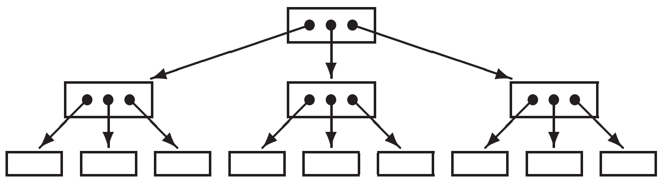
- Suppose you have create an index on Customer(CustNo).
- A B$^+$-tree consists of branch blocks and leaf blocks.
  The branch blocks guide the search for a row:



- If the requested CustNo is $\leq$ 1502, continue in Block 2.
  For CustNo $>$ 1502 and CustNo $\leq$ 3900, go to Block 3.
  ...
  If CustNo $>$ 7850, continue in Block 6.

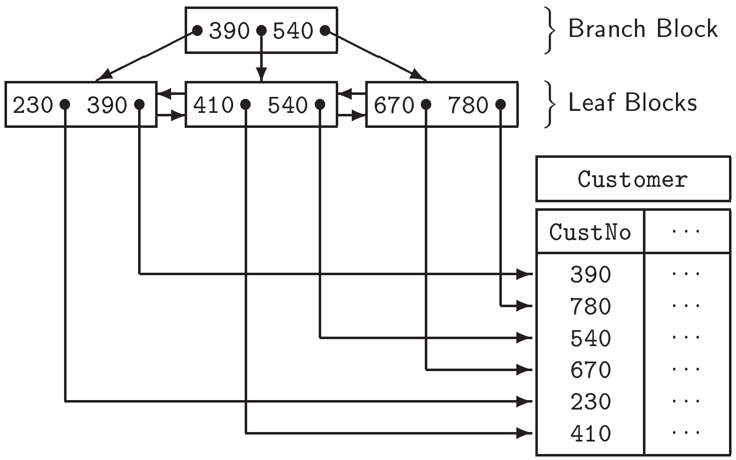
B$^+$-Trees (3)

- The referenced blocks have the same structure, so that a tree is built:



- The blocks in the lowest level ("leaf blocks") contain all occuring customer numbers (in ordered sequence) together with the ROWID of the corresponding Customer row.
  These blocks have a different structure than the branch blocks (see next slide). The topmost block is called "root".

B$^+$-Trees (4)



B$^+$-Trees (5)

- In a B-tree, all leaf blocks must have the same distance (number of edges) from the root. Thus B-trees are balanced.

  This guarantees that we never need to follow a long chain of links in order to access a leaf node. The complexity of searching is $O(\log(n))$, where n is the number of entries.

- In order to make insertion and deletion more efficient, the B-tree guarantees only that each block is at least half full.

  (Except the root, i.e. the topmost block, which might contain only a single customer number.) If we had to keep the blocks completely full, an insertion of one tuple could force us to change every block of the tree. With the relaxed requirement, Insertion/Deletion are possible in $O\big(\log(n)\big)$.

B$^+$-Trees (6)

- Real branching factors are much higher than shown above.
- A database block of 2KB can probably contain about 100 customer numbers and the corresponding ROWIDs. Thus:

Height	Min. Number of Rows	Max. Number of Rows
1	1	100
2	$2 * 50 = 100$	$100^2 = 10\,000$
3	$2 * 50^2 = 5\,000$	$100^3 = 1\,000\,000$
4	$2 * 50^3 = 250\,000$	$100^4 = 100\,000\,000$

- Height 1: Only root, which is at the same time leaf.

  Height 2: Root as branch node, plus leaf blocks, Slide 15-13.

  Height 3: Two levels of branch nodes, see Slide 15-12.

B$^+$-Trees (7)

- So for our `Customer` table with 500 000 entries, the B-tree might have height 3 or height 4.

  If we assume that blocks are in average three quarters full, this would already allow for $75^3 = 421\,875$ rows.

- With a tree of height 3, it takes us 4 block accesses to get the row for a given customer number.

  Three for the index and one for fetching the row from the table with the ROWID we got from the index. If the row was migrated, we would need 5 block accesses in total.

- Even with height 4, the query can be executed in 70ms.

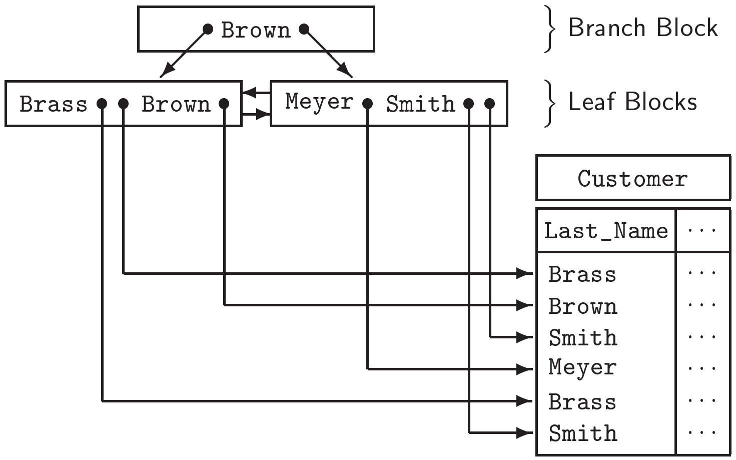
  E.g. each random block access needs 14ms on average.

