Part 4: B-Tree Indexes

References:

- Elmasri/Navathe: Fundamentals of Database Systems, 3nd Ed., 6. Index Structures for Files, 16.3 Physical Database Design in Relational Databases, 16.4 An Overview of Database Tuning in Relational Systems
- Silberschatz/Korth/Sudarshan: Database System Concepts, 3rd Ed., 11. Indexing and Hashing
- Ramakrishnan/Gehrke: Database Management Systems, 2nd Ed., Mc-Graw Hill, 2000, 8. File Organizations and Indexes, 9. Tree-Structured Indexing, 16. Physical Database Design and Tuning.
- Kemper/Eickler: Datenbanksysteme (in German), Chap. 7, Oldenbourg, 1997.
- Michael J. Corey, Michael Abbey, Daniel J. Dechichio, Ian Abramson: Oracle8 Tuning. Osborne/ORACLE Press, 1998, ISBN 0-07-882390-0, 608 pages, ca. $44.99.
- Oracle 8i Concepts, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76965-01. Page 10-23 ff: "Indexes"
- Oracle 8i Administrator's Guide, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76956-01. 14. Managing Indexes.
- Oracle 8i Designing and Tuning for Performance, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76992-01. 12. Data Access Methods.
- Oracle 8i SQL Reference, Release 2 (8.1.6), Oracle Corp., 1999, Part No. A76989-01. CREATE INDEX, page 7-291 ff.
- Oracle8 Administrator's Guide, Release 8.0, Oracle Corp., 1997, Part No. A58397-01. Appendix A: "Space Estimations for Schema Objects".
- Gray/Reuter: Transaction Processing, Morgan Kaufmann, 1993, Chapter 15.

Objectives

After completing this chapter, you should be able to:

- write a short paragraph about what indexes are.
- explain the B-tree data structure.
- decide whether a given index is useful for a given query, select good indexes for an application.
- explain why indexes have not only advantages.
- enumerate input data about the application that is necessary for physical database design.
- write `CREATE INDEX` statements in Oracle SQL.

Overview

Motivation (1)

- Consider a table with information about customers:

  CUSTOMERS(CUSTNO, FIRST_NAME, LAST_NAME, STREET, CITY, STATE, ZIP, PHONE, EMAIL)

CUSTOMERS			
CUSTNO	FIRST_NAME	LAST_NAME	...
1000001	John	Smith	...
1000002	Ann	Miller	...
1000003	David	Meyer	...
⋮	⋮	⋮	⋮

- Assume further that there are 2 Million customers (rows in the table).

Motivation (2)

- Suppose that a specific customer record is queried:

```
SELECT *
FROM   CUSTOMERS
WHERE  CUSTNO = 1000002
```

- If there are no special access structures, the query is executed with a full table scan:

```
for each row C in CUSTOMERS do
    if C.CUSTNO = 1000002 then print C; fi
od
```

- I.e. all 2 Million rows are read from the disk.

Motivation (3)

- Average lengths: `CUSTNO` 5, `FIRST_NAME` 7, `LAST_NAME` 7, `STREET` 20, `CITY` 10, `STATE` 2, `ZIP` 5, `PHONE` 10, `EMAIL` 20.
- Then an average row needs needs 100 Byte.

  86 Byte for the data, 9 for the lengths, 3 for the row header, and 2 for the row directory entry.
- 17 rows fit into a 2K block (with `PCTFREE=10`).

  2048 Byte (block size) − 90 Byte (block header) − 205 Byte (space reserve) = 1753 Byte. 1753/100 = 17.5.
- At least 117648 blocks, i.e. 230 MB, are needed.

  Because of deletions, blocks might in fact be utilized to less than their full capacity. Then the table needs more blocks.

Motivation (4)

- Even if the entire table is stored in one extent of contiguous blocks, a full table scan will need about 12 seconds.

  Assuming that the disk reads 20MB/s in a sequential scan.

- A response time of 2 seconds is the maximum which does not hinder users during interactive work.
- Some authors even require a “subsecond response time”.
- Full table scans of big tables do not profit from caching.

Motivation (5)

- One must consider not only a single query run in isolation, but the entire system load.
- Suppose that 100 employees enter orders in parallel, and for each order the customer data must be accessed.
- Since the DBMS (using the full table scan) can process only five queries per minute, each employee can only enter one order every 20 minutes.

  If two full table scans run interleaved, the head has to move back and forth, and the total time will be more than double of the time needed for a single full table scan.

Motivation (6)

- DB systems offer special data structures (indexes) that allow to find all rows with a given attribute value without reading and checking every row.
- Consider how useful an index is in a book:
  It is the only way to find all occurrences of a keyword without reading the entire text.
- An typical B-tree index in a DBMS is very similar:
  A (sorted) list of all occurring values for a specific column together with references to the rows that contain the respective value.

Motivation (7)

- Indexes are sometimes called “inverted files”:
  - ◇ The heap file that contains the table data supports the mapping from ROWIDs to attribute values.

    A file containing a text document maps positions in the text to words (i.e. it defines what is the first word, the second word, and so on).
  - ◇ An index supports the mapping from values for a specific attribute (or attribute combination) to ROWIDs.

    An index to the document maps words to positions in the text.

Motivation (8)

- In order to solve the example query, the DBMS will first search the index over CUSTOMERS(CUSTNO) (e.g. 4 block accesses).
- In the index entry for the given customer number 1000002, it will find the ROWID of the requested CUSTOMERS-row.
- Finally, it reads the row with this ROWID from the CUSTOMERS table (1 block access, 2 if row migrated).
- In total, the query is executed in about 50 msec.

Physical Data Independence (1)

- The DBMS automatically creates indexes for keys. Other indexes must be explicitly created:

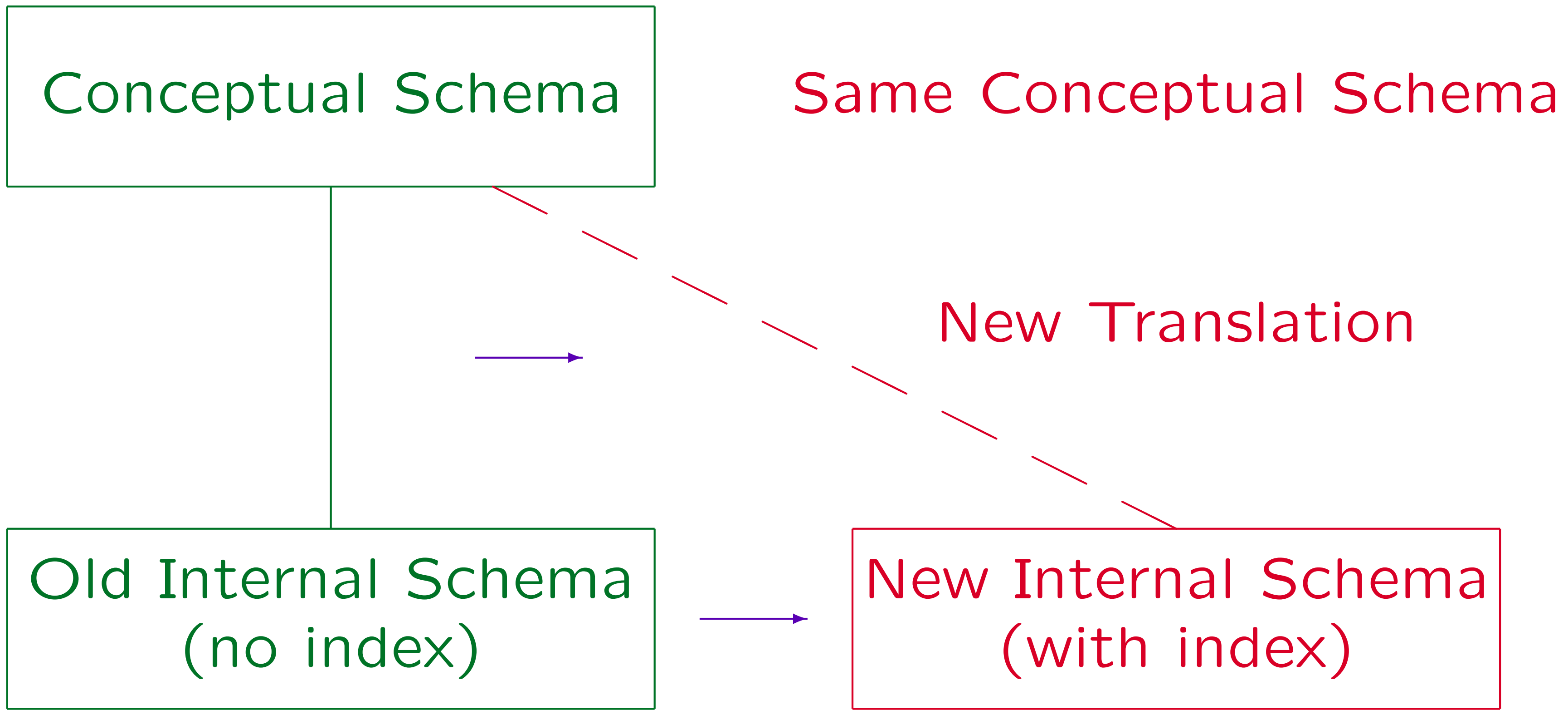
```
CREATE INDEX IND_CUST1 ON CUSTOMERS(LAST_NAME)
```

- The query optimizer will now use the index, whenever it thinks that this would be advantageous, e.g. for:

```
SELECT *
FROM   CUSTOMERS
WHERE  LAST_NAME = 'Jones'
AND    CITY = 'New York'
```

- The query does not have to be changed to use the index.

Physical Data Independence (2)


Conceptual Schema
Same Conceptual Schema
New Translation
Old Internal Schema (no index)
New Internal Schema (with index)

Overview

B^+-Trees: Overview

- The usual data structure for an index is the B^+-tree.

  Every modern DBMS contains some variant of B-trees plus maybe other index structures for special applications.

- B-trees are named after their inventor, Rudolf Bayer.

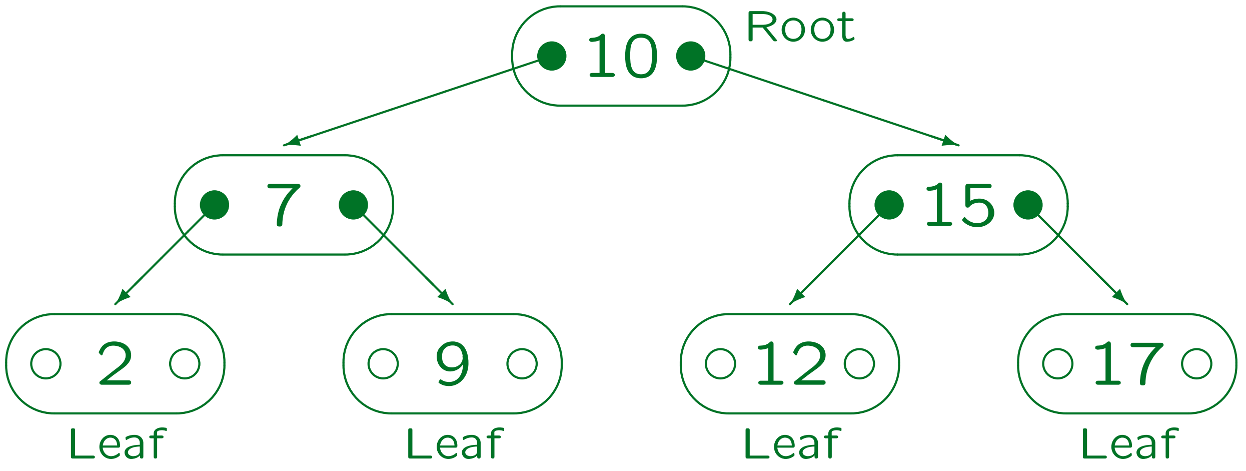
  Bayer/McCreight: Organization and Maintenance of Large Ordered Indices. Acta Informatica 1(3), 173–189, 1972.

- B^*-trees and B^+-trees are (normally) synonyms.

  They differ from standard B-trees in having all data in the leaf blocks (see below).

- In general, B-trees work like binary search trees (should be known). But they are not the same!

Binary Search Trees



- For every node N of the tree:
  - ◇ The left subtree of N contains only values smaller than the value in N.
  - ◇ The right subtree of N contains only values greater than the value in N.

B-Trees vs. Binary Trees

- In a B-tree, the branching factor (fan out) is much higher than 2. A whole block must be read from the disk: All information in it should be used.
- Normal binary trees can degenerate to a linear list. B-trees are balanced, so this cannot happen.

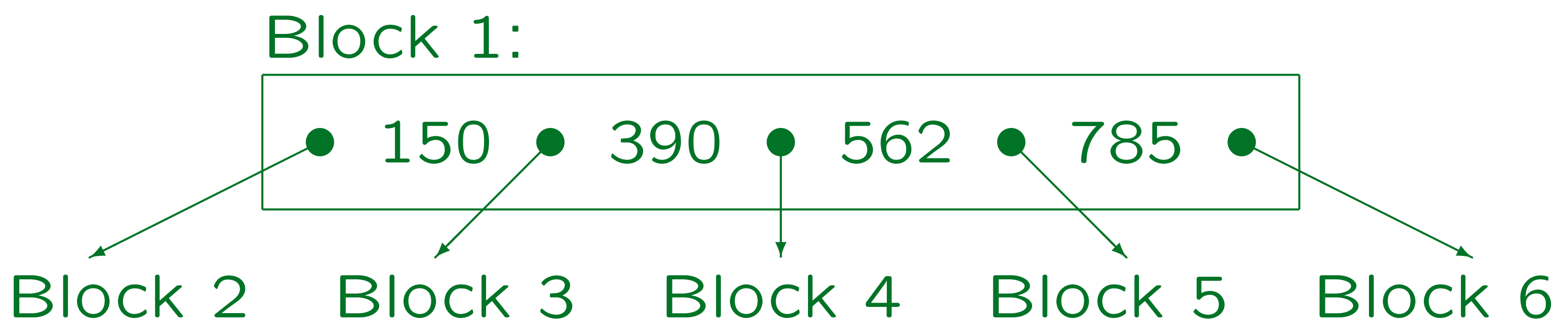
  E.g. if values are inserted in ordered sequence, they are always inserted to the right in a normal binary tree.
- In a B^+-tree (not in a B-tree) the values in inner nodes (non-leaves) are repeated in the leaf nodes.

  The tree height might decrease, since the pointer to the row is needed only in the leaf nodes. Also one can easily get a sorted sequence.

B^+-Trees: Structure (1)

- 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$150, continue in Block 2.

  For `CUSTNO` $>$ 150 and `CUSTNO` $\leq$ 390, go to Block 3.

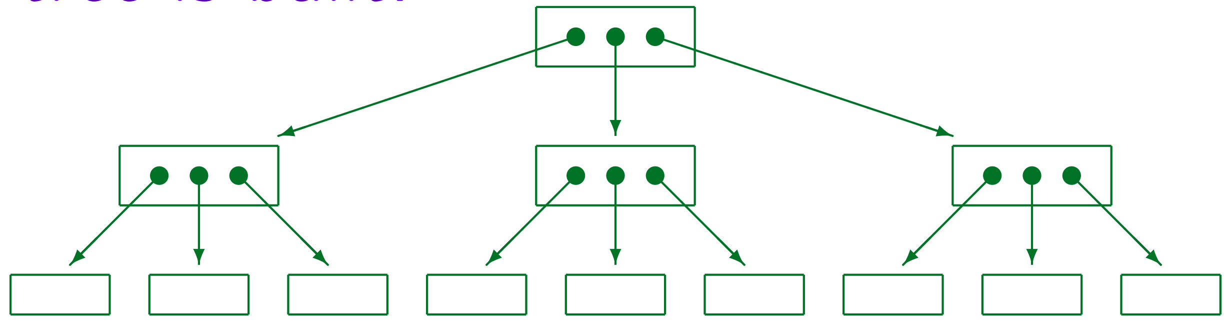
  ...

