Part 10: B-Tree Indexes

References:

- Elmasri/Navathe: Fundamentals of Database Systems, 3nd Ed.,
  Chap. 6: "Index Structures for Files", Section 16.3: "Physical Database Design in Relational Databases"
  Section 16.4: "An Overview of Database Tuning in Relational Systems"
- Silberschatz/Korth/Sudarshan: Database System Concepts, 3rd Ed., Chap. 11: "Indexing and Hashing"
- Ramakrishnan/Gehrke: Database Management Systems, 2nd Ed., Mc-Graw Hill, 2000, Chap. 8: "File Organizations and Indexes", Chap. 9: "Tree-Structured Indexing", Chap. 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.
  Ch. 14: "Managing Indexes".
- Oracle 8i Designing and Tuning for Performance, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76992-01.
  Ch. 12: "Data Access Methods".
- Oracle 8i SQL Reference, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76989-01.
  CREATE INDEX, page 7-291 ff.
- Oracle8 Administrator's Guide, Release 8.0, Oracle Corporation, 1997, Part No. A58397-01.
  Appendix A: "Space Estimations for Schema Objects".
- Gray/Reuter: Transaction Processing, Morgan Kaufmann, 1993, Chapter 15.
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.

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 for a given query whether it can be evaluated faster using an index.
- explain disadvantages of indexes.
- enumerate input data about the application that is necessary for physical database design.
- select good indexes for a given application.
- write `CREATE INDEX` statements in Oracle SQL.

Overview

1. Motivation
2. B-Trees
3. Query Evaluation with Indexes, Index Selection
4. Index Declaration, Data Dictionary
5. Physical Design Summary

Motivation (1)

- Let a table with information about customers be given:

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

- Consider the query for the data of a specific customer:

```
SELECT *
FROM   CUSTOMERS C
WHERE  C.CUSTNO = 7635
```

- Query evaluation with a full table scan (reads all rows):

```
for each row C in CUSTOMERS do
    if C.CUSTNO = 7635 then print C; fi
od
```

Motivation (2)

- 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.
- So 17 rows fit into a 2K block (with `PCTFREE` = 10).
  2048 Byte (block size) − 90 Byte (block header) − 205 Byte (space reserve) = 1753 Byte. $1753/100 = 17.5$.
- Suppose the database contains 2 Million customers.
  Then at least 117 648 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 (3)

- Even if the entire table is stored in one extent of contiguous DB 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 considered the maximum which does not hinder users during interactive work.
  Some authors require a "subsecond response time".
- Full table scans of big tables do not profit from caching.
- Suppose that 100 employees enter orders in parallel, and for each order the customer data must be accessed.
- Since the DBMS can process only five queries per minute, each employee can only enter one order every 20 minutes.

Motivation (4)

- Without special access structures, there is no way to avoid the full table scan.
- Database systems offer special data structures, called indexes, which allow to find all rows with a given attribute value without reading and checking every row in the table.
- 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 index in a DBMS is very similar: A (sorted) list of all occurring values for a specific column together with references to the rows which contain the respective value.

Motivation (5)

- Indexes are sometimes called "inverted files":
  - The heap file that contains the table data supports the mapping from ROWIDs to attribute values.
  - An index supports the mapping from values for a specific attribute (or attribute combination) to ROWIDs.
- In order to solve the example query, the DBMS will first search the index over `CUSTOMER(CUSTNO)` (e.g. 4 block accesses).
- In the index entry for the given customer number 7635, it will find the ROWID of the requested `CUSTOMER`-row.
- Finally, it reads the row with this ROWID from the `CUSTOMER` table (1 block access, 2 if row migrated).

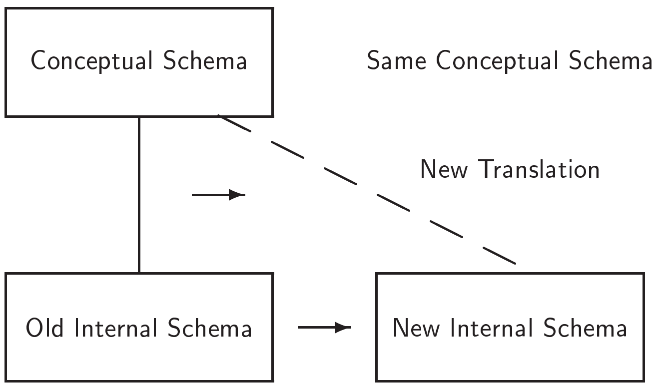
Physical Data Independence (1)

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

  ```
  CREATE INDEX IND_CUST1 ON CUSTOMER(LAST_NAME)
  ```
- The query optimizer will now automatically use the index, whenever it thinks that this would be advantageous, e.g. for:

  ```
  SELECT *
  FROM   CUSTOMER
  WHERE  LAST_NAME = 'Jones'
  AND    CITY = 'New York'
  ```
- The query does not have to be changed to use the index.
- The existence of indexes is transparent to the SQL user.

