DBA Certification Course

(Summer 2008)

Chapter 4: Database Objects

- Data Types
- Integrity Constraints
- Generated Columns
- Temporary Tables

Objectives

After completing this chapter, you should be able to:

- write `CREATE TABLE` statements in DB2
- use generated columns in DB2

Literature (1)

- Hana Curtis: DB2 9 Fundamentals exam 730 prep, Part 5: Working with DB2 objects

  [http://www.ibm.com/developerworks/edu/dm-dw-db2-cert7305.html]
- Clara Liu, Raul Chong, Dwaine Snow, Sylvia Qi: Understanding DB2: Learning Visually with Examples

  IBM Press/Pearson, 2005, ISBN 0-13-185916-1, 895 pages.
- DB2 for Linux, UNIX, and Windows Version 9 Information Center

  [http://publib.boulder.ibm.com/infocenter/db2luw/v9//index.jsp]

Literature (2)

- Don Chamberlin: A Complete Guide to DB2 Universal Database

  Morgan Kaufmann, 1998, ISBN 1-55860-482-0, 795 pages.
- Roger E. Sanders: DB2 Universal Database V8.1 Certification Exam 700 Study Guide

  Prentice Hall / IBM Press, 2004, ISBN 0-13-142465-3, 416 pages.

Overview

1. Data Types
2. Tables
3. Indexes
4. Triggers
5. Typed Tables

Built-in Data Types (1)

Numeric Types:

- `DECIMAL(`p`,`s`)`: Decimal number with p digits in total, of these p places to the right of the decimal point.

  E.g. `DECIMAL(2,1)` permits values from `-9.9` to `+9.9`. `NUMERIC(`p`,`s`)` is a synonym for `DECIMAL(`p`,`s`)`. If s is omitted, it defaults to `0`. If p is omitted, it defaults to 5. One can also write `DEC` or `NUM` instead of `DECIMAL`. Values of type `DECIMAL(`p`,`s`)` are stored in $\lfloor p/2 \rfloor + 1$ bytes.

- `SMALLINT`: 16 bit binary integer (-32 768 to +32 767)
- `INTEGER`: 32 bit (-2 147 483 648 to +2 147 483 647)
- `BIGINT`: 64 bit (more than 18 decimal places)

Built-in Data Types (2)

Numeric Types, continued:

- `REAL`: 32 bit floating point number

  `FLOAT(`n`)` with $1 \leq n \leq 24$ is a synonym for `REAL`. The parameter n is the minimum number of bits for the mantissa.

- `DOUBLE`: 64 bit floating point number

  `FLOAT(`n`)` with $25 \leq n \leq 53$ is a synonym for `DOUBLE`. One can also write `DOUBLE PRECISION` or simply `DOUBLE`.

Built-in Data Types (3)

String Types:

- `CHAR(`n`)`: Fixed length single-byte character string.

  A value of type `CHAR(`n`)` needs n bytes. The length n can be at most 254. It defaults to 1.

- `VARCHAR(`n`)`: Variable-length single-byte string.

  A value of type `VARCHAR(`n`)` needs $m + 4$ bytes, where m is the actual length ($m \leq n$). The maximum `VARCHAR`-length is 32 672 (but a row must fit in one page, and `VARCHAR` arguments in `SYSFUN` functions are restricted to 4000 bytes).

- `GRAPHIC(`n`)`: Double byte, fixed length string.

  E.g. for asian languages. Maximum n is 127.

- `VARGRAPHIC(`n`)`: Double byte, variable length string.

Built-in Data Types (4)

Codepage:

- When a database is created, a codepage and a territory are specified:

```
CREATE DATABASE MYDB ON D:
USING CODESET 1252 TERRITORY US
COLLATE USING SYSTEM
CATALOG TABLESPACE MANAGED BY DATABASE
    (FILE 'E:\SYSCAT.DAT', 5000)
```

- All string data is encoded in the specified codepage.

  If this is a single byte codepage, the database is called a single byte database, and the `GRAPHIC`/`VARGRAPHIC` types cannot be used. If it is a double byte codepage, types like `VARCHAR` can be used, but they contain a mixture of single byte and double byte characters.

Built-in Data Types (5)

Unicode in DB2:

- New databases should be created as Unicode databases (parameter `CODEPAGE` should be `UTF-8`), because this is required for XML support.
- Then `CHAR`, `VARCHAR`, `LONG VARCAHR`, `CLOB` are stored in UTF-8, and `GRAPHIC`, `VARGRAPHIC`, `LONG VARGRAPHIC`, `DBCLOB` are stored in UCS-2.

  UCS-2 means that all characters have a fixed length of 16 bit (this normally means that only characters of the "Basic Multiligual Pane" can be represented, not arbitrary Unicode characters. However, it seems that surrogate pairs can be safely stored, they are only treated as two characters in some contexts.)

Built-in Data Types (6)

Unicode in DB2, continued:

- In UTF-8, one character needs 1 to 4 bytes.

  One byte is needed for ASCII characters, 2 bytes suffice e.g. for ISO Latin-1 characters (and more).

- The parameter n in e.g. `VARCHAR(`n`)` specifies the maximal length in bytes.
- Functions like `CHARACTER_LENGTH` have a second parameter with values `OCTETS`, `CODEUNITS16`, `CODEUNITS32`.

  E.g. `VALUES(CHARACTER_LENGTH('Jörg', OCTETS))` gives `5`.

Built-in Data Types (7)

Binary Data:

- For character data, the DBMS might do a conversion if client and server use different encodings.
- This would destroy binary data.
- Therefore, it is important to add "`FOR BIT DATA`" to `CHAR`-types that are actually used for binary data:

```
CREATE TABLE PROFESSOR(
        NAME VARCHAR(40) NOT NULL PRIMARY KEY,
        PICTURE VARCHAR(4000) FOR BIT DATA)
```

  4000 bytes might be too restricted, see `BLOB` below.

