Part 3: Physical Storage of Relations

References:

- Ramez Elmasri, Shamkant B. Navathe: Fundamentals of Database Systems, 3rd Edition. Section 5.5,5.7.
- Raghu Ramakrishnan, Johannes Gehrke: Database Management Systems, 2nd Edition. Section 7.3, 7.5–7.8.
- Silberschatz/Korth/Sudarshan: Database System Concepts, 3rd Ed., Chap 10.
- Kemper/Eickler: Datenbanksysteme (in German), Chap. 7, Oldenbourg, 1997.
- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom: Database System Implementation. Chapter 3.
- Theo Härder, Erhard Rahm: Datenbanksysteme, Konzepte und Techniken der Implementierung (in German).
- Michael J. Corey, Michael Abbey, Daniel J. Dechichio, Ian Abramson: Oracle8 Tuning.
- Jason S. Couchman: Oracle8i Certified Professional: DBA Certification Exam Guide with CDROM. Osborne/ORACLE Press, ISBN 0-07-213060-1, ca. 1257 pages, ca. $99.99.
- Mark Gurry, Peter Corrigan: Oracle Performance Tuning, 2nd Edition (with disk).
- Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques.
- Oracle 8i Concepts, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76965-01.
- Oracle 8i Designing and Tuning for Performance, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76992-01.

Overview

ROWIDs/TIDs (1)

- ROWIDs (row identifiers) are physical pointers to rows. They are called also TID (tuple identifier).
- Indexes provide a fast way to look up the ROWIDs of those table rows that contain a given value in a certain column.

  An index over column `A` of a table `R` can be understood as an auxillary table `I(A,ROWID)`. The first column contains all data values that currently appear in `R.A`, the second column contains the ROWIDs of the matching rows in `R`. The index is not organized as a heap file, but e.g. as a B-tree, which gives fast access to the entry for a specific value (see below). One could organize the original table as a B-tree, but then only one attribute could be indexed (since B-trees basically store the entries sorted by `A`).

ROWIDs/TIDs (2)

- So the Row Manager must support two ways to access a row:
  - ◇ Read all rows of the table in a full table scan.
  - ◇ Get a particular row given its address (ROWID).
- The access via the ROWID should be especially fast, i.e. normally only a single block access.
- Therefore, ROWIDs usually contain the physical address of the block in which the row is stored.

  I.e. the file number and the block number within the file. Plus e.g. the number of the row within the block.

ROWIDs/TIDs (3)

- Most DBMS guarantee that ROWIDs/TIDs do not change for the entire lifetime of a tuple.

  Except when the tuple is exported and imported again. That would basically create a new row with the same values.

- The reason that ROWIDs should be kept stable is
  - ◇ there can be many indexes for the same table. If the ROWID of a tuple should change, all would have to be updated.
  - ◇ some DBMS (e.g. Oracle) make ROWIDs available on the user level.

Oracle ROWIDs (1)

- In Oracle, every table has a "pseudocolumn" `ROWID`, which can be queried like a real column:

```
SELECT ROWID, FIRST, LAST
FROM   STUDENTS
```

- The column is not listed with `describe` or `SELECT *`.
- It is not possible to update the column `ROWID`.

  It is not stored, but computed from the storage position of the row.

- The pseudocolumn can also be used in conditions:

```
SELECT FIRST, LAST
FROM   STUDENTS
WHERE  ROWID = 'AAACiMAACAAAAYnAAA';
```

Oracle ROWIDs (2)

- An Oracle8 ROWID consists of:
  - ◇ `SUBSTR(ROWID,1,6)`: Data object number.

    This identifies the segment. I do not see why it is necessary. Old Oracle 7 ROWIDs did not contain this part. The data object number is e.g. shown in `USER_OBJECTS`.
  - ◇ `SUBSTR(ROWID,7,3)`: Relative file number.
  - ◇ `SUBSTR(ROWID,10,6)`: Block number in the file.
  - ◇ `SUBSTR(ROWID,16,3)`: Row number in the block.
- A base 64 encoding is used for the numbers.

  Six bits per character (0–63) are coded using the characters `A-Z`, `a-z`, `0-9`, `+` and `/`. E.g. `AAC` is the number 2.

Oracle ROWIDs (3)

- There is a package of stored functions for decoding the components of a ROWID:

```
SELECT DBMS_ROWID.ROWID_OBJECT(ROWID),
       DBMS_ROWID.ROWID_RELATIVE_FNO(ROWID),
       DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID),
       DBMS_ROWID.ROWID_ROW_NUMBER(ROWID),
       FIRST, LAST
FROM   STUDENT
```

- Rows in a block are numbered 0, 1, 2, . . .