Physical Data Independence (2)



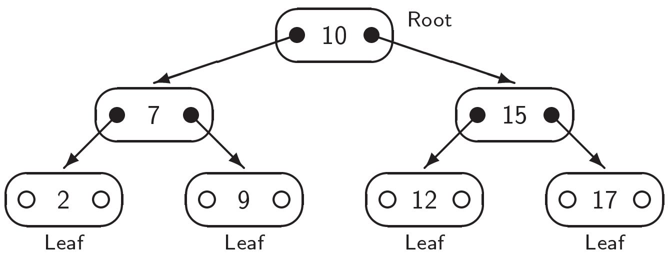
Overview

1. Motivation
2. B-Trees
3. Query Evaluation with Indexes, Index Selection
4. Index Declaration, Data Dictionary
5. Physical Design Summary

B^+-Trees: Overview

- The standard 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 standard binary search trees.
- However, since anyway whole blocks must be read from external memory, the branching factor is much higher than 2.

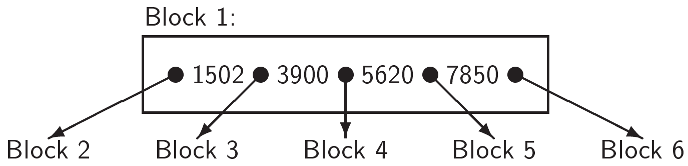
Repetition: Binary Search Trees



- 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: Structure (1)

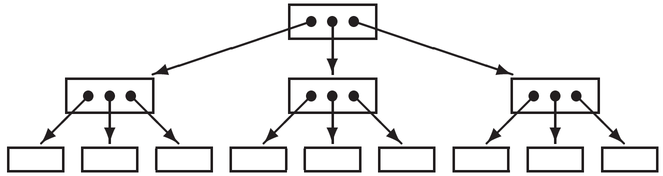
- Consider an index on `CUSTOMER(CUSTNO)`.
- A B$^+$-tree consists of branch blocks and leaf blocks. The branch blocks guide the search for a row:



- If the requested `CUSTNO` is ≤ 1502, continue in Block 2.
  For `CUSTNO` > 1502 and `CUSTNO` ≤ 3900, go to Block 3.
  ...
  If `CUSTNO` > 7850, continue in Block 6.

B$^+$-Trees: Structure (2)

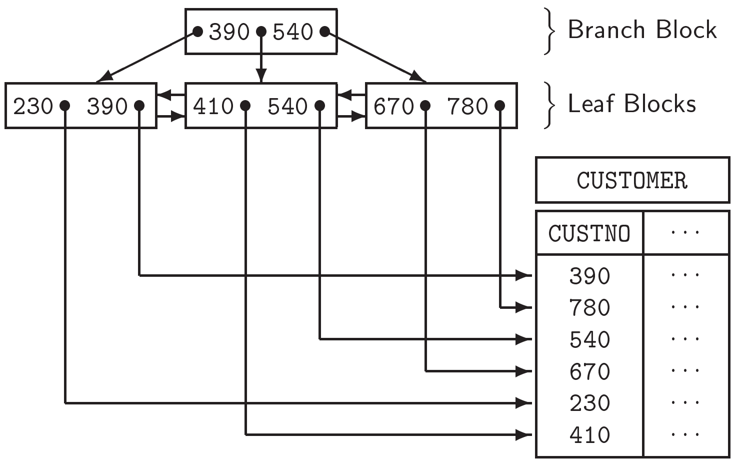
- The referenced blocks have the same structure, so that a tree is built:



- The blocks in the lowest level ("leaf blocks") contain all 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".

