Part 11: More Data Structures for Relations

References:

- Elmasri/Navathe: Fundamentals of Database Systems, 3rd Ed., Chap. 5: "Record Storage and Primary File Organizations", Chap. 6: "Index Structures for Files", 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., Chap. 8: "File Organizations and Indexes", Chap. 9: "Tree-Structured Indexing", Chap. 10: "Hashed-Based Indexing", Chap. 16: "Physical Database Design and Tuning".
- Kemper/Eickler: Datenbanksysteme (in German), Chap. 7, Oldenbourg, 1997.
- Oracle 8i Concepts, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76965-01.
- Oracle 8i SQL Reference, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76989-01.
- Oracle 8i Designing and Tuning for Performance, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76992-01.
- Gray/Reuter: Transaction Processing, Morgan Kaufmann, 1993, Chapter 15.
- Jason S. Couchman: Oracle8i Certified Professional: DBA Certification Exam Guide with CDROM.
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.

Objectives

After completing this chapter, you should be able to:

- explain the advanced data structures introduced in this chapter (clusters, index-organized tables, hash methods, bitmap indexes, partitioned tables): How do they work?
- list advantages and disadvantages of each data structure.
- select the best data structure for a given application.

Overview

1. Clusters
2. Hash-Based Indexes
3. Partitioned Tables
4. Bitmap Indexes
5. Index-Organized Tables

Single-Table Clusters (1)

- Suppose you often need all invoices of a given customer:

```
SELECT SUM(Amount)
FROM   Invoice
WHERE  CustNo = 7635
```

- Even if you have an index over `Invoice(CustNo)`, if the customer has 10 invoices, it is very likely that they are stored in 10 different blocks.
- However, instead of storing `Invoice` as a heap file, you can request to have the rows clustered by `CustNo` (in Oracle).
- In this case, the 10 rows for the invoices by a given customer will be stored in the same block (if possible).

Single-Table Clusters (2)

Advantages of Clusters:

- In the above example, if you need to access two blocks from the index, you need to access 12 blocks in total without the cluster and 3 blocks in total with the cluster, so your query is performed 4 times faster.
- Note that you have this advantage only if there are multiple rows with the same value for the cluster column.

  E.g. clustering by a key brings no advantages.
- There should also be not too many rows with the same value for the cluster column — normally all these rows should fit into a single block.

Single-Table Clusters (3)

- The value for the cluster column is stored only once for each set of rows with the same value.

  This gives a slight reduction in the needed disk space.
- Since the rows are stored grouped by the cluster column, e.g. this query can be evaluated without sorting:

```
SELECT   CustNo, SUM(AMOUNT)
FROM     Invoice
GROUP BY CustNo
```

- In general, any application of an index where one column value appears in several rows becomes faster.

  Except index-only execution plans and execution plans where ROWIDs are sorted or intersected.

Single-Table Clusters (4)

How Clusters work:

- You must specify in the cluster definition how much space all rows with the same value in the cluster attribute will need.
- E.g. you estimate that 600 Bytes are needed for all invoices of one customer (a customer has on average 10 invoices, each needs 60 Bytes).
- So Oracle will put rows of 3 customers into each 2K block.

  You must calculate with the block header. See below.
- When the first invoice of a new customer is inserted, Oracle assigns it to a block which contains invoices of no more than three other customers (and still has empty space).

Single-Table Clusters (5)

- Oracle enters the `CustNo` together with the chosen block into an index. Every cluster needs such an index over the cluster column.
- Then all further invoices for this customer will go to the same block. Oracle locates the block for a given `CustNo` via the index.
- If the block becomes full, overflow blocks are chained to it.
- Oracle does not reserve 600 Bytes for each customer. If one customer has many invoices and needs 1500 Byte, but the other three customers in that block need only 100 Byte each, this is still no problem.

Single-Table Clusters (6)

Disadvantages of Clusters:

- If the cluster column is updated, the row will be migrated to the block containing the rows with the new value. So do not define clusters on columns which are normally updated.
- Clusters reduce the flexibility:
  - If you estimate the size of the groups too small, invoices for one customer will be distributed over many blocks.