- One disk can support about 10 such queries per second.

B$^+$-Trees (8)

- Table access via an index profit from caching of disk blocks: E.g. it is very likely that the root node of the index and some part of the next level will be in the buffer.
- Since the height of the B-tree grows only logarithmically in the number of rows, B-trees never become very high.
  Heights greater than 4 or 5 are rare in practice.
- The index on `CustNo` was a unique index — there is only one ROWID for every attribute value.
- B-trees also support non-unique indexes, e.g. on `Last_Name`. Then the leaf blocks contain possibly more than one ROWID for every attribute value.

B$^+$-Trees (9)



B$^+$-Trees (10)

- A high branching factor (and thus a small tree height) is only possible if the data in the indexed column is not too long.
  E.g. if you create an index over a column which contains strings of length 500, the tree will be higher (however, it still grows logarithmically). Oracle requires that the data to be indexed is not larger than about half of the block size.
- It suffices to store a prefix of the actual data in the branch blocks if this prefix already allows to discriminate between the blocks on the next level. (The full version of column data is anyway stored in the leaf blocks.)
  E.g. in the above example, it suffices to store `'B'` in the root node. Oracle uses such a “rear compression”.

Advantages of Indexes

- The selection $\sigma_{A=c}(R)$ is much faster if there is an Index on the attribute A of R.
- Also a join $R \underset{A=B}{\bowtie} S$ can profit from an index on R.A or S.B.
- A B-tree index also helps with range-queries of the form $\sigma_{A<c_1 \wedge A>c_2}(R)$.
- Some queries can even be answered entirely out of the index, without accessing the table itself: E.g. is there a customer with a given number?
  This is important for enforcing key/foreign key constraints.
- Sorting on A might sometimes profit from an index.
  This may speed up `ORDER BY`, `GROUP BY`, `DISTINCT`.

Index Variants (1)

- You can create an index on any column of a table. It does not have to be a key attribute.

  E.g. if you create an index on `Customer.City`, the leaves of the B^+tree will contain values like `'Pittsburgh'`, together with physical pointers to all rows for customers from this city.

- You can declare any number of indexes on the same relation.

  E.g. you can declare one index on `CustNo`, and another index on `City`. If one query has conditions for both attributes, the query optimizer will try to figure out automatically which index(es) to use.

Index Variants (2)

- You can also create an index on more than one column of a table. This will especially speed up queries which specify values for both columns:

```
SELECT *
FROM   Customer
WHERE  First_Name='John' AND Last_Name='Smith'
```

- If the index is a B-tree, it can also be used as an index for the first attribute.

  Since the first attribute of the combination is the main sorting criterion. So here it would probably be better to declare the index on `Last_Name, First_Name`.

  This can be used as an index on `Last_Name`.

Disadvantages of Indexes

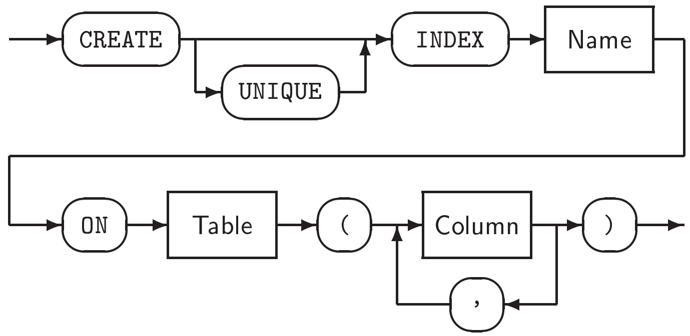
- Indexes use disk space.

  The example index needs 30-50% of the space of the table.

- Queries become faster, but updates become slower because the indexes must be updated, too.
- It is not a good idea to have too many indexes on one relation.
- You should check whether the query optimizer really uses your indexes.
- It makes no sense to declare indexes on small tables.
- A full table scan can be much faster if anyway nearly all blocks must be accessed.

Indexes in SQL (1)

- SQL command for creating an index:



- E.g.: `CREATE INDEX CustInd1 ON Customer(City)`

Indexes in SQL (2)

- The CREATE INDEX command is not contained in the SQL standards but it is supported by most DBMS.
- UNIQUE means, that for every value for the index column there is only one tuple.
    - I.e. the index column is a key. Older SQL versions had no key declarations, so a unique indexes were used.
- ORACLE creates automatically an index for PRIMARY KEY and UNIQUE constraints.
- Command for deleting indexes:

→ DROP → INDEX → Name →

Overview

Other Physical Design Issues

- You can define various storage parameters for tables.
- E.g. if you know the size and the growth rate you can try to ensure that a table is stored sequentially in one chunk of disk space (or only a few big pieces), not scattered over the entire disk (in many small pieces).
- The distribution of tables etc. among different disks is also important for performance.
- Your DBMS may offer other access structures besides B-trees, e.g. hash methods.
    - Oracle allows to cluster tables together (good for joins).
- In exceptional cases, you might want to denormalize tables.