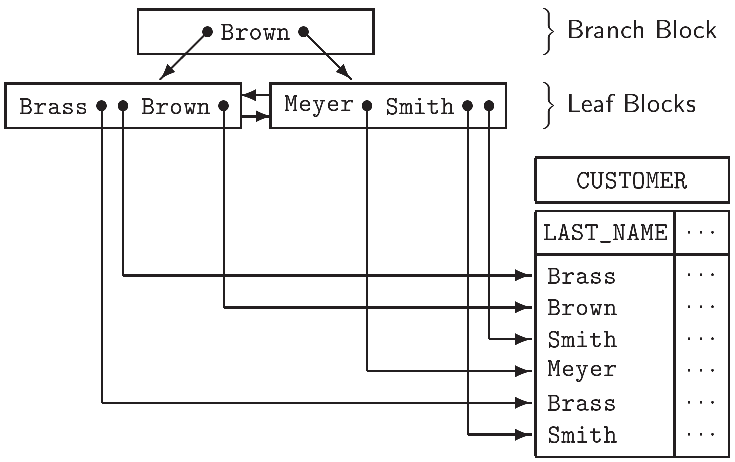
B$^+$-Trees: Structure (3)



B$^+$-Trees: Structure (4)

- 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.
  An index in a book may reference several pages for a keyword.
- An example is shown on the next slide.
  In that example, the "search keys" of the index (which are not keys of the table) are also strings, not numbers as before. Any data type can be used, one only needs to define an order on it (which is used for ordering the data in the leaf blocks and for navigating in the branch blocks).

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 guarantees that one never needs 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 entry.) If blocks had to be completely full, an insertion of one tuple could require a change in every block of the tree. With the relaxed requirement, Insertion/Deletion are possible in $O\big(\log(n)\big)$.

B$^+$-Trees: Structure (7)

Insertion Algorithm:

- 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. We only need to remember the path through the tree, because we 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: Structure (8)

Insertion Algorithm, Continued:

- 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 the branch block has space, we are done.
- Otherwise the branch block is split into two, and the insertion goes up to the next higher 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: Performance (1)

- 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 10-16.
  Height 3: Two levels of branch nodes, see Slide 10-15.

B^+-Trees: Performance (2)

- So for the `CUSTOMER` table with 2 Million entries, the B-tree will have height 4.
  If the blocksize were 4K (i.e. max 200 entries/block), the height would be 3 or 4 (4 is very unlikely: min. 2 Mio. rows).
- With a tree of height 4, it takes 5 block accesses to get the row for a given customer number.
  Four for the index and one for fetching the row from the table given the ROWID from the index. If the row was migrated, 6 block accesses would be needed in total.
- So the query can be executed in 50ms.
  A fast modern disk needs e.g. 10ms per random block access.
- One disk can support about 14 queries per second (70% load).

B^+-Trees: Performance (3)

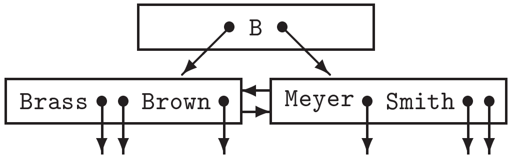
- Table access via an index profit from caching of disk blocks: E.g. it is very likely that the root node of the index and some part of the next level will be in the buffer.
- Since the height of the B-tree grows only logarithmically in the number of rows, B-trees never become very high.
  Heights greater than 4 or 5 are rare in practice.
- 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 the indexed column 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.

B^+-Trees: Performance (4)

- 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 stored in the leaf blocks, so no information is lost.

- E.g. in the above example, it suffices to store `'B'` in the root node. Oracle uses such a "rear compression".



Overview

1. Motivation
2. B-Trees
3. Query Evaluation with Indexes, Index Selection
4. Index Declaration, Data Dictionary
5. Physical Design Summary

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   CUSTOMER
WHERE  LAST_NAME = 'Smith'
AND    CITY = 'New York'
```

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

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

Applications of Indexes (2)

Multiple Indexes on One Table:

- There can be any number of indexes on the same table.
- E.g. 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`.
- An alternative is to access the matching ROWIDs from both indexes, intersect them, and access the table only for rows which satisfy both conditions.

Applications of Indexes (3)

Indexes on Attribute Combinations:

- It is possible to create indexes on attribute combinations:

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

- Then the entries in the B-tree are more or less the concatenation of LAST_NAME and CITY.
- A query like the above one (containing 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.

  A pure index on LAST_NAME would be slightly better, since the tree height is smaller (the tree contains no cities).

Applications of Indexes (4)

Using Indexes for Range Queries:

- A B-tree index can also be used for range queries (since it contains the entries in sorted order).
- An index on CUSTOMER(CUSTNO) can be used for this query:

```
SELECT *
FROM   CUSTOMER
WHERE  CUSTNO BETWEEN 7000 AND 7999
```

- An index on INVOICE(AMOUNT) can be used for this query:

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


Applications of Indexes (5)

String Pattern Matching with LIKE:

- The following query can profit from an index on LAST_NAME:

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

- This is actually equivalent to the range query:

```
SELECT *
FROM   CUSTOMER
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 (6)

Applications for Joins:

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

```
SELECT C.LAST_NAME
FROM   INVOICE I, CUSTOMER C
WHERE  I.AMOUNT >= 10000
AND    C.CUSTNO = I.CUSTNO
```

- One way to evaluate this query is to use an index on INVOICE(AMOUNT) to locate large invoices and then to use an index on CUSTOMER(CUSTNO) to find the customer data.

  See the chapter about query evaluation for more possibilities.

Applications of Indexes (7)

Applications for ORDER BY and GROUP BY:

- Consider this query:

```
SELECT   *
FROM     CUSTOMER
WHERE    CUSTNO BETWEEN 7000 AND 7999
ORDER BY CUSTNO
```

- A B-tree index produces the rows in the sort order of the indexed attribute, so if the index on CUSTNO is used to evaluate the condition, the ORDER BY is for free.
- In the same way, GROUP BY/DISTINCT profit from an index.
- It is possible to use an index only for the purpose of sorting.

Applications of Indexes (8)

Index-only Query Execution Plans:

- Some queries can be answered only from the index without fetching the rows from the indexed table. E.g. here it suffices the count the ROWIDs from the index on CUSTOMER(CUSTNO):

```
SELECT COUNT(*)
FROM   CUSTOMER
WHERE  CUSTNO BETWEEN 7000 AND 7999
```

- Combined indexes are useful for such queries. E.g. this query can be answered from an index on LAST_NAME, CITY:

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


Applications of Indexes (9)

Indexes and Keys / Foreign Keys:

- When a key (PRIMARY KEY or UNIQUE) is declared, Oracle automatically creates an index on the key attributes.
- When new rows are inserted, this index helps to check the key constraint (a UNIQUE index enforces the constraint).
- Consider the foreign key: INVOICE.CUSTNO $\rightarrow$ CUSTOMER.
  - If an INVOICE is inserted, the system will use the index on CUSTOMER(CUSTNO).

    The table CUSTOMER does not have to be accessed.
  - If a CUSTOMER is deleted, an index on INVOICE(CUSTNO) would be helpful (or else a full table scan is needed).

Applications of Indexes (10)

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 often this is useful: An index of a column which contains 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: Advantages of Indexes

- The selection $\sigma_{\mathsf{A}=\mathsf{c}}(\mathsf{R})$ is much faster if there is an Index on the attribute A of R.
- Also a join $\mathsf{R} \underset{\mathsf{A}=\mathsf{B}}{\bowtie} \mathsf{S}$ can profit from an index on R.A or S.B.
- A B-tree index also helps with range-queries of the form $\sigma_{\mathsf{A}<\mathsf{c}_1 \,\wedge\, \mathsf{A}>\mathsf{c}_2}(\mathsf{R})$ (includes special cases of `LIKE`).
- 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`.

Comparison with a Full Table Scan (1)

- Using an index is only advantageous if the index selects only a small fraction of all rows.
- "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.
- A sequential scan of all contiguously stored blocks is much faster than random accesses, e.g. 2 500 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.

Comparison with a Full Table Scan (2)

- So a selective access becomes interesting when less than $100/2500 = 4\%$ of the blocks is accessed.
- 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.

Comparison with a Full Table Scan (3)

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

  Except for joins, the ROWIDs are actually returned in the order of the index attribute (only groups for the same attribute value are ordered by ROWID).
- 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.

Comparison with a Full Table Scan (4)

- Normally, each block contains many rows.
- E.g. if every block contains 100 rows, and 3% of the rows are requested (that are randomly distributed over the blocks), it is very likely that $\geq 80\%$ of the blocks must be accessed.
- If, however, only one row fits in each block, accessing 3% is even advantageous with completely random accesses.
- So the selectivity of the search condition must be seen relative to the number of rows per block.
- If the ROWIDs are not sorted by block number, and there are not enough buffer frames, it might even be that the same block must be read multiple times (see next slide).

Comparison with a Full Table Scan (5)

- E.g. consider a table with 100 000 rows and 1000 blocks, i.e. 100 rows fit in each block.
- The full table scan reads 8 MB in 0.4 seconds.
- Suppose that 10% of the rows are selected.
- If the ROWIDs are not ordered, 10 000 random block accesses will be needed.
- 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.
- So if the query is evaluated with the index, it runs 80 seconds (not counting the index accesses).