    And these blocks will not be consecutively stored on the disk and Oracle always fetches all of these blocks.
  - If you estimate the size too large, you waste disk space and therefore also full table scans will take longer.

Single-Table Clusters (7)

How to Create a Cluster:

- First you create the cluster and specify:
  - The name and data type of the cluster column(s).
  - The disk space for each distinct value of this column.
  - Storage parameters like in the CREATE TABLE command.

```
CREATE CLUSTER Invoice_Clust(CustNo NUMBER(7))
       SIZE 512
       TABLESPACE USER_DATA
       STORAGE(INITIAL 200K NEXT 50K
               PCTINCREASE 100)
       PCTFREE 10 PCTUSED 80
```


Single-Table Clusters (8)

- Then you create a table in this cluster:

```
CREATE TABLE Invoice(INo NUMBER(10) PRIMARY KEY,
                     CustNo NUMBER(7) NOT NULL,
                     Amount NUMBER(7,2) NOT NULL,
                     Issued DATE NOT NULL)
             CLUSTER Invoice_Clust(CustNo)
```

- Then you must create an index on the cluster before rows can be inserted:

```
CREATE INDEX CustNo_Idx ON CLUSTER Invoice_Clust
```

- You can do insertions, queries etc. on invoice as usual.

  The existence of a cluster is transparent to the application.

Single-Table Clusters (9)

Experiment:

- I created a cluster as shown above and inserted 7 invoices.

```
SELECT DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID),
       INo, CustNo FROM Invoice
```

BLOCK_NUMBER	INo	CustNo
583	101	1
583	102	2
583	103	3
583	107	3
584	104	4
584	105	5
584	106	5

Single-Table Clusters (10)

- I inserted the invoices in the order of ascending `INo`, but they are printed in the order in which they are stored.
- I had expected that Oracle puts invoices for 4 customers in one block, but because of the block overhead Oracle could assign only 3 customers to each block.
- I repeated the experiment with "`SIZE 400`", and then rows of 4 customers were stored in a single block.
- Note that without the cluster definition, Oracle would put all the given rows into a single block (and there would still be lots of free space).

Multiple-Table Clusters (1)

- You can assign more than one table to the same cluster (of course, the table must also contain a customer number).
- E.g. we could store `Customer` and `Invoice` together:

```
CREATE TABLE Customer(
                CustNo NUMBER(7) PRIMARY KEY,
                First_Name VARCHAR(20) NOT NULL,
                ...)
             CLUSTER Invoice_Clust(CustNo)
```

- Then rows from `Customer` and `Invoice` with the same `CustNo` are stored in the same block.

  Remember that without clusters, each block (even each segment) contains only rows from one table.

Multiple-Table Clusters (2)

- This makes joins between `Customer` and `Invoice` especially fast. More or less, the join is already precomputed in the way the rows are stored on disk.
- However, full table scans become slower now, because rows of `Invoice` are interspersed between the rows of `Customer`. So many more blocks are needed in the cluster than would be needed to store only the rows of `Customer`.
- Cluster indexes are structured a bit different than table indexes (they contain only the block number, not all ROWIDs). So Oracle has to create another index on `Customer(CustNo)` to enforce the `PRIMARY KEY` constraint.

Summary

- Clusters can bring significant performance improvements for queries using a non-unique index or containing joins.
- Don't use clusters on columns which are updated.
- You must be able to estimate the disk space needed for all rows having the same value in the cluster column.
- All these rows should fit into one block.
- Unless there is very little variation in this size, clusters do not utilize the disk space as good as a heap file.
- Full table scans will most likely become slower.
- You can cluster a table only with respect to one attribute (or one attribute combination).