  If `CUSTNO` $>$ 785, continue in Block 6.

Because of the limited space, 3-digit customer numbers are used.

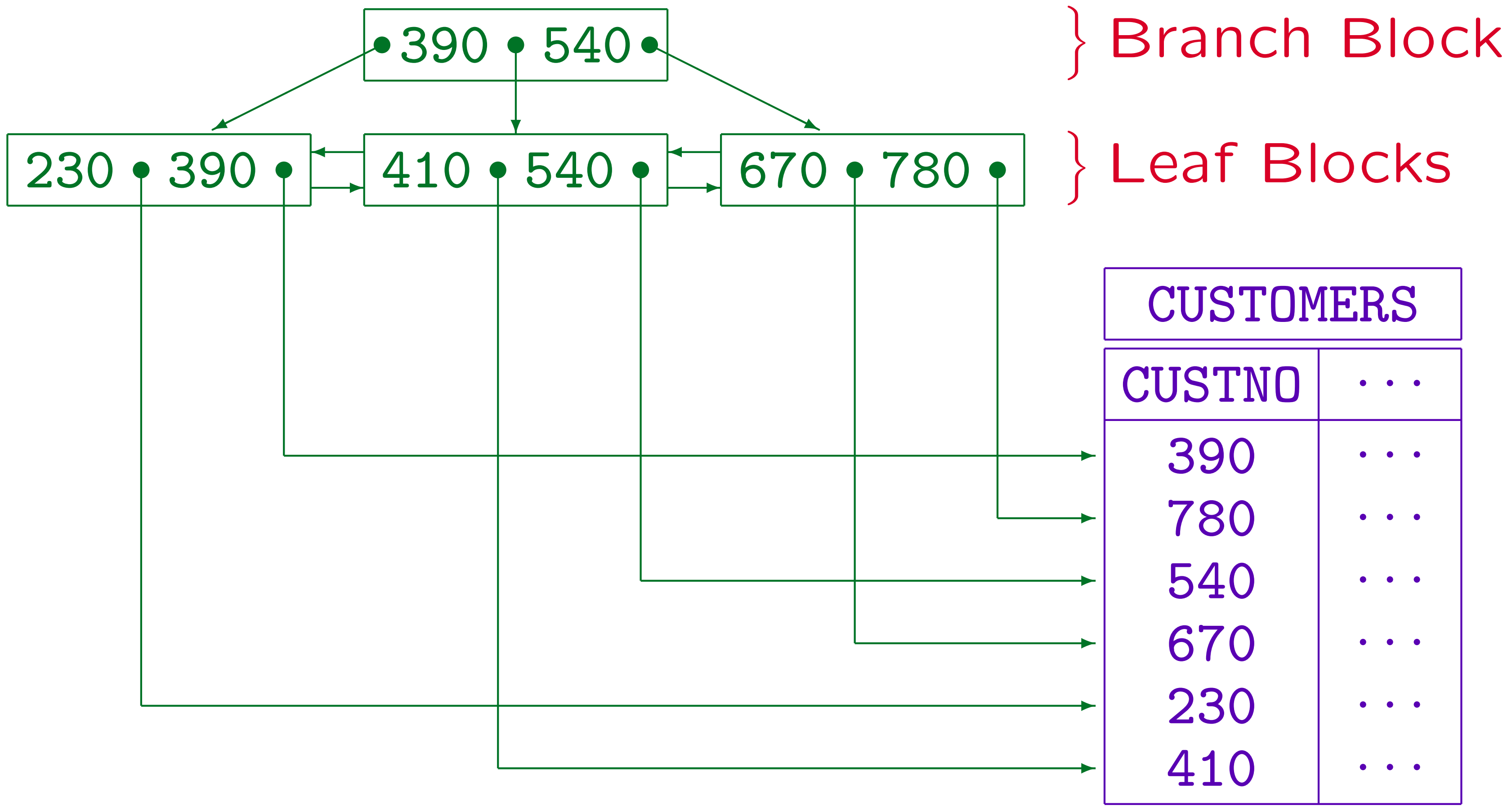
B^+-Trees: Structure (2)

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



- The blocks in the lowest level ("leaf blocks") contain all occurring customer numbers (in ordered sequence) together with the address ("ROWID") of the corresponding `CUSTOMERS` row.

B^+-Trees: Structure (3)



B^+-Trees: Structure (4)

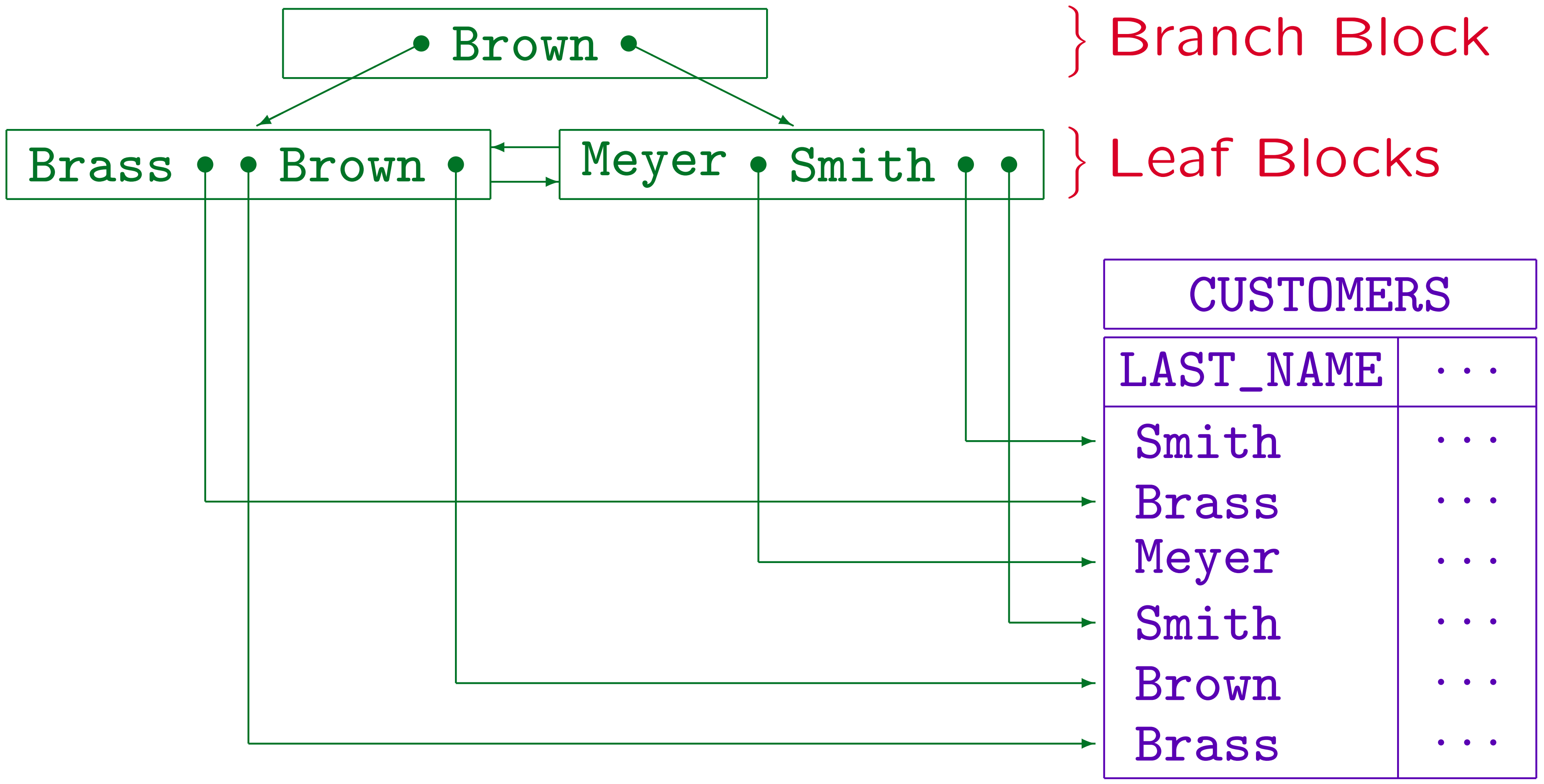
- The index on CUSTNO was a unique index — there is only one row for every value (CUSTNO is a key).
- B-trees also support non-unique indexes, e.g. on LAST_NAME. Then the leaf blocks can contain more than one row address for the same column value.