Built-in Data Types (8)

Long Objects:

- It is possible to store very long data values (files) in the database as one of the large object types:
  - ◇ `CLOB(`n`)`: Character Large Object
  - ◇ `DBCLOB(`n`)`: Double Byte Character Large Object
  - ◇ `BLOB(`n`)`: Binary Large Object
- The parameter n defines the maximal size in bytes, e.g., `500K`, `10M`, or `1G`. The maximal size is `2G`.

  Of course, only the necessary space is actually used, so the maximal size is mainly a constraint. But see also Slide 4-16.
- `LONG VARCHAR/VARGRAPHIC`: Old types for long data.

Built-in Data Types (9)

Long Objects, continued:

- Limitations for working with long objects:
  - ◇ One cannot do direct comparisons with =, <>, <, >, <=, >=, `IN`, `BETWEEN`. However, one can use `LIKE`.
  - ◇ For this reason, also duplicate elimination with `DISTINCT`, als well as `GROUP BY`, `ORDER BY`, and keys cannot be used.
  - ◇ No aggregation functions can be used.
  - ◇ Only `UNION ALL` can be used (not `UNION` etc.).
  - ◇ Indexes cannot be created on `LOB` columns.

Built-in Data Types (10)

Long Objects, continued:

- LOB values are stored separately from the other data of the row, possibly in a different tablespace:

```
CREATE TABLE PROFESSOR(
        NAME VARCHAR(40) NOT NULL PRIMARY KEY,
        PICTURE BLOB(10M))
        IN TBSP1
        INDEX IN TBSP2
        LONG IN TBSP3
```

  Note that if different tablespaces are used for the different parts of a table, these must be DMS tablespaces. The tablespace used for LOBs (`LONG` data) must be a large tablespace.

Built-in Data Types (11)

Long Objects, continued:

- Inside the table row, a LOB descriptor is stored that permits to locate the real data.

  The size of the descriptor depends on the declared maximum size n: It varies from 72 bytes ($n <$ `1K`) to 316 bytes ($n =$ `2G`).
- Application programs can work with LOB locators and thereby avoid to transfer the complete large object between server and client.

  Locators help to do as much as possible of the processing on the server, and also to defer the work, and actually avoid it for intermediate results: E.g. if one concatenates two LOBs, the locator refers to a data structure that describes this concatenation, but it is not actually done unless the value is stored as a new LOB in the database.

Built-in Data Types (12)

Long Objects, continued:

- A LOB column can be excluded from logging:

```
CREATE TABLE PROFESSOR(
        NAME VARCHAR(40) NOT NULL PRIMARY KEY,
        PICTURE BLOB(10M) NOT LOGGED)
```

- Normal ROLLBACK and crash recovery is still possible.

  This works with shadow pages: The old pages are only overwritten after the successful end of the transaction, when the new value is safely stored in new pages.

- However, in case of a disk failure, when the rollforward is done from an old backup copy, these LOB values are replaced by binary zero bytes.

Built-in Data Types (13)

Long Objects, continued:

- One can request compact storage of LOB values:

```
CREATE TABLE PROFESSOR(
        NAME VARCHAR(40) NOT NULL PRIMARY KEY,
        PICTURE BLOB(10M) COMPACT)
```

- This means that only the really needed space (rounded to the next multiple of 1K) is used.

  It seems that otherwise the next power of 2 (between 1K and 64M) is used. Note that `COMPACT` is not a compression.

- The price to pay is that updates to LOB values are less efficient with `COMPACT`.

Built-in Data Types (14)

Date and Time Types:

- `DATE`: Day, Month, Year.

  4 Byte. There are no special `DATE` constants, strings of the form `'DD.MM.YYYY'`, `'MM/DD/YYYY'` and `'YYYY-MM-DD'` are automatically converted. The output format depends on the declared territory for the database, one can also use the `DATETIME` option of the `BIND` command.

- `TIME`: Hour, Minute, Second.

  3 Byte. DB2 understands e.g. `'20:15:00'`, `20:15'`, `'20.15.00'`, `'20.15'` and `'8:15 PM'`.

- `TIMESTAMP`: Date and Time (including Microsecond).

  10 Byte. This has only a single format: `'YYYY-MM-DD-HH.MI.SS.SSSSSS'`.

Built-in Data Types (15)

Date and Time Types, continued:

- There are "special registers" (functions without parameters) that return the current date and time:
  - ◇ CURRENT DATE
  - ◇ CURRENT TIME
  - ◇ CURRENT TIMESTAMP

    This returns the current date and time when the processing of the query began. If the same special register is accessed several times during the evaluation of one query, the same value is returned.
  - ◇ CURRENT TIMEZONE

    Difference between Coordinated Universal Time (UTC) and local time on the server as a decimal number of the form HHMMSS.

Built-in Data Types (16)

Date and Time Types, continued:

- If one subtracts a DATE value from a DATE value, one gets a DECIMAL(8,0) value that encodes a date duration in the form YYYYMMDD.

  In the same way, one can subtract a TIME value from a TIME value (gives a DECIMAL(6,0), format HHMMSS), or a TIMESTAMP from a TIMESTAMP (gives a DECIMAL(20,6), format YYYYMMDDHHMISS.SSSSSS).
- A special syntax is available to add or subtract durations to/from the date/time types, e.g.

```
DATE_DUE + 7 DAYS > CURRENT DATE
```


Distinct Types (1)

- Distinct types are one form of user-defined types (UDTs). They are defined based on a built-in type:

```
CREATE DISTINCT TYPE EX_NO
        AS INTEGER WITH COMPARISONS
```

  The keywords "WITH COMPARISONS" are required for all types that support the standard comparison operators =, <>, <, >, <=, >= (i.e. all types except the LOB and LONG types). The keywords are supposed to remind the user that the comparison operators are inherited from the base type. Other operations like "+" are not automatically inherited.
- This means that EX_NO will internally have the same set of values as the built-in type INTEGER.

  It is not possible to exclude e.g. negative numbers.