Overview

1. Clusters
2. Hash-Based Indexes
3. Partitioned Tables
4. Bitmap Indexes
5. Index-Organized Tables

Motivation (1)

- If you have a B-tree of height 4, you still need 5 block accesses to fetch a row via this index.
  The root of the index is probably in the buffer cache.
- Hash methods ideally need only one block access.
- The idea is to assign every row to a specific block based on some computation on the value of the indexed attribute.
- E.g. we want a very fast access to the `Customer` table for a given customer number `CustNo`.
- When customer numbers are sequential numbers, and we know that 22 rows fit into one block, we could store the `Customer` row for `CustNo` = N in block $N/22$ (rounded).

Motivation (2)

- Then, given a specific customer number, e.g. 7356, we know that the row for this customer is stored in block $7356/22 = 334$, so we can get it with a single block access.
- So the advantage of the hash method is to replace the index lookup, which tells you in which block a row is stored, by a computation (which does not need any block accesses).
- You can do such computations not only on numeric columns, but also on string-valued columns (e.g. add the ASCII codes of the letters) (this is not a good hash function).
- Also, the column used as input to the hash function does not have to be the key of the table.

Hash Tables (1)

- Hash tables were part of your data structures course.
- The general idea is that you use an array with h locations for storing the rows (h = hash table size).
  Note that now each array position can store only one row.
  We talk about blocks (having space for multiple rows) later.
- When you insert a new row, you somehow compute a number between 1 and h based on the indexed attribute.
- This is the array location in which the row will be stored.
- The function which takes a specific value for the indexed attribute and returns the position in the table is called the hash function.

Hash Tables (2)

- E.g. for customer numbers in the range `1001..9999` you can use a hash table of size 8999 and `CustNo` − 1000 as hash function.
- Let us now assume that only 500 numbers in this range are actually in use (so the table `Customer` has only 500 rows).
- Then an array of size 8999 would waste too much storage.
- Hash methods perform bad if the table fills up completely, so let us use a table of size 701.

  One usually chooses a prime number as hash table size.
- As hash function we use `(CustNo MOD 701) + 1`.

  `CustNo MOD 701` is the remainder of `CustNo / 701`.

Hash Tables (3)

- This leads to the problem of collisions. It might be that two different customer rows are mapped to the same array entry.
- There are different strategies for solving the problem, e.g.:
  - Use the next free entry.
  - Attach to every array entry a linked list of all customer rows which should be stored in this array location.
- When you use hash functions on string-valued attributes, such collisions are unavoidable, since you cannot have one array element for every possible string.
- For a good hash function, it should be equally likely that every array entry is hit (e.g. returning always 1 is bad).

Hash Tables on External Storage

- In the customer example, 22 rows fit into one block.
- Then the hash table entries 1..22 are stored in the first block, the entries 23..44 in the second block, and so on.
- For collisions, we use a combined strategy:
  - Any free entry in the same block can be used.

    Especially, since the rows are of variable length, we simply try to fit all rows into this block which belong there. The exact table entry is no longer important.
  - If the block is full, overflow blocks are chained to this block. This decreases performance (one block access is no longer sufficient) and should not happen too often.

Hash-Tables in Oracle (1)

- Oracle has two different kinds of clusters: Index Clusters (see above) and Hash Clusters.
- Hash clusters must define the total number of different entries (the hash table size).
- E.g. you expect that you will have 500 000 customers, and that each customer row needs about 77 bytes:

```
CREATE CLUSTER Customer_Clust(CustNo NUMBER(7))
        SIZE 77
        HASHKEYS 500000 HASH IS CustNo
        STORAGE(INITIAL 50M NEXT 5M)
        PCTFREE 10 PCTUSED 80
```

Hash-Tables in Oracle (2)

- From the SIZE parameter, Oracle computes how many entries fit into one block. In the example, the result will be around 22.
- Then rows with CustNo between 1 and 22 will be stored in the first block, rows with CustNo between 23 and 44 in the second block, etc.
- So if customer numbers only start with 1001, use

  ```
  HASHKEYS 500000 HASH IS (CustNo - 1000)
  ```

- You do not have to specify a hash function:

  ```
  HASHKEYS 500000
  ```

  Then Oracle will use its internal hash function.