  Although LAST_NAME is not a key of the table, it is sometimes called the "search key" of the B-tree.

- It is also possible to create B-trees over the combination of two or more columns.

  Then the indexed values are basically the concatenation of column values (with e.g. a separator character).

B^+-Trees: Structure (5)



B^+-Trees: Structure (6)

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

  This ensures that the chain of links which must be followed in order to access a leaf node is never long. For B-trees, the complexity of searching (tree height) is $O(\log(n))$, where n is the number of entries.

- Nodes in the same tree can contain differently many values, but each node must be at least half full.

  (Except the root, which might contain only a single customer number.) If the blocks had to be completely full, an insertion of one tuple could require a change of every block in the tree. With the relaxed requirement, insertion/deletion are possible in $O(\log(n))$.

B^+-Trees: Insertion (1)

- Starting at the root block, one navigates through the branch blocks down the tree to find the leaf block that must contain the new entry.

  This works in exactly the same way as a search whether the value already exists in the tree. However, one must remember the path through the tree, because one might have to follow it back later in the algorithm.

- If the leaf block still has space, the entry is inserted and we are done.
- Otherwise the leaf block is evenly split into two blocks in order to make space for the new entry.

B^+-Trees: Insertion (2)

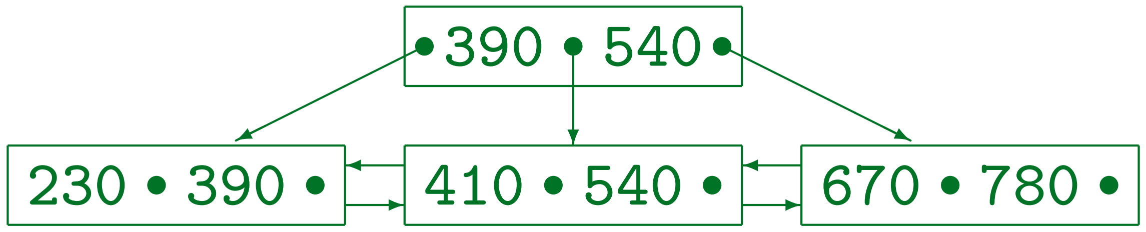
- Since the block was completely full before the split, the resulting blocks are at least half full as required.
- The split requires a new entry in the branch block above (there are now two pointers to leaf blocks instead of one).
- If that branch block has space: Insert entry. Done.
- Otherwise the branch block is split into two, and the insertion goes up to the parent branch block.

  Finally, if the root node is split into two blocks, they become standard branch blocks. A new root is constructed above them (containing only a single entry and two pointers).

B^+-Trees: Insertion (3)

Exercise:

- Consider again the B^+-tree from Slide 4-20:



- Assume that each node can contain at most two values (and must contain at least one value).
- Insert the value 123.
- Give an example for a value that can now be inserted without splitting any further nodes.

B^+-Trees: Performance (1)

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

Height	Min. Num. Rows	Max. Num. Rows
1	1	100
2	$2 * 50 = 100$	$100^2 = 10000$
3	$2 * 50^2 = 5000$	$100^3 = 1000000$
4	$2 * 50^3 = 250000$	$100^4 = 100000000$

Height 1: Only root, which is at the same time leaf.
Height 2: Root as branch node, plus leaf blocks, as on Slide 4-20.

B^+-Trees: Performance (2)

- For the CUSTOMERS table with 2000000 entries, the B-tree will have height 4.

  Height 5 would require at least $2 * 50^4 = 12.5$ mio rows.

- A tree of height 4 requires 5 (possibly 6) block accesses to get the row for a given customer number.

  Four for the index and one for fetching the row from the table with the ROWID obtained from the index. In case of a "migrated row" (should be seldom), 2 block accesses are needed for fetching the row.

- The query can be executed in 50ms. One disk can support 14 such queries per second (70% load).

  Current disks need about 10ms per random block access.

B^+-Trees: Performance (3)

- Table accesses 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 5 or 6 are rare in practice.

B^+-Trees: Performance (4)

- A high branching factor (and thus a small tree height) is possible only if the data in the indexed column is not too long.

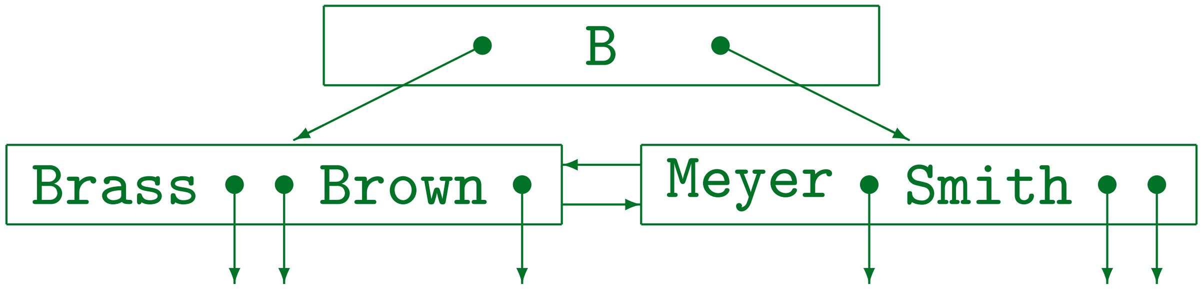
  E.g. an index over a column that contains strings of length 500 will need a higher tree (which still grows logarithmically). In Oracle, the indexed values may not be larger than about half of the block size.

- It suffices to store a prefix of the actual data in the branch blocks if this prefix already allows discrimination between the blocks on the next level.

  The full version of column data is anyway stored in the leaf blocks. Therefore, no information is lost.

B^+-Trees: Performance (5)

- E.g. in the above example, it suffices to store `'B'` in the root node.



- Oracle uses such a "rear compression".
- Furthermore, a "front compression" is possible by storing a common prefix only once.

Overview

Applications of Indexes (1)

- The selection $\sigma_{A=c}(R)$ is much faster if there is an index on the attribute A of R.
- E.g. consider the following query:

```
SELECT *
FROM   CUSTOMERS
WHERE  LAST_NAME = 'Smith'
AND    CITY = 'New York'
```

- If there is an index on LAST_NAME, the system first retrieves the rows with the given last name (via the index), and then check the city for each such row:

$$\sigma_{\texttt{CITY='New York'}}(\sigma_{\texttt{LAST\_NAME='Smith'}}(\texttt{RESULTS})).$$

Applications of Indexes (2)

Multiple Indexes on One Table:

- There can be several indexes on the same table.
- For instance, there could be an index on the column `LAST_NAME` and another index on the column `CITY`.
- Then there are several alternatives for evaluating the query, and the optimizer has to decide for the most efficient way.
- Since there are less people named Smith than living in New York, it might be best to use the index on `LAST_NAME` (as before).

Applications of Indexes (3)

ROWID Intersection:

- It is also possible to apply both indexes:
  - ◇ The index on `CUSTOMERS(LAST_NAME)` is used to get the ROWIDs of rows with `LAST_NAME = 'Smith'`.
  - ◇ The index on `CUSTOMERS(CITY)` is used to get the ROWIDs of rows with `CITY = 'New York'`.
  - ◇ Then the two sets of ROWIDs are intersected.

    This can be very efficient when one ensures that the ROWIDs in each index entry are sorted.
  - ◇ Only rows satisfying both conditions are retrieved from the table.

Applications of Indexes (4)

Indexes on Attribute Combinations:

- One can create indexes on attribute combinations:

```
CREATE INDEX CUST2 ON CUSTOMERS(LAST_NAME,
                                CITY)
```

- Then the entries in the B-tree are more or less the concatenation of `LAST_NAME` and `CITY`.
- A query that contains values for both columns can be very efficiently executed with such an index.

- The index can also be used for queries which specify only a `LAST_NAME`, since this is a prefix of the combined entry (see next slide).