Distinct Types (2)

- The values of the new type EX_NO are considered as distinct from the values of the source type INTEGER, and the values of any other distinct type:

```
CREATE DISTINCT TYPE POINTS
        AS INTEGER WITH COMPARISONS
```

  It is possible to have a column and a distinct type with the same name. But it might be clearer to name types e.g. with the suffix "_T".
- Table columns can be declared with a distinct type:

```
CREATE TABLE EXERCISES (
        NO    EX_NO       NOT NULL PRIMARY KEY,
        TOPIC VARCHAR(40) NOT NULL,
        MAXPT POINTS      NOT NULL)
```

Distinct Types (3)

- Now a meaningless comparison between exercise number and points gives a type error:

```
SELECT *
FROM   EXERCISES
WHERE  POINTS < NO     -- Type Error
```

Note that this would be possible if both columns were of type INTEGER.

- So distinct types help to detect errors and make the semantics of columns clearer.

Furthermore, the source type is defined only in one place. If we later in the design think that a SMALLINT would suffice, this can be easily changed in the schema creation SQL script. However, once the tables are created and filled, this change is not possible.

Distinct Types (4)

- EX_NO values are distinct from INTEGER values:

```
SELECT POINTS
FROM   EXERCISES
WHERE  NO = 1     -- Type Error
```

- DB2 automatically creates casting functions which must be explicitly called here:
  - ◇ FUNCTION EX_NO(INTEGER) RETURNS EX_NO
  - ◇ FUNCTION INTEGER(EX_NO) RETURNS INTEGER
- The WHERE-condition can be written: NO = EX_NO(1).
- Alternatively, this is also possible: INTEGER(NO) = 1.

Distinct Types (5)

- If the source type were SMALLINT, DB2 would generate three casting functions:
  - ◇ FUNCTION EX_NO(SMALLINT) RETURNS EX_NO
  - ◇ FUNCTION EX_NO(INTEGER) RETURNS EX_NO
  - ◇ FUNCTION SMALLINT(EX_NO) RETURNS SMALLINT

The reason is that constants like 1 are considered to be of type INTEGER (even if they would fit into a SMALLINT), but the conversion to an EX_NO must of course be possible. The automatic promotion of function arguments is only done in the direction SMALLINT → INTEGER → DECIMAL → REAL → DOUBLE. Also for source type CHAR(n) three casting functions are constructed, because string constants are supposed to have type VARCHAR(n). Automatic type promotion: CHAR → VARCHAR → LONG VARCHAR → CLOB. Note also that type name synonyms are replaced, e.g. for base type NUMERIC, the casting function is called DECIMAL.

Distinct Types (6)

- Of course, an explicit type conversion can also be used (this invokes the casting function):

```
SELECT POINTS
FROM   EXERCISES
WHERE  NO = CAST(1 AS EX_NO)     -- ok
```

- For assignments, e.g. in an UPDATE or INSERT statement, or in the INTO clause of a SELECT query, the rules are more liberal: Here a conversion between base type and distinct type is done automatically.

This simplifies application programs, where the host variables can only have a type that corresponds to a built-in type of DB2.

Distinct Types (7)

- No data type functions (except =, <, . . .) are inherited from the source type to the distinct type.
- E.g. one cannot add exercise numbers (which is good, because this would not make sense).
- However, it would make sense to add points.
- Then it is necessary to explicitly declare a user-defined function that performs addition of points.

  This function can be named "+", so that the infix operator + can also be used for points. DB2 supports the function call syntax also for operators when these are put into "...", e.g. one can write `SELECT "+"(1,2) FROM ....`

Distinct Types (8)

- One does not have to define a function implementation, but declares that the function "+" for integers can be used ("sourced function").

```
CREATE FUNCTION "+"(POINTS, POINTS) RETURNS POINTS
       SOURCE   "+"(INTEGER, INTEGER)
```

DB2 permits function overloading: There can be any number of functions with the same name in the same schema, provided that the list of argument types differ (when ignoring the parameters of types like `VARCHAR`). When a function is called, because of type promotions (see Slide 4-26) the argument types do not have to match exactly. The function is selected with the minimal promotion in the leftmost argument where there is a difference. After that, position of the schema on the function search path is used as a tie breaker.

Distinct Types (9)

- When the new function "+" is called,
  - ◇ the arguments are cast from `POINTS` to `INTEGER`,
  - ◇ the source function is called (the built-in function "+" for integers)
  - ◇ and the result is cast to the return type `POINTS`.

  It is not required that the specified source function operates exactly on the source type of the distinct type, it suffices if the required casts are possible. It is also not required that the two functions have the same name.

- One can write `VARCHAR()` and `DECIMAL()`, then the limits from the source function are taken.

Distinct Types (10)

- If one wants to multiply a `POINTS` value with a `DOUBLE` (e.g. for computing a percentage), one must either use explicit casts or declare a function:

```
CREATE FUNCTION "*"(POINTS, DOUBLE) RETURNS POINTS
       SOURCE   "*"(DOUBLE, DOUBLE)
```

- Also the required aggregation functions must be specifically declared for the new type:

```
CREATE FUNCTION AVG(POINTS) RETURNS DOUBLE
       SOURCE AVG(DOUBLE)
```

Distinct Types (11)

- Relevant data dictionary tables (schema `SYSCAT`): `DATATYPES`, `FUNCTIONS`, `FUNCPARMS`.
- Functions are uniquely identified by schema, name, and argument type list.
- Because in commands like "`DROP FUNCTION`" the argument list is not given, a second name, the "specific name" is assigned to each function.

  Functions cannot be called with the specific name. One can define a specific name after the return type with the keyword `SPECIFIC`. Otherwise the DBMS chooses one (see data dictionary).

User-Defined Functions

- One can also define external user-defined functions in languages like C or Java.
- Then also a `CREATE FUNCTION` statement is needed that refers to the object file and entry point:

```
CREATE FUNCTION COMP_GRADE(POINTS) RETURNS GRADE
       EXTERNAL NAME '/db/udf/points!comp_grade'
       DETERMINISTIC
       NO EXTERNAL ACTION
       LANGUAGE C
       PARAMETER STYLE DB2SQL
       NO SQL
```


Overview

CREATE TABLE: Repetition

- The basic syntax of the `CREATE TABLE` statement was explained in the course "Databases I":

```
CREATE TABLE COURSE(
       CRN NUMERIC(5) NOT NULL PRIMARY KEY,
       TITLE VARCHAR(80) NOT NULL,
       PROF_FNAME VARCHAR(20),
       PROF_LNAME VARCHAR(20),
       CREDITS NUMERIC(2) CHECK(CREDITS >= 0)
           DEFAULT 3,
       FOREIGN KEY (PROF_FNAME, PROF_LNAME)
           REFERENCES FACULTY ON DELETE CASCADE)
```

Default Values (1)

- In DB2, one can write `WITH DEFAULT` instead of only `DEFAULT` (as in the standard), but the keyword `WITH` is optional.
- In DB2, it is possible not to write a value after the keyword `DEFAULT`. Then the number 0, the empty string etc. is chosen ("system default value").