Summary (1)

- Hash clusters can bring significant performance improvements for queries containing equality conditions, e.g. CustNo = 7356, or joins.
- Since a hash cluster contains the actual rows, and not the ROWIDs, you can have hash access to a table only for one attribute (or attribute combination).

  The attribute does not have to be the key.
  However, it should normally not be updated.

- Hash-based indexes are of no help for range queries.

  The rows are not stored in sorted order. Also, when you use an attribute combination, an equality condition for only one attribute does not allow to compute the hash function.

Summary (2)

- Hash-based indexes are very bad for full table scans.

  Unless the hash table is nearly full, but then you will probably have many overfow blocks.

- As in index-based clusters, you need to know how much space every distinct value in the indexed column will need.

  And this should not be more than one block.

- But in addition, you also need to know how many distinct values there will be (more or less the table size).
- So hash clusters are very inflexible when the table grows. You probably have to recreate it from time to time with a larger value for HASHKEYS.

Outlook

- The literature and textbooks also contain "dynamic hash methods" or "external hash methods".
- These methods can adapt to changes in the table size without the need to reconstruct the entire table.

  Typically, they split a block and need to rehash only the entries in this block. In this way they avoid chained blocks.

- These methods need to keep some additional information, so they might need more than one block access.
- But normally this information can be kept in main memory.
- Oracle8 only contains the standard static hash method described above.

Overview

Motivation (1)

- Suppose you have sales data for the last three years:
  `SALES(Year, Month, Region, Amount)`
  where `Year` only has the values 1997, 1998, 1999.
- Then it is an option to create instead three tables, e.g. `SALES_1997(Month, Region, Amount)` and define:

```
CREATE VIEW SALES(Year, Month, Region, Amount)
AS          SELECT 1997, Month, Region, Amount
            FROM SALES_1997
UNION ALL   SELECT 1998, Month, Region, Amount
            FROM SALES_1998
UNION ALL   SELECT 1999, Month, Region, Amount
            FROM SALES_1999
```


Motivation (2)

- Consider a query which specifies a value for year, e.g.

```
SELECT SUM(Amount)
FROM   SALES
WHERE  YEAR = 1998
```

- Then a sufficiently intelligent optimizer will do only a full table scan of `SALES_1998`, whereas otherwise it would have to read the entire `SALES` table.
- Also, in order to delete all data about 1997, you simply redefine the view `SALES` and drop the table `SALES_1997`.

  In contrast, `DELETE FROM SALES WHERE YEAR=1997` takes a lot of time and rollback segment space, and does not make the table and indexes any shorter.

Motivation (3)

- There is the small problem that the view is not updatable in SQL-92. So when you insert tuples, you have to insert it into the right `SALES_*` table.
- However, a `UNION ALL` is very fast to evaluate. So even a full table scan over all `SALES_*` tables is not slower that if we had used only a single `SALES` table.
- You can also place the different `SALES_*` tables on different disks and get a kind of "striping" (even parallel evaluation might be possible).
- A disadvantage is that you must administer 3 tables now.
- Also foreign keys referencing `SALES` are no longer possible.

Motivation (4)

- Obviously, this kind of partitioning is only effective when a column contains a very small number of different values.
- However, a very good optimizer can also make use of CHECK-constraints (semantic optimization):

```
CREATE TABLE SALES1(
             YEAR NUMBER(4)
             CHECK(YEAR BETWEEN 1985 AND 1989),
             ...)
CREATE VIEW  SALES AS
             SELECT * FROM SALES1 UNION ALL
             SELECT * FROM SALES2
```


Partitioned Tables in Oracle

- The above examples should have run in Oracle 7.3.

  You have to buy the Partitioning option and use ALTER SESSION SET PARTITION_VIEW_ENABLED = TRUE.
- Oracle 8 has a new syntax for partioned tables:

```
CREATE TABLE SALES(YEAR NUMBER(4), ...)
PARTITION BY RANGE (YEAR)
(PARTITION SALES1 VALUES LESS THAN (1998),
 PARTITION SALES2 VALUES LESS THAN (1999),
 PARTITION SALES3 VALUES LESS THAN (MAXVALUE))
```

- You can specify tablespace etc. separately for each part.
- When looking at query evaluation plans, I didn't see any improvement.