Applications of Indexes (5)

Using Only a Prefix of the Index Attributes:

- An index on an attribute combination (A, B) can be used for queries that specify only a value for A: This is the main sorting criterion for the index entries.

  I.e. the entries with the same A-value are stored in contiguous index entries, often in the same disk block.

- It cannot be used if only a value for B is known.
- A pure index on A would be slightly better, since the tree height is smaller.

  Furthermore, the returned ROWIDs are now not sorted. But since indexes also cost something, one would seldom have both indexes.

Applications of Indexes (6)

Using Indexes for Range Queries:

- A B-tree index can also be used for range queries (since it contains the entries in sorted order).

  But indexes only help if only a small fraction of the rows is selected.

- An index on CUSTOMERS(ZIP) would be useful here:

```
SELECT *
FROM   CUSTOMERS
WHERE  ZIP BETWEEN 15000 AND 15999
```

- An index on INVOICES(AMOUNT) can be used here:

```
SELECT *
FROM   INVOICES
WHERE  AMOUNT >= 10000
```

Applications of Indexes (7)

Exercise:

- Suppose the CUSTOMERS table has also a (redundant) field for the total sales. Now consider this query:

```
SELECT *
FROM   CUSTOMERS
WHERE  CITY = 'PITTSBURGH'
AND    TOTAL_SALES > 1000
```

- Which of the following indexes is best?
  - ☐ An index on CITY
  - ☐ An index on TOTAL_SALES
  - ☐ An index on (CITY, TOTAL_SALES)
  - ☐ An index on (TOTAL_SALES, CITY)

Applications of Indexes (8)

String Pattern Matching with `LIKE`:

- This query can profit from an index on `LAST_NAME`:

```
SELECT *
FROM   CUSTOMERS
WHERE  LAST_NAME LIKE 'Br%'
```

- The query is actually equivalent to the range query:

```
SELECT *
FROM   CUSTOMERS
WHERE  LAST_NAME >= 'Br' AND LAST_NAME < 'Bs'
```

- However, this only works if a prefix of the string is known. E.g. `LIKE '%rown'` needs a full table scan.

Applications of Indexes (9)

Applications for Joins:

- A join $R \underset{A=B}{\bowtie} S$ can use an index on $R.A$ or $S.B$.
- E.g. consider this query:

```
SELECT C.LAST_NAME
FROM   INVOICES I, CUSTOMERS C
WHERE  I.AMOUNT >= 10000
AND    C.CUSTNO = I.CUSTNO
```

- First, large invoices `I` are located with an index on `INVOICES(AMOUNT)`. Then `I.CUSTNO` is known, and the index on `CUSTOMERS(CUSTNO)` can be applied to get `C`.

  This is only one possibility to evaluate the given query.

Applications of Indexes (10)

Index-only Query Execution Plans:

- Some queries can be answered entirely from the index without fetching the rows from the table.

  Since retrieving the rows by ROWID costs 1–2 block accesses per row, eliminating this step can be a big achievement.

- E.g. here it suffices to count the ROWIDs from the index on `CUSTOMERS(ZIP)`:

```
SELECT COUNT(*)
FROM   CUSTOMERS
WHERE  ZIP = 15260
```

- Often useful: Indexes on attribute combinations.

Applications of Indexes (11)

Applications for ORDER BY:

- Consider this query:

```
SELECT   *
FROM     INVOICES
WHERE    AMOUNT >= 10000
ORDER BY AMOUNT
```

- A B-tree produces the rows in the sort order of the indexed attribute. If the index on AMOUNT is used to evaluate the condition, the ORDER BY is for free.
- Even without condition on the attribute, one could do a “full index scan” for the purpose of sorting. But this pays off only in index-only evaluation plans.

Applications of Indexes (12)

Using Indexes for GROUP BY/DISTINCT:

- An index stores for each attribute value the set of ROWIDs. Therefore, DISTINCT and GROUP BY are "precomputed" if a fitting index is used.
- E.g. this query can be answered from an index on LAST_NAME, CITY (without accessing the table):

```
SELECT DISTINCT CITY
FROM   CUSTOMERS
WHERE  LAST_NAME = 'Smith'
```

Applications of Indexes (13)

Indexes and Keys:

- When a key (`PRIMARY KEY` or `UNIQUE`) is declared, most DBMS (including Oracle) automatically create an index on the key attributes.
- Indexes can be declared as `UNIQUE`, e.g.

```
CREATE UNIQUE INDEX CUSTKEY ON CUSTOMERS(CUSTNO)
```

- Then any attempt to insert two rows with the same value for `CUSTNO` will fail.

  In this way, the index enforces the key constraint. In most DBMS (e.g. Oracle), unique indexes are older than keys in the `CREATE TABLE`.

Applications of Indexes (14)

Indexes and Foreign Keys:

- Consider the foreign key:

  `INVOICES.CUSTNO` $\longrightarrow$ `CUSTOMERS`.

- Two operations might violate this constraint:
  - ◇ If a row is inserted into `INVOICES`, the system will use the index on `CUSTOMERS(CUSTNO)`.

    The table `CUSTOMERS` does not have to be accessed. This index always exists, because it is needed to implement the key.

  - ◇ If a row in `CUSTOMERS` is deleted, an index on `INVOICE(CUSTNO)` would be helpful (or else a full table scan is needed).

Applications of Indexes (15)

Indexes and Null Values:

- In Oracle, rows with a null value in the indexed attribute are not represented in the index.

  If the index is on a combination of attributes, only rows having a null value in all of these attributes are excluded.

- Sometimes this prevents an index-only access plan.

- But in this way, an index of a column with many null values is much smaller than the entire table.

  E.g. one must keep track of a small number of rows in a big table. Solution: Create a new column and set it everywhere null except in this small number of rows. An index over the new column will contain only the ROWIDs of the given rows.

Summary: Index Advantages

- The selection $\sigma_{A=c}(R)$ is much faster if there is an Index on the attribute A of R.
- A join $R \underset{A=B}{\bowtie} S$ can use an index on $R.A$ or $S.B$.
- A B-tree index also helps with range-queries of the form $\sigma_{A<c_1 \wedge A>c_2}(R)$ (incl. special cases of `LIKE`).
- Some queries can even be answered entirely out of the index, without accessing the table itself.

  This is important for enforcing key/foreign key constraints.
- Sorting might sometimes profit from an index.

  This may speed up `ORDER BY`, `GROUP BY`, `DISTINCT`.

Index vs. Full Table Scan (1)

- Using an index is only advantageous if the index selects only a small fraction of all rows.
- Otherwise a full table scan might be better.
- “Oracle8i Designing and Tuning for Performance” mentions as a general guideline that indexes are useful when a table is often queried for less than 2–4% or the rows.

  Under certain assumptions, see page 12-2 of the manual.

Index vs. Full Table Scan (2)

- A sequential scan of all contiguously stored blocks is much faster than random accesses, e.g. 2500 blocks of 8K can be read sequentially per second, but only 100 randomly.

  Assuming 20 MB/s sequentially, 10ms/block randomly.
  With 2K blocks, 10 000 blocks/s could be read sequentially.

- It is expected that this difference will grow in future.
- Thus a selective access becomes interesting when less than $100/2500 = 4\%$ of the blocks is accessed.

  This ignores the CPU cost for checking each of the blocks read in the sequential scan. But the CPU is anyway much faster than the disk.

Index vs. Full Table Scan (3)

- If, however, the ROWIDs are sorted by block numbers, the accesses are not random and take never more time than sequential reading.

  Ignoring the overhead for the index lookup and the management of a series of single block accesses.

- The example disk (ST318405) has on average 471 sectors of 512 byte per track, i.e. 29 blocks of 8K.
- So a track can be skipped only if gaps of 29 or more blocks occur.

  Otherwise using ordered ROWIDs can at least not be faster than half the time required for a full sequential read.

Index vs. Full Table Scan (4)

- In Oracle, the ROWIDs for a single index entry are ordered (e.g. `WHERE LAST_NAME = 'Smith'`).
- However, for range queries, joins, `LIKE`, `ORDER BY`, `GROUP BY`, `DISTINCT` several index entries are accessed, and the ROWIDs are returned in random order.