  For `CHAR(`n`)`, the empty string means of course n blanks. For date/time columns, the earlierst possible value is taken.
- However, if one does not use the `DEFAULT` specification at all, the default value is `NULL`.

Default Values (2)

- Note that specifying a non-null default value does not prevent the insertion of a null value, one only has to explicitly specify `NULL` in the `VALUES` list.
- The only safe way to prevent null values is to use the `NOT NULL` column constraint.

  One could also use a `CHECK`-constraint with an "`IS NOT NULL`" condition, but then DB2 still does not allow to use the column in a primary or unique key (which requires that the column is not nullable).

Generated Columns (1)

- One can let DB2 automatically generate values for an artificial key:

```
CREATE TABLE T(
        ID NUMERIC(4) NOT NULL PRIMARY LEY
            GENERATED ALWAYS AS IDENTITY
                (START WITH 1000 INCREMENT BY 1),
        A VARCHAR(10))
```

- Now insertions cannot specify a value for the column "`ID`", i.e. one must write

```
INSERT INTO T(A) VALUES ('First')
```

- This will insert the row `(1000, 'First')`.

Generated Columns (2)

- Each table can have at most one identity column.

  Identity columns must have type `SMALLINT`, `INTEGER`, `BIGINT`, or `DECIMAL` without fractional part (i.e. a whole number, not necessarily positive).
- One can also use "`GENERATED BY DEFAULT`":
  - ◇ Useful if the table is unloaded and loaded again.
  - ◇ A value is generated only when the insertion command does not specify one.
  - ◇ Each table has a counter for the next value to be generated. This is not changed if a value is explicitly specified. Thus, values generated in the future might violate the uniqueness.

Generated Columns (3)

- Besides `START WITH` and `INCREMENT BY`, there are:
  - ◇ `MAXVALUE` n: No value greater than n is generated.

    The default is `NO MAXVALUE`. There is also a `MINVALUE`. Sequences can be descending if `INCREMENT BY` is negative.
  - ◇ `CYCLE`: If the maximum value is reached, the generator starts again with the minimum value.

    The default is `NO CYCLE`: Trying to generate another value after the `MAXVALUE` was reached gives an error.
  - ◇ `ORDER`: Specifies that identity values must be generated in the order of request.

    The default is `NO ORDER`.

Generated Columns (4)

- Identity generation parameters, continued:
  - ◇ `CACHE` n: The DBMS pre-allocates n values of the sequence (and stores them in memory).

    Then only one log entry is necessary per n values (furthermore, it might be possible to avoid synchronous writes, where the application must wait until the write succeeded). However, when the system crashes or a decativation occurs because the last application disconnects, the cashed values are lost, i.e. there can be bigger holes in the generated sequence. The default is `CACHE 20`. One can turn this feature off with `NO CACHE`.
- Note that holes in the sequences are also possible when a transaction is rolled back: Once a value was generated, it is not recycled.

Generated Columns (5)

- One can also define computed columns:

```
CREATE TABLE T(
       TAG INTEGER NOT NULL,
       TAG_NAME VARCHAR(10) GENERATED ALWAYS AS
              (CASE A WHEN 1 THEN 'Monday'
                     ...                  END))
```

- However, it seems that a view would be preferable in this example.

  It is possible to call user-defined functions to compute the generated value (under certain conditions, e.g. without external action). If that function needs a lot of runtime, the generated column might be the better alternative. The defining expression cannot use subqueries or special registers (e.g. `CURRENT DATE` is not allowed).

Generated Columns (6)

- One important application of generated columns is that one can define indexes on them.

  E.g. if queries contain conditions of the form `f(A) = 100`, it is not possible to use an index on `A`, unless the system knows the reverse function. But one can define a column `B` generated as `f(A)`, and define an index on `B`. The DB2 optimizer notes that it can apply this index for the condition `f(A) = 100`, and even for certain conditions on `A`.
- If a column is generated with an expression, it must be `GENERATED ALWAYS` (not `GENERATED BY DEFAULT`).
- Generated identity columns are always considered `NOT NULL`, even if this is not explicitly specified.

  And even if it is only generated `BY DEFAULT`.

Sequences (1)

- A sequence works like the generator for identity columns, but it is not tied to a specific table.
- Thus, the generated values can be used in a more flexible way.
- When a sequence is created, the same parameters as for identity columns can be specified:

```
CREATE SEQUENCE S AS DECIMAL(5)
        START WITH 100
        INCREMENT BY 1
```

- If `AS` ⟨`Type`⟩ is left out, `INTEGER` is assumed.

Sequences (2)

- One gets the next value for a sequence with the expression:

```
NEXT VALUE FOR S
```

(where `S` is the name of the sequence).

- The last generated value is returned by

```
PREVIOUS VALUE FOR S
```

This is helpful if the same value must be entered in two or more places (e.g. an invoice numer is needed also in all line items).

- A sequence can be altered, e.g. with

```
ALTER SEQUENCE S RESTART WITH 200
```


Row Length (1)

- For the normal storage format the maximal row length is calculated as follows:
  - ◇ The lengths of the datatypes of the columns are added,

    See above. E.g.: `char`(n) needs n bytes, `varchar`(n) needs up to $n+4$ bytes (depending on actual string length), `integer` needs 4 bytes, `smallint` needs 2 bytes, `decimal`(p,s) needs $\lfloor p/2 \rfloor + 1$ bytes ($\lfloor \ldots \rfloor$ means "round down", e.g. `decimal(1)` needs 1 byte).
  - ◇ plus one additional byte for each column that can be null.
- Maximum row length for 4K pages: 4005 Byte.