Comparison with a Full Table Scan (6)

- 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.
- 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.

Comparison with a Full Table Scan (7)

- Note that "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 (except for very small tables): There will be only a single result row.

  However, doing this iteratively in a join might be worse than other join methods (see chapter about query evaluation).

Comparison with a Full Table Scan (8)

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 `CUSTOMER(CUSTNO)`. With 2 Million customers, if one block can contain 100 entries, and blocks are on average 75% full, already 26 667 leaf blocks are need.

  Plus about 361 branch blocks.

- So the size of this index is 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 space needed for the indexes might be larger 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.
- Updates only become slower if an indexed column is updated. So 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.

- It is often suggested to have not more than about 4–6 indexes on one relation.

  This is not a hard rule. Violate it if you have reasons.

- Check whether the query optimizer really uses your indexes!

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.
- 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: Row size and clustering effects (storage order) must be considered.
- 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 ROWIDs are in random order, less than about 4% of the blocks must be selected to make the index effective.
- If the query contains an equality condition with a constant (or program parameter), i.e. `R.A = '...'`, then the index returns ROWIDs in sorted order.
- Otherwise (ROWIDs are combined from several index entries), ROWIDs are not sorted.
- Indexes can be useful for range queries and joins if only a small percentage of the rows is accessed.

Summary: Index Selection (4)

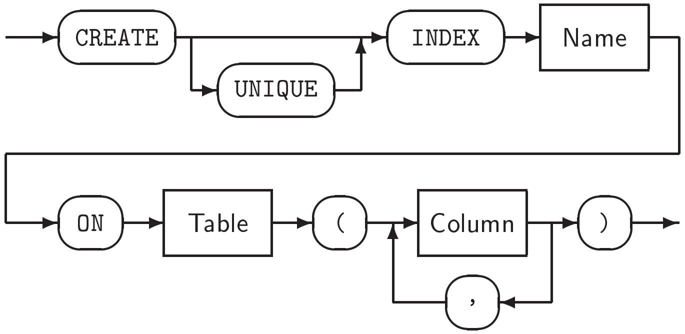
- Indexes produce ROWIDs in the sort order of the indexed attribute. This can be a plus for `ORDER BY`, `GROUP BY`, `DISTINCT`.
  But it usually is not effective to use an index only for sorting, except in case of an index-only execution plan (i.e. all required attributes are already contained in the index).
- If an index on a combination of attributes is created, put the attribute first which 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 (and one loses the order of the ROWIDs), but usually it makes no sense to declare an index on (A, B) and one on A.

Overview

1. Motivation
2. B-Trees
3. Query Evaluation with Indexes, Index Selection
4. Index Declaration, Data Dictionary
5. Physical Design Summary

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:

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
       CUSTOMER(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 the indexes that are automatically created to enforce keys:

```
CREATE TABLE CUSTOMER(
             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)

- In principle, you can experiment with indexes: Create an index, see whether your queries run faster, delete it if not.
- However, keep in mind that 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.
- Also, such experiments do not replace careful planning: There are too many possible combinations of indexes, you cannot try them all.

Bulk Loads

- If a large table is loaded with data (e.g. 1 million rows are inserted in one operation), it is recommended to create indexes (including the ones for keys) only afterwards.
- The total time spent for updating the indexes for each tuple might be larger than creating the entire index in one operation.
- 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.

  Also later, recreating an index may speed up the system.

Indexes in the Data Dictionary (1)

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

  There are also ALL_INDEXES/DBA_INDEXES as usual.
- I.e. which indexes exist on CUSTOMER?