  Except for joins, the ROWIDs are returned in the order of the index attribute (only for the same attribute value, the ROWIDs are ordered).
- The DBMS could sort the ROWIDs before accessing the base table, but if the number of ROWIDs is large, this is also an expensive operation.

  Oracle sorts ROWIDs only for intersecting them.

Index vs. Full Table Scan (5)

- The selectivity of the search condition must be seen relative to the number of rows per block.

  It is a big mistake to assume that when less than 2–4% of the rows are requested, an index is always advantageous.

- E.g. if every block contains 100 rows, and 3% of the rows are requested, it is very likely that $\geq 80\%$ of the blocks must be accessed.

  Assuming that the rows are randomly distributed over the blocks.

- If, however, only one row fits in each block, accessing 3% is even advantageous with completely random accesses.

Index vs. Full Table Scan (6)

- If the ROWIDs are not sorted by block number, and there are not enough buffer frames, it can happen that the same block must be read multiple times.
- E.g. consider a table with 100000 rows stored in 1000 blocks (of 8 KB each).
- Suppose that 10% of the rows are selected.
- The full table scan reads 8 MB in 0.4 seconds.

  Assuming 20 MB/s for a sequential read.
- Using an index, the query runs 80 seconds!

  Assuming a small buffer cache and unordered ROWIDs, see next slide.

Index vs. Full Table Scan (7)

- If the ROWIDs are not ordered, 10 000 random block accesses will be needed.

  10% of 100 000 rows are selected, each needs a block access.

- If the buffer cache has space only for 200 blocks, and accesses are at random, only 20% can be answered from the cache, so 8 000 real random disk accesses will be needed.

- Thus, if the query is evaluated with the index, it runs 80 seconds (not counting the index accesses).

Index vs. Full Table Scan (8)

- Source of the problem is that ROWIDs are often returned from the index in attribute order, not in ROWID order.
- Some systems avoid these problems by trying to keep the file sorted by the indexed attribute ("clustered index").

  Difficult if there are insertions/deletions, stable ROWIDs are needed, and blocks should be stored contiguously.
- Obviously, there can be only one clustered index for a table.

Index vs. Full Table Scan (9)

- Even if the system does not enforce such clustering (like Oracle), it might measure possibly existing clustering for the purpose of query optimization (`ANALYZE TABLE`).

  E.g. if the rows to a table were inserted in the sort order of the index attribute, the index is much more effective than if the storage order of the rows is random.

- However, Oracle has clusters (see next chapter).
- In the meantime, Oracle also has Index-Organized Tables (IOTs), which are very similar to a clustering index: There, the entire row is stored in the B-tree.

Index vs. Full Table Scan (10)

- "Index only" queries do not have such problems: The index entries are always stored in sorted order.

  Although the leaf blocks are not necessarily stored in contiguous locations on the disk.

- E.g. for an index on (`LAST_NAME, CITY`), it is likely that the index entries for the same name are stored in one block (maybe two).

- Using an index for a key attribute is nearly always useful: There will be only a single result row.

  Except for very small tables. Using a key attribute index iteratively in a join might be worse than other join methods (see Chapter 6).

Index vs. Full Table Scan (11)

Indexes and Small Tables:

- For small tables, a full table scan is almost always faster.

  Even if the index consists only of a single block, two random block accesses are needed (one for the table itself). This takes e.g. 20ms. About 50 contiguous blocks of 8K can be read in the same time (one random seek, latency time, and one and a half tracks of data). Of course, caching might change the picture (the index block will remain in memory).

- If a table has only a few blocks (say 5–10), use only indexes to enforce the key constraints.

  In Oracle one can request that specific tables are cached. This is good for small tables that are often accessed.

Disadvantages of Indexes (1)

Indexes use Disk Space:

- Consider the index on `CUSTOMERS(CUSTNO)`. For 2 Million rows, already 26667 leaf blocks are needed.

  Assuming that one block can contain 100 entries, and blocks are on average 75% full. In addition about 361 branch blocks are needed.

- This index has about one quarter of the table size.

  It is advantageous here that the indexed column is relatively short in comparison with the row length. Otherwise the index would be larger.

- If more than one index is created for a table, the indexes might need more space than the table itself.

Disadvantages of Indexes (2)

Slower Updates:

- Queries become faster, but insertions and deletions become slower because the indexes must be updated, too.

  The insertion into an index basically takes the same time as the insertion into a heap file. Thus, already a single index on a table roughly doubles the time needed for an insertion.

- Updates only become slower if an indexed column is updated. It might be a good idea to avoid indexes on columns which are very frequently updated.

  There is a tradeoff: The index might be needed for queries.

Disadvantages of Indexes (3)

Query Optimization Takes Longer:

- If there are many indexes that can be used for a given query, query optimization may take longer.
- However, normally the additional time spent for query optimization is amortized over many executions of the query.

  Therefore, it is important that the system detects that the same query is executed again.
- In extreme cases, the time limit built into most query optimizers might lead to a bad query evaluation plan when there are too many possibilities.

Summary: Index Selection (1)

- For every key (`PRIMARY KEY` and `UNIQUE`) a unique index is required that enforces the constraint.

  The DBMS automatically creates these indexes.

- Small tables (e.g. 5–10 blocks) should not have any other indexes.

- Consider creating indexes on foreign key columns if there are deletions on the referenced table.

- Tables should not have too many indexes.

  E.g. not more than 4–6. The more frequent updates on the table are (relative to queries), the fewer indexes are acceptable. Especially one should avoid indexes on frequently updated columns.

Summary: Index Selection (2)

- Indexes should only be created if they speed up query evaluation.

  A necessary condition for this is that the optimizer chooses to use them in query execution plans. Check this.

- The condition to be evaluated with the index should return only a small percentage of the rows.

  More important than the percentage of the rows is the percentage of blocks: This depends on row size and clustering effects.

- Thus, the indexed column must contain many different values. E.g. not only `male`, `female`.

  It might confuse the optimizer if values are not equally distributed.

Summary: Index Selection (3)

- If ROWIDs are sorted by blocks, selecting 10-20% of the blocks might still be useful.

  If the query contains an equality condition with a constant (or program parameter), i.e. `R.A = '...'`, then an index on `R(A)` returns ROWIDs in sorted order. Otherwise (ROWIDs are combined from several index entries), ROWIDs are not sorted.

- If ROWIDs are in random order, less than about 4% of the blocks must be selected to make the index effective.

- Indexes can be useful for range queries and joins if a very small percentage of the rows is accessed.

Summary: Index Selection (4)

- Indexes produce ROWIDs in the sort order of the indexed attribute. This can be a plus for `ORDER BY`, `GROUP BY`, `DISTINCT`.

  But it usually is not effective to use an index only for sorting, except in case of an index-only execution plan (i.e. all required attributes are already contained in the index).