Row Length (2)

- There is an alternative storage format, which can be chosen by adding "`VALUE COMPRESSION`" to the table declaration:

```
CREATE TABLE COURSE(
        CRN NUMERIC(5) NOT NULL PRIMARY KEY,
        ...)
        VALUE COMPRESSION
```

- This storage format permits to store null values and empty strings in a more compact way.

  It is not real value compression, only a different stoarge format for rows. In the meantime, DB2 also has a real compression.

Row Length (3)

Advantages of VALUE COMPRESSION:

- With VALUE COMPRESSION, a column that is null always needs 3 Byte, and empty VARCHAR, VARGRAPHIC and LOB columns need only 2 Byte.

  Without VALUE COMPRESSION, a nullable CHAR(10) column would always need 11 Byte, even if it is null.

- Variable-length data types are stored in a more compact way. E.g. a VARCHAR(n) column needs at most $n + 2$ Byte, even if it is nullable.

  Without VALUE COMPRESSION, it needs $n + 4$ Byte if it is not nullable, and $n + 5$ Byte if it can be NULL.

Row Length (4)

Disadvantages of VALUE COMPRESSION:

- If a null or empty column is later updated, the row grows, which might lead to migrated rows.
- The storage format for fixed-length data types like INTEGER and CHAR(n) needs two additional bytes, no matter whether the column is nullable or not.

  The normal storage format seems to assume that most columns have a fixed size, and variable-sized columns like VARCHAR(n) need an additional overhead. In the VALUE COMPRESSION storage format all columns are variable-size.

- Plus there is an overhead of 2 bytes per row.

Row Length (5)

Example:

- Consider the following table:

```
CREATE TABLE T(
       A INTEGER NOT NULL,
       B CHAR(10))
```

- Without VALUE COMPRESSION, this table has a fixed rowsize of $4 + 10 + 1 = 15$ Byte.
- With VALUE COMPRESSION, the rowsize is
  - ◇ normally $2 + 6 + 12 = 20$ Byte,
  - ◇ but only $2 + 6 + 3 = 11$ Byte if B is null.

Row Length (6)

COMPRESS SYSTEM DEFAULT:

- In the VALUE COMPRESSION row format, one can add "COMPRESS SYSTEM DEFAULT" to specific columns, e.g.

```
A INTEGER NOT NULL COMPRESS SYSTEM DEFAULT
```

  COMPRESS SYSTEM DEFAULT can only be used when VALUE COMPRESSION is specified for the table.

- Then the value 0 (the system default for numeric columns) is stored with only 3 Byte (instead of 6).

  For variable length string columns, this has no effect, since there the empty string is anyway stored in only 2 Byte. However, for CHAR(n), COMPRESS SYSTEM DEFAULT permits storage of the empty string in 3 Byte (compared to n Byte).

Space Requirements: Tables

- The DBMS needs 68 Bytes of overhead for each page. For 4K pages, this gives $4096 - 68 = 4028$.

  The overhead is the same for all page sizes. I do not understand why the maximum row size is 4005 for 4K pages. Even with the overhead of 10 Bytes per row (see below) it should be 4018.

- Number of rows per 4K page:

$$\lfloor 4028/(\text{average row size} + 10) \rfloor$$

- Number of pages needed for a table (estimate):

$$(\text{number of rows}/\text{number of rows per page}) * 1.1$$

  The factor 1.1 is for overhead. This is only an estimate. E.g., if row size varies significantly, more space may be needed.

Row Compression (1)

- Since Version 9, DB2 has also the possibility to compress table rows with a static dictionary-based compression algorithm.

  The advantages are disk space savings, and improved I/O and buffer performance (at the expense of more work for the CPU). Also log entries are compressed.

- Compression of up to 80% has been reported.

  It seems that even groups of columns can be compressed to a single byte. The size of the dictionary is relatively small, typically about 100 KB. It is normally kept in memory.

- Long, LOB and XML data cannot be compressed.

Row Compression (2)

- Compression is enabled with

```
ALTER TABLE T COMPRESS YES
```

- But this has no effect until a dictionary is built:

```
REORG TABLE T
```

  If there is already a dictionary, one can choose `KEEPDICTIONARY` or `RESETDICTIONARY`. A dictionary can also be built with the `INSPECT` command (gives an estimate for the compression rate).

- A table can have compressed and uncompressed rows. E.g. after `ALTER TABLE COMPRESS NO`, only new or updated rows are uncompressed.

  `REORG TABLE T RESETDICTIONARY`: uncompress all rows, delete dictionary.

Append Mode

- Tables can be placed in append mode:

```
ALTER TABLE T APPEND ON
```

- Then rows will only be inserted at the end.

  Otherwise, the free space map is searched and a first fit algorithm is used.

- This speeds up insertions and saves the disk space for the free space map.
- Furthermore, it leads to a clustering on the insertion order (e.g. on a generated key value).
- Of course, if rows are deleted, holes remain.

Restrict on Drop

- One can make accidental deletion of a table a bit more difficult:

  `CREATE TABLE T (...) WITH RESTRICT ON DROP`

  If the table exists already, use: `ALTER TABLE T ADD RESTRICT ON DROP`.

- Then "`DROP TABLE T`" gives an error message (and the table is not deleted).
- However, one can enter

  `ALTER TABLE T DROP RESTRICT ON DROP`

  and then drop the table.

Not Logged Initially (1)

- One can specify that changes to the table in the same transaction (unit of work) as the table creation are not written to the log:

  `CREATE TABLE T (...) NOT LOGGED INITIALLY`

  I.e. changes via `INSERT`, `UPDATE`, `DELETE`, `ALTER TABLE`, `CREATE INDEX`, `DROP INDEX` are not recorded in the log. However, catalog changes and storage reservations are logged.

- This is especially useful if a large table is to be filled from an external file.

  Then the data might be too large for the log. Furthermore, if something goes wrong, the external file still exists and the operation could be easily be repeated.