```
SELECT INDEX_NAME
FROM   ALL_INDEXES
WHERE  TABLE_NAME = 'CUSTOMER'
AND    TABLE_OWNER = USER
```

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

  There can be an index and a table with the same name.

Indexes in the Data Dictionary (2)

- You find also the TABLESPACE_NAME and the physical storage parameters in IND.

  Note that NEXT_EXTENT is the size of the now next extent, not necessarily the value of NEXT in the index declaration (if you have a non-zero PCTINCREASE).
- If you analyze the corresponding table, all indexes on it are also analyzed.
- You need to do this in order to fill out (or update) size data in IND (e.g. BLEVEL, LEAF_BLOCKS, DISTINCT_KEYS, ...).
- BLEVEL: Height of the B-tree minus 1.

  E.g. BLEVEL=0 means the root is the only leaf block.

Indexes in the Data Dictionary (3)

- LEAF_BLOCKS: Number of leaf blocks.
- DISTINCT_KEYS: Number of different values in the indexed column.
- 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.
- 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.

Indexes in the Data Dictionary (4)

- CLUSTERING_FACTOR: The number of block access to the table we would need if we use the index 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.
- All these statistics are important for the optimizer.
- The Oracle data dictionary does not contain the number of branch blocks.

  The total number of blocks allocated for the index (including free blocks) can be determined from USER_SEGMENTS.

Indexes in the Data Dictionary (5)

- To find out which columns are indexed, use USER_IND_COLUMNS. It has the following columns:
  - INDEX_NAME: Name of the index.
  - TABLE_NAME: Name of the indexed table.
  - COLUMN_NAME: Name of the indexed column.
  - COLUMN_POSITION: Sequence number of this column in case of an index over a column combination (1, 2, ...).
  - COLUMN_LENGTH: "Indexed column length".

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

Storage Size for Indexes (1)

- These formulas are from the Oracle8 Administrator's Guide, I haven't found something similar in the 8i documentation.
- The result is only an estimate.
- 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)

- First compute the available space per block:
  - The block header needs 161 bytes.
    $113 + (\texttt{INITRANS} * 24)$, INITRANS is by default 2.
  - PCTFREE applies as for table data blocks.
- Then compute the 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.
- From this, the number of leaf blocks can be calculated.
- Oracle suggests to add 5% for the branch blocks.

Storage Size for Indexes (3)

Experiment:

- ```
  CREATE TABLE R(A VARCHAR(10)
                 CONSTRAINT R_KEY 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 = 'R_KEY';`
  PCTFREE (column PCT_FREE in IND) is 10.
  The DB_BLOCK_SIZE is 2048.
- For $n = 88$, BLEVEL is still 0, for $n = 89$ a single block is no longer sufficient for the index and BLEVEL becomes 1.

Storage Size for Indexes (4)

- 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.
- 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 $2 + 1 + 2 * 6 + 1 + 10 = 36$ Byte. Thus, $1682/36 = 46$ should be possible. Also other experiments show that the formula is not always accurate.

Overview

1. Motivation
2. B-Trees
3. Query Evaluation with Indexes, Index Selection
4. Index Declaration, Data Dictionary
5. Physical Design Summary
  ```

Physical Database Design

- 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).
- Don't think too early during DB design about performance:
  - Conceptual design is difficult enough — separate the problems.
  - If you later switch to another DBMS or your load profile changes, your conceptual design remains valid.

Inputs to Physical Database 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 DB ever-growing?
- How are column values distributed? Will there be many different values or will the same value be often repeated?
  Are there especially common values?
- Will rows grow over time via updates?
  Will some columns be null at insertion and filled out later?

Inputs to Physical Database Design (2)

- Which application programs exist and which queries and updates do they execute?
- How often is each application program executed?
- Especially you need the following information:
  - 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 Database Design (3)

- 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 Database Design (4)

- 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 much main memory does our machine have?
  How many disks? What is the capacity of these disks?

  How many disk controllers does the machine have? How many disks can be connected? What is the maximum transfer rate? Can we buy more main memory, disks, etc.?
- Do we have to use a particular DBMS? What is our budget for buying a DBMS? And for updates/support?

Iterative/Experimental Approach

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

Decisions during Physical Design

- How should the tables be stored (heap file, index cluster, hash cluster)? Which tables should be clustered together?
- How big should the initial extents be (ideally they should contain the entire table)?
- What space reserve is needed for updates (`PCTFREE`)?
- Which indexes would be useful?
- Which tables should be partioned (horizontally/vertically)?
- Should some tables be specially cached?
- Should tables be denormalized (introducing redundant information for performance reasons)?

Outlook (1)

Oracle has also:

- Clusters for storing table data (in order 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 (block is computed from column value).
- Bitmap indexes (for columns with few different values, for each row and each possible value there is one bit).
- Index-organized tables (instead of ROWIDs, the index contains the complete rows).
- Object-relational features, e.g. non-first normal form tables: Table entries can be arrays.

Outlook (2)

- The literature contains many more index data structures:
  - E.g. there are indexes which can support an access via attribute A, via B, and via the combination (A, B).

    With B-trees you would need two indexes, e.g. one over (A, B) and one over B.
  - E.g. there are special indexes for geometric data, where you look for all points in a given rectangle, or 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
```