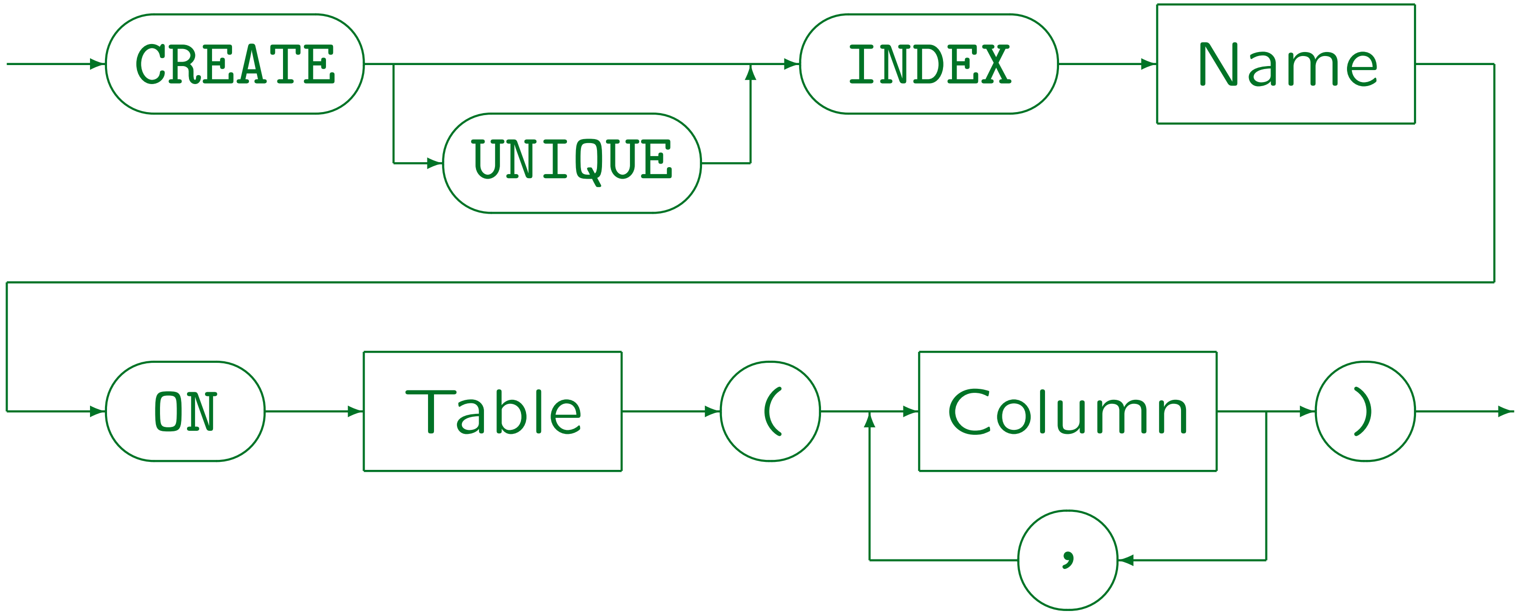
- For indexes on attribute combinations, put the attribute first that is also used without the other.

  An index on (A, B) can be used like an index on A, but not like an index on B. The performance is slightly worse than the index only on A (and one loses the order of the ROWIDs), but one would usually not declare an index on (A, B) and one on A.

Overview

Indexes in SQL (1)

- SQL command for creating an index:



- E.g.: `CREATE INDEX CUSTIND1 ON CUSTOMERS(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 (and most other DBMS) automatically create an index for `PRIMARY KEY`/`UNIQUE` constraints.

  If constraint names are defined for the keys, Oracle uses the constraint name as names for the indexes.

  In Oracle, there can be an index and a table with the same name.

Indexes in SQL (3)

- In Oracle, one can specify the same storage parameters as for a table (except PCT_USED):

```
CREATE UNIQUE INDEX IND3 ON
       CUSTOMERS(LAST_NAME, FIRST_NAME, CITY)
TABLESPACE USER_DATA
STORAGE(INITIAL 40K
        NEXT 10K
        PCTINCREASE 0) PCTFREE 20;
```

Probably, PCTFREE is the space left in the leaf blocks when the index is first created (for future insertions, to keep the sorted order if possible).

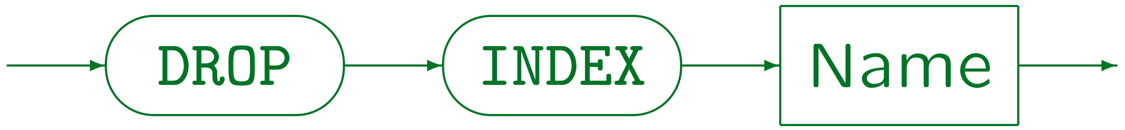
Indexes in SQL (4)

- Storage parameters can also be specified for indexes that are automatically created to enforce keys:

```
CREATE TABLE CUSTOMERS(
             CUSTNO NUMERIC(8),
             FIRST_NAME VARCHAR(30),
             ...,
             CONSTRAINT CUST_KEY
             PRIMARY KEY(CUSTNO)
             USING INDEX
             PCTFREE 20  TABLESPACE IND2
             STORAGE(INITIAL 200K NEXT 100K))
PCTFREE 10  TABLESPACE TAB1  STORAGE(...)
```

Indexes in SQL (5)

- Command for deleting indexes:



- In principle, one can experiment with indexes:
  - ◇ Create an index,
  - ◇ check whether the queries run faster,
  - ◇ delete it if not.
- But see next slide.

Indexes in SQL (6)

- For large tables, creating an index is an expensive operation: The DBMS must first sort the entire table.

  This also needs temporary storage space.

- Thus, such experiments can be done only at the weekend, not during the main business hours.
- But such experiments do not replace careful planning: There are too many possible combinations of indexes, one cannot try them all.

Bulk Loads (1)

- If a large table is loaded with data (e.g. 1 million rows are inserted in one operation), it is recommended to create indexes only afterwards.

  This includes the indexes for keys. Keys can be added with an `ALTER TABLE` command.

- The total time spent for updating the indexes for each tuple might be larger than creating the entire index in one operation.

Bulk Loads (2)

- Even more important is that when the index is created, the leaf blocks are filled completely (with the `PCTFREE` space reserve) and stored on the disk in the sort sequence.
- Later insertions might require to split blocks. Then both will be only half full and one of them has to be moved to another disk location.

  In general, indexes that had a lot of updates are not as good for range queries or full index scans as freshly created indexes. Then recreating the index may speed up such queries.

Data Dictionary: Indexes (1)

- The view IND (synonym for USER_INDEXES) contains one row for each index created by the current user.

  There are also ALL_INDEXES/DBA_INDEXES as usual.

- The most important columns are:
  - ◇ TABLE_NAME, TABLE_OWNER: Indexed table.
  - ◇ INDEX_NAME: Name of the index.
  - ◇ UNIQUENESS: UNIQUE or NONUNIQUE.
  - ◇ TABLESPACE_NAME, INITIAL_EXTENT, NEXT_EXTENT, PCT_INCREASE, PCT_FREE, . . . : Storage parameters.

Data Dictionary: Indexes (2)

- The following columns in `IND`/`USER_INDEXES` contain values only after the corresponding table was analyzed (with `ANALYZE TABLE`):
  - ◇ `BLEVEL`: Height of the B-tree minus 1.

    E.g. `BLEVEL=0` means the root is the only leaf block.
  - ◇ `LEAF_BLOCKS`: Number of leaf blocks.
  - ◇ `DISTINCT_KEYS`: Number of different values in the indexed column.

    If the column is a key, this would be the same as the number of rows in the table.

Data Dictionary: Indexes (3)

- Columns in IND/USER_INDEXES, continued:
  - ◇ AVG_LEAF_BLOCKS_PER_KEY: Number of different leaf blocks in which the same key appears.

    This is 1 for an index over a key attribute. It only grows bigger than 1 if many rows have the same attribute value. As before, this column is only filled after an ANALYZE TABLE.
  - ◇ AVG_DATA_BLOCKS_PER_KEY: Average number of data blocks that contain rows with the same attribute value.

    I.e. we must expect that much block accesses to the table for a given column value. Again, ANALYZE TABLE must be used.

Data Dictionary: Indexes (4)

- Columns in `IND/USER_INDEXES`, continued:
  - ◇ `CLUSTERING_FACTOR`: The number of block access to the table that would be needed if the index were used to produce a sorted sequence of rows.

    If this value is near to the number of blocks in the table, the rows in the table are already well ordered. If this value is near to the number of rows in the table, the rows are randomly stored. Again, `ANALYZE TABLE` is needed to fill this column.
- These statistics are important for the optimizer (to estimate the runtime of a query evaluation plan).

Data Dictionary: Indexes (5)

- The Oracle data dictionary does not contain the number of branch blocks.

  But the total number of blocks allocated for the index (including free blocks) can be determined from `USER_SEGMENTS`. Also, the number of branch blocks is usually much smaller than the number of leaf blocks.

- Exercise: Write a query that prints the names of all indexes on the table `CUSTOMERS`, together with the number of block accesses that are needed to evaluate a query of the form

  `SELECT * FROM CUSTOMERS WHERE` $A = c$

  with the index (count branch, leaf, and table blocks).