Not Logged Initially (2)

- If something goes wrong, the table is marked as inaccessible and can only be dropped.

  Of course, if the transaction is rolled back, the table creation is also rolled back, i.e. table is deleted. However, if there is a disk error later, and the rest of the database is restored from a backup (done before this transaction) and rollforward (using the log), then the table exists in the catalog, but the data is missing.

- After the transaction, one should do a backup.

  Then no rollforward with the missing log entries will be necessary.

- Note that all further transations are automatically recorded in the log.

  It is not necessary to explicitly stop the `NOT LOGGED` mode.

Not Logged Initially (3)

- If an error occurs during the execution of a command, the entire transaction is rolled back.

  E.g.. this happens if an insertion violates a key. Normally, one only gets an error message. The problem is that when the error is detected, the command might already be partially executed. With logging, the DBMS can execute something like `ROLLBACK TO SAVEPOINT`. This is not possible without logging.

- If one later wants this mode again, one can use

  `ALTER TABLE T ACTIVATE NOT LOGGED INITIALLY.`

  Note that if something goes wrong, the entire table is destroyed, including the old data. One can add "`WITH EMPTY TABLE`" if one wants to delete the table contents before the new data is inserted.

Create Table Like (1)

- One can create a table with the same schema as an existing table or view:

```
CREATE TABLE T LIKE S
```

- Then the new table `T` will have the same columns as the existing table/view `S`, with the same data types, nullability, and default values.

  One can add "`EXCLUDING COLUMN DEFAULTS`" if one does not want to copy this column characteristic. "`INCLUDING ...`" is the default.

  The LBAC security policy and protected columns are also inherited by the new table.

- The data is not copied. `T` will be empty.

Create Table Like (2)

- No other table charactistics are copied, e.g. `T` will not have any primary, unique, or foreign keys, and no indexes or triggers.

  Also physical specifications like the tablespace are not copied, they must be explicitly specified or the default is taken.

- It is possible to copy the settings for generated identity columns from the original table. This is done with

```
INCLUDING IDENTITY COLUMN ATTRIBUTES
```

  The last two keywords are optional. Here, the default is `EXCLUDING ...`. Then it will be a normal numeric column in the cloned table.

Create Table As

- It is also possible to specify the columns of a table by means of a query:

```
CREATE TABLE T AS
(SELECT A, B FROM S)
WITH NO DATA
```

  Instead of "`WITH NO DATA`" one can also write "`DEFINITION ONLY`".

- This is very similar to "`CREATE TABLE LIKE`".

  Of course, here, one can choose also a subset of the columns, or combine columns from different tables in a join query. "`CREATE TABLE AS`" is also used for defining materialized queries (views).

Temporary Tables (1)

- Temporary tables are local to a session.
- Even if parallel sessions declare temporary tables with the same name, each has its own instance.
- Therefore, no locking is needed.
- Temporary tables are automatically deleted when the session terminates.

  Or the database connection is interrupted.

- Temporary tables are not entered into the system catalog (data dictionary).

Temporary Tables (2)

- A session can contain several transactions.
- One can choose between `ON COMMIT PRESERVE ROWS` and `ON COMMIT DELETE ROWS` (the default).
- One can request that changes to temporary tables are not logged. This increases the efficiency.

  This is not the default, although it is very common for temporary tables. If changes are logged, a `ROLLBACK` works as for normal tables. If `NOT LOGGED` is specified, one has the choice between `ON ROLLBACK DELETE ROWS` and `ON ROLLBACK PRESERVE ROWS` (updates in the transaction are not undone for the temporary table, because the log information is missing). The creation or dropping of a temporary table is logged in any case, and this action is undone on `ROLLBACK`, however, without log information, the table will still be empty after undoing a `DROP TABLE`.

Temporary Tables (3)

- Temporary tables are declared, not created (probably because they are not persistent):

```
DECLARE GLOBAL TEMPORARY TABLE T (
        A INTEGER NOT NULL)
        ON COMMIT PRESERVE ROWS
        NOT LOGGED
```

- There are no local temporary tables in DB2.

  Even if a temporary table is declared in the code between `BEGIN` and `END`, it exists for the entire session, not only the block. Furthermore, the system sometimes uses internally temporary tables for the evaluation of a query (exist only locally during evaluation of a single query). Another explanation might be that this syntax conforms to the SQL standard, and the standard has also other forms of temporary tables.

Temporary Tables (4)

- A user temporary tablespace must exist before global temporary tables can be declared.

  And one must have the `USE` privilege for that tablespace, or `DBADM` or `SYSADM` authority. One can select a specific tablespace with `IN` as for normal tables.

- A temporary table is always in the schema "`SESSION`".
- So after the above temporary table was declared (even without explicit schema), data must be inserted in the form

```
INSERT INTO SESSION.T VALUES (1)
```


Temporary Tables (5)

- If the clause "`WITH REPLACE`" is added to the table declaration, a temporary table with the same name is automatically dropped, if one should exist.
- Keys and foreign keys cannot be used for temporary tables.
- Columns of temporary tables cannot be of long, LOB, or XML type.
- One can declare a temporary table `LIKE` another table, or `AS` the result schema of a query.

ALTER TABLE (1)

- The purpose of the ALTER TABLE command is to modify the schema of an existing table.
- In principle, one could copy the data to a temporary location, drop the table, create the table again with the modified schema, and copy the data back. But:
  - ◇ Copying the data is impractical for large tables.
  - ◇ Table entries may be referenced in foreign keys: One might end up recreating the entire DB.

    Also indexes, grants, views, triggers, stored procedures etc. reference tables. Some of this information will be lost when the table is dropped.

ALTER TABLE (2)

- Examples for table schema changes:
  - ◇ New columns can be added to a table.

    Because columns can be added, it is safer to specify columns in application programs instead of `SELECT *`.
  - ◇ The width of existing columns can be increased.

    E.g. from `VARCHAR(20)` to `VARCHAR(30)`.
    This is actually not possible in the SQL-92 standard, but in all three DBMS (Oracle, DB2, SQL Server).
  - ◇ Constraints can be removed or added.

    Some systems also have the possibility to disable a constraint, so that it is no longer checked, but still stored in the system. It can then later be enabled again.