Vertical Partitioning

- Splitting the rows of a table into multiple tables is called horizontal partitioning.

  Be careful. Some authors use the opposite names.
- You can also partition a table vertically. E.g. consider:

  Course(<u>CRN</u>, Title, Instr, Time, Room, Syllabus)
- If Syllabus is a longer text, but seldom accessed, it will make full table scans of Course unnecessary slow.
- So you could split the table into:

  Course_Data(<u>CRN</u>, Title, Instr, Time, Room)

  Course_Syllabus(<u>CRN</u> → Course_Data, Syllabus)
- You can define Course as a view with an outer join.

  Syllabus can be null. Enforcing its definition is difficult.

Overview

1. Clusters
2. Hash-Based Indexes
3. Partitioned Tables
4. Bitmap Indexes
5. Index-Organized Tables

Bitmap Indexes (1)

- B-tree indexes are effective when a column has many different values, i.e. each value appears only in a small number of rows.
- Bitmap indexes are useful when a column has few distinct values, i.e. the same value appears in a large number of rows. E.g. male/female, family status, region (east, central, west).

  Oracle suggests that "if the values in a column are repeated more than a hundred times, then the column is candidate for a bitmap index."
- Bitmap indexes are often useful for data warehouse applications.
- Oracle supports bitmap indexes only in its enterprise version.

Bitmap Indexes (2)

- Suppose a bitmap index was created on attribute A of table R.
- For each distinct value a that occurs in column A of R, the system stores a bitmap B_a (array of boolean values).
- The bitmap contains an entry for each row r in the table R:

$$B_a[r] = \begin{cases} 1 & \text{if } r.A = a, \\ 0 & \text{if } r.A \neq a. \end{cases}$$

- The DBMS evaluates e.g.

```
SELECT * FROM R WHERE A = 'a'
```

  by searching the bitmap for 1 bits.

Bitmap Indexes (3)

- Each position in the bitstring can be translated into a ROWID which can be used to fetch the row from the table.
- In order to do this, Oracle stores a fixed number of bits per block (before compression), namely as many bits as there can be rows per block.

  The command `ALTER TABLE` R `MINIMIZE ROWS_PER_BLOCK` computes the currently maximal number of rows per block and ensures that future insertions cannot increase the number. It can be used only before creating bitmap indexes on the table. If one does not use this command (or cancels it with `NOMINIMIZE`), Oracle somehow computes the maximum number of rows possible. This is more effective for a table that has attributes of fixed length.

Bitmap Indexes (4)

- E.g. consider an index over `CUSTOMERS(STATE)`. If the table has 2 Million rows, the bitstring for each state needs 244 KB (as compared to the table that needs 230 MB disk space).

  A bitmap index works by compressing a tuple of e.g. 100 Byte to a single bit for the purpose of evaluating a condition like `STATE = 'PA'`.
- In addition, Oracle compress the bitmaps before storing them.

  The compression algorithm is not disclosed. E.g. it would be possible to replace long sequences of 0-bits simply by their number. Oracle states that a bitmap consists of a number of segments (none of which is larger than half the block size). These segments have to be sorted if one needs to merge bitmaps (i.e. wants the bits in the storage order).

Bitmap Indexes (5)

- Already the ROWIDs for an average state (40 000 customers) in a B-tree index would need 240 KB (6 Byte per ROWID).

    If one counts only the ROWIDs, and does not consider the compression of bitmaps, the bitmap index is more space effective for columns that contain 48 distinct values or less.

- The Oracle Tuning Manual contains an example of a table with 1 Million rows having a B-tree index of 15 MB, where the corresponding bitmap index is smaller even for 500 000 different values in the column (each value appears only twice).

    For 100 000 different values, the bitmap index is about 5 MB.

- Normally, one would use a bitmap index only for attributes with a limited number of distinct values.

Bitmap Indexes (6)

- The real power of bitmap indexes is that complex conditions (with logical connectives AND, OR, NOT) can be executed on bitstrings before the relation itself is accessed.