Data Dictionary: Indexes (6)

- USER_IND_COLUMNS shows which columns are indexed. The most important columns are:
  - ◇ INDEX_NAME: Name of the index.
  - ◇ TABLE_NAME: Name of the indexed table.
  - ◇ COLUMN_NAME: Name of the indexed column.
  - ◇ COLUMN_POSITION: Sequence number (1, 2, ...) of this column if index over a column combination.
  - ◇ COLUMN_LENGTH: "Indexed column length".

    For VARCHAR data types, it is the same as the declared maximal length, for NUMERIC types, it is 22 (maximal number of bytes).

Storage Size for Indexes (1)

- These formulas are from the Oracle8 Administrator's Guide. The result is only an estimate.

  I haven't found something similar in the 8i documentation.

- Each index entry (in the leaf blocks) consists of a header, a ROWID (6 bytes?), and the column data.

Header (2 Bytes)	ROWID (6 Bytes)	Column1 Length	Column1 Data	. . .

- In non-unique indexes the ROWID needs one length byte. (It is treated like another column.)

Storage Size for Indexes (2)

- Available space per block:
  - ◇ The block header needs 161 bytes.

    $113 + (\texttt{INITRANS} * 24)$, where `INITRANS` is by default 2 for indexes.
  - ◇ `PCTFREE` applies as for table data blocks.
- Average length of a leaf block entry:
  - ◇ The header needs 2 bytes.
  - ◇ Column data is stored as in table rows.

    But the required length bytes are slightly different: 1 Byte is needed to store the column length ≤ 127, 2 Bytes otherwise.

Storage Size for Indexes (3)

- Number of entries per leaf block:

$$\texttt{TRUNC}\left(\frac{\text{Available Space per Block}}{\text{Average Entry Length}}\right)$$

  `TRUNC` means to round downwards (same as `FLOOR`).

- Number of leaf blocks:

$$\texttt{CEIL}\left(\frac{\text{Number of rows with non-null column value}}{\text{Number of entries per leaf block}}\right)$$

- Oracle suggests to add 5% for the branch blocks.

  This depends on the branching factor: For instance, in a complete binary tree of height h, there are 2^{h-1} leaf blocks and $2^{h-1}-1$ branch blocks. But such extreme cases are very seldom in practice.

Storage Size for Indexes (4)

Experiment:

- `CREATE TABLE R(A VARCHAR(10)`
  ```
                   CONSTRAINT I PRIMARY KEY);
  ```
- Fill the table with n rows containing attribute values `'ABCDE10001'`, `'ABCDE10002'`, etc.
- `ANALYZE TABLE R COMPUTE STATISTICS;`
- `SELECT BLEVEL FROM IND WHERE INDEX_NAME = 'I';`

  `PCTFREE` (column `PCT_FREE` in `IND`) is 10.
  The `DB_BLOCK_SIZE` is 2048.

Storage Size for Indexes (5)

- For $n = 88$, `BLEVEL` is still 0, for $n = 89$ a single block is no longer sufficient for the index and `BLEVEL` becomes 1.
- The above formula would give:
  - ◇ Available space per block:

    $$2048 - 161 - 205 = 1682.$$
  - ◇ Each entry needs $2 + 6 + 1 + 10 = 19$ Bytes.
  - ◇ Thus, $1682/19 = 88.53$ index entries can be stored per leaf block.

Storage Size for Indexes (6)

- If one does not declare `A` as a key, but creates a non-unique index on `R(A)` and inserts two rows for each value of `A`, the leaf block can contain at most 42 index entries.
- An index entry (with two ROWIDs) would now need

  $$2 + 1 + 2 * 6 + 1 + 1 + 10 = 36 \text{ Byte.}$$

  Thus, $1682/36 = 46$ should be possible.

  Also other experiments show that the formula is not always accurate.

Overview

Physical Database Design (1)

- The purpose of physical database design is to ensure that the database system meets the performance requirements.
- Physical database design depends heavily on
  - ◇ The concrete DBMS chosen for the implementation.
  - ◇ The table sizes and how often each application is executed (load profile).

Physical Database Design (2)

- Don't think too early during database design about performance:
  - ◇ Conceptual design is difficult enough — separate the problems.
  - ◇ If one must later switch to another DBMS or the load profile changes, the conceptual design remains still valid.
- Accept compromises to a clear design for performance reasons only if experiments show that otherwise the performance requirements cannot be met.

Inputs to Physical Design (1)

- How big the tables will be:
  - ◇ Number of rows,
  - ◇ size of column data (min/max/avg),
  - ◇ frequency of null values.
- How will the tables grow over time?

  How many rows will each table have in one year?
- Which strategies should be used for purging the data?

  When can rows be deleted? Or is the database ever-growing?

Inputs to Physical Design (2)

- How are column values distributed?
  - ◇ Will there be many different values or will the same value be often repeated?
  - ◇ Are there especially common values?

    I.e. is the distribution uneven?
- Will rows grow over time via updates?

  Will some columns be null at insertion and filled out later?

Inputs to Physical Design (3)

- Which application programs exist and which queries and updates do they execute?
- How often is each application program executed?
- Especially the following information is needed:
  - ◇ Which columns are used in equality conditions, range conditions, joins, group by.
  - ◇ Which columns of tables are accessed together?
  - ◇ Which columns are updated? How often do these updates happen? How frequent are insertions into each table? What about deletions?

Inputs to Physical Design (4)

- Performance requirements:
  - ◇ What response time is needed for which step in an application program?

    Commonly executed interactive programs need fast answers, seldomly executed programs could run longer.
  - ◇ Can reports be generated over night?

    Or do they have to be generated during main business hours?
  - ◇ Are there unpredictable ad-hoc SQL queries?

    Can we keep them out of our main DB during business hours? Would it be acceptable if ad-hoc queries use a slightly outdated or aggregated copy of the data?

Inputs to Physical Design (5)

- Can we have scheduled maintenance times or do we need to be up 7 days a week, 24 hours a day?
- How fast must the system be up again after a power failure (system crash) or after a disk failure?
- How big is the hardware?

  How much main memory does the machine have? How many disks? What is the capacity of these disks? How many disk controllers? How many disks can be connected? What is the maximum transfer rate? Is there money for buying more main memory, disks, etc.?
- Do we have to use a particular DBMS?

  What is the budget for buying a DBMS? And for updates/support?

Experimental Approach

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

  Creating a new index is simple in relational systems. However, if one has to buy entirely new hardware because performance criteria are not met, one has a problem. Thus, it is important to think about realistic system loads during the design.

- There are (expensive) tools for simulating given loads.
- Don't start using the system before you are sure that it will work.

Physical Design Decisions

- How should the tables be stored?

  Heap file, index cluster, hash cluster, index-organized table? Which tables should be clustered together? See Chapter 5.

- How big should the initial extents be?
- What space reserve is needed for updates (`PCTFREE`)?
- Which indexes would be useful?
- Should tables be partioned (horizontally/vertically)?
- Should some tables be specially cached?
- Should tables be denormalized (introducing redundant information for performance reasons)?

Outlook (1)

Further Oracle Data Structures:

- Clusters for storing table data

  This permits to store rows for the same attribute value together. It is even possible to store rows from different tables in one cluster (makes joins very fast).

- Hash clusters

  Here, the block (storage position) is computed from column value. This is the fastest possible access for conditions of the form $A = c$, but it is less flexible than a B-tree.

- Bitmap indexes

  Good for columns with few different values, for each row and each possible value there is one bit.

Outlook (2)

Further Oracle Data Structures, continued:

- Index-organized tables

  Instead of ROWIDs, the index contains the complete rows.

- Function-based indexes

  Instead of indexing an attribute value, the search key of the index can be a function of the row.

- Object-relational features

  E.g. non-first normal form tables: Table entries can be arrays.

Outlook (3)

- The literature contains many more data structures for indexes:
  - ◇ E.g. there are special indexes for geometric data, where one can search all points in a given rectangle, the nearest point to a given point, etc.
- In general, an index allows special ways to compute certain parameterized queries. E.g. a Hash-index on `R(A)` supports

```
SELECT ROWID FROM R WHERE A=:1
```