ALTER TABLE (3)

- ALTER TABLE was not contained in SQL-86.
- It is contained in the SQL-92 standard, but concrete DBMS implementations differ quite heavily in the syntax and in what exactly can be changed.
- The SQL-92 standard offers these possibilities:
  - ◇ Columns can be added to or dropped from tables.
  - ◇ The default column value can be changed, but the data type cannot.
  - ◇ Constraints can be added or removed.

ALTER TABLE (4)

Adding Columns:

- E.g. add a column "EXTRA_PT" to "STUDENTS":

```
ALTER TABLE STUDENTS
      ADD COLUMN EXTRA_PT NUMERIC(4,1)
                          CHECK(EXTRA_PT >= 0)
```

- The keyword "COLUMN" is optional.
- The new column will first be null for all existing rows.
- It can then be updated to some other value.

  It might, however, create efficiency problems if the rows become much longer than they were when the were inserted.

ALTER TABLE (5)

- If a default value is specified, the new column can be NOT NULL:

```
ALTER TABLE STUDENTS
  ADD EXTRA_PT NUMERIC(4,1) DEFAULT 0 NOT NULL
```

- The new column is added as the last (rightmost) column of the table.

  There is no way to add it at any other position.

- Views (stored queries) are not affected, because SELECT * is replaced by by an explicit column list when the view definition is processed.
- There is no way to drop or rename a column.

ALTER TABLE (6)

Modifying Columns (SQL-92 and DB2):

- In SQL-92, the only allowed modification of a column is to change its default value:

```
ALTER TABLE EXERCISES
ALTER COLUMN MAXPT SET DEFAULT 12
```

- The default can be set to null also with this syntax:

```
ALTER TABLE EXERCISES
ALTER COLUMN MAXPT DROP DEFAULT
```

- In SQL-92, it is not possible to change the data type of a column. In DB2, it is (see next slide).

ALTER TABLE (7)

Modifying Columns (DB2):

- Earlier, the only modification of a column data type was to increase the size of a VARCHAR-column:

```
ALTER TABLE EXERCISES
      ALTER TOPIC SET DATA TYPE VARCHAR(100)
```

- In the meantime, DB2 has become very generous with changing the data type of columns (still only larger types can be selected).
- E.g., one can change a SMALLINT column to INTEGER.

  However, after such changes a reorganization is recommended.

ALTER TABLE (8)

Adding/Removing Constraints:

- Add a constraint:

```
ALTER TABLE STUDENTS
      ADD CONSTRAINT EXTRA_DEFINED
      CHECK(EXTRA_PT >= 0)
```

  Only table constraints can be added, but column constraints are anyway only syntactic shorthands.

- Remove a named constraint:

```
ALTER TABLE STUDENTS
      DROP CONSTRAINT EXTRA_DEFINED
```

  In SQL-92, one must add the keyword RESTRICT or CASCADE (makes sense only for keys referenced in foreign keys: CASCADE means to remove those foreign keys, too).

ALTER TABLE (9)

- In DB2, the NULL/NOT NULL status of a column can be changed with this syntax:
  - Excluding null values:
    ```
    ALTER TABLE EXERCISES
        ALTER TOPIC SET NOT NULL
    ```
  - Allowing null values:
    ```
    ALTER TABLE EXERCISES
        ALTER TOPIC DROP NOT NULL
    ```
- In DB2, a primary key can be dropped even without name:
  ```
  ALTER TABLE STUDENTS DROP PRIMARY KEY
  ```


Overview

Indexes (1)

- Indexes permit to access rows with a given value for a column without scanning all rows in the table.
- Indexes are created on a specific column or column combination of a specific table:
  ```
  CREATE INDEX X ON T(A)
  ```
- A typical data structure for indexes are B^+ trees.

  The leaf nodes contain all values of the column in sorted order together with pointers to the corresponding rows (RIDs). RIDs are also called ROWIDs or TIDs (tuple identifier).
- Indexes and their use in query evaluation have been discussed in the course "Databases IIB".

Indexes (2)

- As usual, indexes can be declared as unique, meaning that there can be only one RID for every distinct column value:
  ```
  CREATE UNIQUE INDEX X ON T(A)
  ```
- A unique index can be created on a column that permits null values, Then there can be only a single row where the column is null.

  I.e. the null value is treated like any other value. In contrast, primary keys and unique (alternative) keys can be created only on columns that are declared as `NOT NULL`. If the table already contains duplicates, the `CREATE UNIQUE INDEX` statement fails.

Indexes (3)

- Unique indexes are created internally for keys declared in the `CREATE TABLE` statement.

  If one wants to select specific parameters for the index, one must first declare the table without the key, then create the necessary index, and then use `ALTER TABLE` to add the key. It checks whether there is already an index that can be used.

- One can add columns to the index that are not used in the search, e.g. that are not considered in the uniqueness condition:

```
CREATE UNIQUE INDEX X ON T(A) INCLUDE (B)
```

  This permits more queries to be answered directly from the index, without fetching the rows via the RID.

Indexes (4)

- The leaf nodes of a B$^+$-tree index contain all column values in sorted order.

  This can be used for merge joins, and for `ORDER BY` or `GROUP BY`.

- As in the `ORDER BY` statement, one can specify for each column, whether the smallest value is listed first (`ASC`) or last (`DESC`):

```
CREATE INDEX X ON T(A ASC, B DESC)
```

  Normally, the leaf nodes are linked in both directions, so that reverse scans are possible (although this is the default, `ALLOW REVERSE SCANS` is often explicitly specified). If this is not needed, use `DISALLOW REVERSE SCANS` (probably saves one pointer, simplifies query optimization).

Indexes (5)

- Indexes are identified by schema and name (just like tables).

  Different tables cannot have indexes with the same name (in the same schema). A table and its index can be in different schemas.

- Indexes and tables are in different namespaces: an index and a table with the same name are possible.

- Each index entry including all overhead can be at most 25% of the page size.

  An index entry needs the same space as a row with the index columns would need plus typically 11 Bytes (for the RID).