  Holes in the sequence are numbers of deleted rows.

- By querying and decoding the `ROWID`, it is possible to find out where a particular row is stored.

Variable-Length Rows (1)

- Often rows in a table have a variable size.

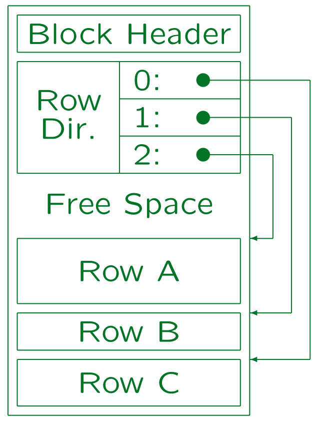
  E.g. because of `VARCHAR` columns.

- Then rows can also grow or shrink via updates.
- Oracle treats all rows as variable-length.

  Since columns can be added to a table with `ALTER TABLE`, one must either copy the entire table at this point, or abandon the idea of fixed-length rows. Also when null values should be stored with less space than the normal column value, the row length becomes variable.

- Variable-length rows are usually managed in a block with a row directory, i.e. a small table giving the offsets (start addresses) of the rows in the block.

Variable-Length Rows (2)



Variable-Length Rows (3)

- The ROWID consists of file number, block number, and the index in the row directory.
- The indirect addressing via the row directory makes it possible that rows are moved within the block:
  - ◇ E.g. Row B is updated and grows slightly.

    Then Row A has to be moved towards the beginning of the block (where there is still free space) to make room.

  - ◇ Or suppose that Row B is deleted.

    Then Row A would be moved towards the end of the block, such that the free space is not fragmented. However, most systems including Oracle merge free space only if necessary to insert a new row.

Variable-Length Rows (4)

- The block header may e.g. contain
  - ◇ Block address, type of segment, table name.
  - ◇ The size of the row directory, size of free space.
  - ◇ Next block in the list of blocks with free space.
  - ◇ A serial version number for this block which is incremented for every update.

    This is needed for crash recovery.

  - ◇ A bit pattern to detect partially written blocks.

    The pattern at the begin and end of the block must agree, they are both inversed on every write.

Variable-Length Rows (5)

- Block overhead in Oracle: ca. 84–107 Byte.
  (Gurry/Corrigan use 90 Byte in computations.)
- In Oracle, the row directory needs two bytes per entry.

  Oracle never releases elements of the row directory. If at some point in time, 50 rows were stored in the block, the row directory will always need 100 bytes, even if it contains only a single row. Of course, if the row is stored in location 50, there is would be in any case no way to shorten the row directory, because the ROWID must be kept stable.

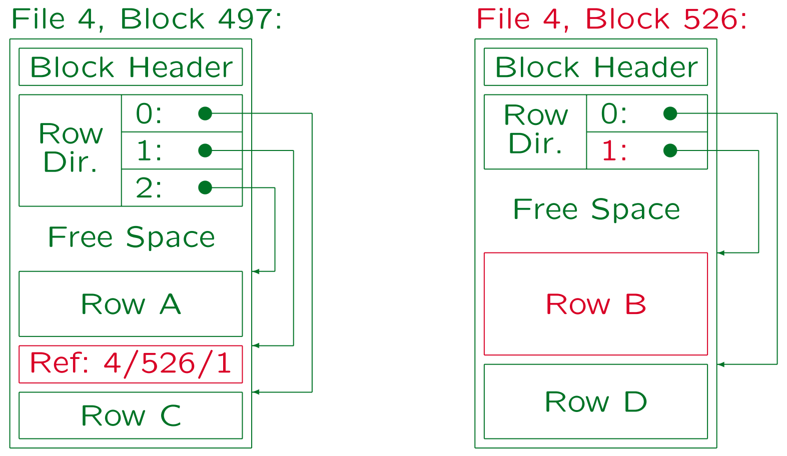
Variable-Length Rows (6)

- If a row grows and there is not enough free space left in the block, it must be moved ("migrated") to another block.
- A pointer must be left behind in this block so that the row can still be found via its ROWID.

  Thus, e.g. its entry in the row directory is still used.
- So now two block accesses are needed in order to retrieve this row, given its ROWID.

  This decreases performance, especially since the row might be stored far away on the disk.

Variable-Length Rows (7)



Variable-Length Rows (10)

- Oracle can also store rows in multiple pieces in different blocks ("chained rows").
- This is only done for rows longer than a block. If the row fits in a block, it is completely moved to another block.
- If there are many chained rows, consider increasing the `DB_BLOCK_SIZE` (requires recreation of the DB).