    Of course, one could in principle do the same with ROWIDs (e.g. ORACLE sometimes intersects ROWIDs from different indexes before accessing the table), but operations on bit strings are much faster (specially supported by the hardware).

- E.g. consider the query:

```
SELECT COUNT(*)
FROM   CUSTOMERS
WHERE  (STATE = 'PA' OR STATE = 'NY')
AND    SEX = 'M' AND FAMILY_STATUS <> 'SINGLE'
```


Bitmap Indexes (7)

- If there are bitmap indexes for the three attributes, the DBMS would compute the bit-or of the bitmaps for `STATE = 'PA'` and `STATE = 'NY'`, then bit-and the result with the bitmap for `SEX = 'M'` and finally do a bit-and with the complement of the `FAMILY_STATUS = 'SINGLE'` bitmap.

- It would then count the number of "1" bits in the result and not access the table "CUSTOMERS" at all.

    So in the end it will have read about 1 MB instead of 230 MB for the full table scan.

- If for another query it would be necessary to access the table in the end, the bitmap index at least returns the ROWIDs in the storage order on disk.

Bitmap Indexes (8)

- Bitmap indexes are created with a command like the following

```
CREATE BITMAP INDEX I_CUST_STATE
ON CUSTOMERS(STATE)
```

- The standard storage parameters can be added.

- Bitmap indexes cannot be `UNIQUE`.

- If one wants a more compact storage format, one can use

```
ALTER TABLE CUSTOMERS MINIMIZE RECORDS_PER_BLOCK
```

    after the table contains a representative set of rows and before the first bitmap index is created.

- If the column allows null values, there will be one bitmap for the value null (Oracle's B-tree indexes do not list null values).

Bitmap Indexes (9)

- One can create bitmap indexes con column combinations, but is is probably uncommon: Bitmap indexes on different columns are easy to combine.

  The efficient merging of bitmaps is one of the strengths of this type of indexes.

- In Oracle, bitmap indexes are not good for heavily updated tables (OLTP applications).

  The problem is that an update will lock an entire “segment” of the bitmap that might represent e.g. 1000 rows. In a B-tree index, an update locks only a single row.

- Oracle will use bitmap indexes only with its newer, rule-based optimizer (use `ANALYZE TABLE` to make statistics available).

Overview

Index-Organized Tables: Structure

- As an alternative to storing the table rows in a heap file and letting ROWIDs from a B-tree index point to these rows, Oracle can store the entire rows in a B-tree index.
- This is called an index-organized table (IOT).
- An IOT is structured like a `UNIQUE` index for the primary key of the table, but instead of containing a ROWID, it contains the other (non-key) columns.

  A primary key must be specified for index-organized tables.

- An index-organized table can be declared in the following way:

```
CREATE TABLE CUSTOMERS(..., PRIMARY KEY(CUSTNO))
ORGANIZATION INDEX
TABLESPACE USER_DATA STORAGE(...)
```


Index-Organized Tables: Advantages

- When an attribute value is found in a standard index, the DBMS must still look up the corresponding row from the heap file (unless index-only QEP). In the IOT, the DBMS directly finds the row (saves at least one block access).
- Range scans in standard indexes result in ROWIDs scattered over the entire heap file. With the IOT, the rows are physically stored in ordered sequence (in the same block or in few blocks: may save many block accesses).

  A full index scan returns rows in sorted sequence.

- Less space is required: With a heap file, the indexed attribute values are stored twice. Also the ROWIDs require storage space, the overhead is doubled (for index and heap file).

Index-Organized Tables: Restrictions

- The rows in an index-organized table have no ROWIDs: They are moved around when B-tree blocks are split, so they have no stable physical address.
- Since the rows have no ROWID, no other indexes can be built on the same table.

  Besides the index on the primary key in which the rows are stored. Of course, if the primary key is a composed key, e.g. (A, B, C), then the index can also be used with only values for A or (A, B).
- Therefore, also no alternative keys (`UNIQUE`-constraints) can be declared.