  Depending on the version, the default size might be 2KB. The block size should be a multiple of the OS block size (often 4KB or 8KB). The parameter can only be set when the database is created. A block size which is too large can decrease the performance for accesses to single rows (e.g. via an index) and also the caching performance.

Overview

Row Format (1)

- Normal row format in Oracle (not chained, not clustered):

Row Header	1st Col. Length	1st Col. Data	2nd Col. Length	2nd Col. Data	...

- The row header contains the number of columns and the number of chain pieces (3 bytes in total).
- The column length is encoded in one byte if below 250. Otherwise it needs three bytes.

Row Format (2)

- The length of the column data depends on the data type. E.g. a `VARCHAR`-string with 5 characters needs 5 byte.

  See below for more information.

- In the order of columns is normally the order of declaration in the `CREATE TABLE` statement.

  But `LONG` columns are moved towards the end. Columns added with `ALTER TABLE` are also added at the end.

- Null values need only the length byte (0).

  If the columns at the end are all filled with null values, they are not stored at all.

Overview

Data Formats (1)

- The storage size of any data value can be determined with the function VSIZE:

```
SELECT SSN, VSIZE(SSN), LNAME, VSIZE(LNAME)
FROM   STUDENT
```

- This is also possible without storing the value:

```
SELECT VSIZE(-1.2), VSIZE('abc')
FROM   DUAL
```

- To see the internal representation of e.g. 123, use

```
SELECT DUMP(123, 16) FROM DUAL
```

The bytes are printed in hexadecimal notation (selected with the argument 16). This also works with other data types, e.g. DUMP('ab',16).

Data Formats (2)

- CHAR(n): A fixed-length string is stored in n Bytes (one character per byte, filled with blanks to the length n).
- VARCHAR(n): Here only the actual characters are stored. (If a VARCHAR(10) column contains 'Jim', it needs 3 Byte.)

Data Formats (4)

- NUMBER(p), NUMBER(p,s): Numbers are stored in scientific notation with mantissa and exponent. E.g. $123 = 1.23 * 10^2$.

  It seems that Oracle really stores it as $1.23 * 100^1$.
  Note that NUMBER(p,s) is an Oracle-specific synonym for NUMERIC(p,s).

- The exponent needs always one byte, the mantissa needs one byte per two digits (leading/training zeros are not stored).

  Even if the column is NUMBER(30), 123 needs only 3 Byte.

Data Formats (5)

- So a positive number with n digits needs

$$1 + \text{ceil}(n/2)$$

  bytes.

  The Oracle 8 Concepts manual says something different.

- Negative numbers need one more byte for the sign.
- Oracle can store up to 38 significant digits, so a number needs at most 21 Byte (or 20 Byte if positive).

Data Formats (6)

- ROWID: Physical pointer to a row, needs 10 bytes.

  Maybe: Object 4 Byte, File+Block 4 Byte, Row 2 Byte (?).

- DATE: Timestamp (Date and Time), needs 7 Byte.

  Year: 2 Byte, Month: 1 Byte, Day: 1 Byte, Hours: 1 Byte, Minutes: 1 Byte, Seconds 1 Byte. In the default format for input/output (DD-MON-YY) only the date portion can be specified and Oracle assumes 0:00am (midnight). However, SYSDATE returns not only the current date, but also the time.

Part 4: B-Tree Indexes

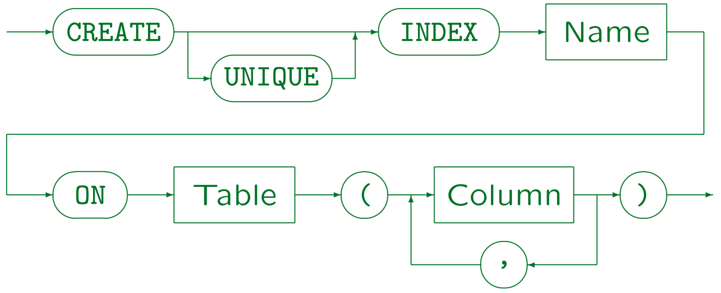
References:

- 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.

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).

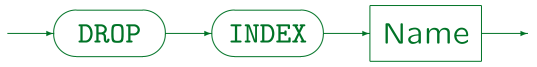
Indexes in SQL (4)

- 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 (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.

Part 5: 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.

Overview

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 two 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 two 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.

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

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 (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.

Overview

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).

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.