Clustering Indexes (1)

- At most one index per table can be declared as clustered index:

```
CREATE INDEX X ON T(A) CLUSTER
```

- Then DB2 will try to store the rows in the table ordered by the index column(s).

  E.g. rows with the same value for `A` will probably be in the same or in nearby blocks. Also a range scan (given an interval of values for `A`) will be very efficient.

- However, when there is no space in the same or a nearby block, the rows might still be stored far away. The clustered index is only a suggestion/hint.

Clustering Indexes (2)

- Unless the rows are inserted in the sort order of the cluster column(s), the clustering property will degrade over time.

  If the rows are inserted in sort order, one can also put the table into `APPEND` mode.

- Then one can do a table reorganization. This stores the rows again in perfect sort order:

  ```
  REORG TABLE T
  ```

  This command has many parameters, see the manual. Access to the table might be restricted during the reorganization. For large tables, the reorganization might need a lot of time and temporary disk space.

Clustering Indexes (3)

- Before the reorganization, one should specify how much space should remain in each block for future insertions:

  ```
  ALTER TABLE T PCTFREE 25
  ```

- Then during a reorganization or a load operation, at least 25% of each block will remain free.

  Unless already the first row is larger: One row is always inserted in a block without restriction.

- Later insertions or updates can use this space.

  Note that this is different from the parameter `PCTFREE` in Oracle: There it is the space reserve for updates, not used by insertions.

Further Index Parameters (1)

- Also indexes have a parameter `PCTFREE`:

  ```
  CREATE INDEX X ON T(A) PCTFREE 50
  ```

- When the index is created or reorganized, 50% of the space in the leaf blocks will remain free.
- Therefore, insertions are possible for some time without splitting the leaf nodes.
- In this way, the leaf nodes remain in sequential order on the disk (stored in contiguous blocks).

  This improves the performance of range scans or using the index for getting the column values in sorted order.

Further Index Parameters (2)

- In non-leaf nodes, it is not so important to leave free space, since they will anyway not be read sequentially.
- Therefore, normally the minimum of the `PCTFREE` value and 10% is left as free space when the index is built or reorganized.

  I.e. if one uses a small `PCTFREE` value (e.g. `0`), this applies also to the non-leaf nodes. If one uses a large `PCTFREE` value (e.g. `50`), only 10% will remain free in the non-leaf nodes. But see `LEVEL2 PCTFREE` below.

- The default value for `PCTFREE` is 10.

Further Index Parameters (3)

- Later, a parameter `LEVEL2 PCTFREE` was introduced, which applies to the branching nodes just above the leaf nodes (i.e. the second level of the B+-tree).

  If this parameter ist set, all higher levels of the tree get the minimum of 10 and the `LEVEL2 PCTFREE` value percent free space. So in that case, `PCTFREE` applies only to the leaf nodes.

- If values are inserted in increasing order, it is not good to split nodes in the middle, since this will leave all nodes except the rightmost ones only half full. Specify `PAGE SPLIT HIGH` in this case.

  Alternatives: `PAGE SPLIT SYMMETRIC` (the default) and `PAGE SPLIT LOW`.

Further Index Parameters (4)

- The official B+-tree algorithm ensures that all nodes (except possibly the root) are at least half full.

  When entries are deleted, and the used space becomes less than 50%, entries are moved from a sibling node or the nodes are merged.

- This is seldom applied in practice.
- In DB2, one can define e.g. `MINPCTUSED 30`. If a leaf node is filled less than 30%, it will be checked whether it can be merged with a sibling node.

  However, for the new type 2 indexes, entries are only physically removed from indexes when there is an exclusive table lock. Otherwise, the entries are only marked as deleted and no merging is done.

Further Index Parameters (5)

- Example with all discussed parameters:

```
CREATE UNIQUE INDEX X ON T(A ASC, B DESC)
       INCLUDE (C)
       CLUSTER
       PCTFREE 50
       LEVEL2 PCTFREE 0
       MINPCTUSED 25
       PAGE SPLIT SYMMETRIC
       COLLECT DETAILED STATISTICS
```

After the table is filled with data, one should use `RUNSTATS` to collect statistics for the optimizer. If the index is created when the table already contains data, one can use `COLLECT STATISTICS` instead.

MDC Tables (1)

- Multidimensional clustering (MDC) is a new storage structure for tables (introduced in Ver. 8).
- It is primarily intended for data warehouse applications (OLAP), but it may also be useful for classical OLTP applications.
- Whereas the clustering implemented with cluster indexes needs periodic table reorganizations, the clustering in MDC tables is automatic.
- MDC tables apply block indexes which are smaller than standard indexes.

MDC Tables (2)

- A typical example is:

```
CREATE TABLE SALES(
       ID INTEGER NOT NULL PRIMARY KEY,
       PROD_NO INTEGER NOT NULL
           REFERENCES PRODUCTS,
       CUST_NO INTEGER NOT NULL
           REFERENCES CUSTOMERS,
       QUANTITY INTEGER NOT NULL,
       ORD_DATE DATE NOT NULL,
       ORD_MONTH INTEGER NOT NULL
           GENERATED ALWAYS AS
               (INTEGER(ORD_DATE)/100))
       ORGANIZE BY DIMENSIONS(PROD_NO, ORD_MONTH)
```


MDC Tables (3)

- In the example, there are two dimensions:
  - ◇ the product number
  - ◇ month and year of the order

    This is a coarsification of the order date: `INTEGER(ORD_DATE)` yields a number like 20080829, dividing it by 100 gives 200808.
- MDC views tables as a multidimensional cube in which each cell contains rows with the same values for the clustering dimentions.
- If there are n values for `PROD_NO` and m values for `ORD_MONTH`, there are $n * m$ cells. Cells contain table rows. Some of the cells may be empty.

MDC Tables (4)

- Cells are stored in blocks. Blocks are a sequence of contiguous disk pages.

  Actually, blocks are extents. The block size is equal to the extent size defined for the table space in which the MDC table is stored. The minium size of a block/extent is two pages.
- Of course, prefetching should be used so that all rows in a block are read together into main memory.
- Empty cells are stored in 0 blocks, i.e. do not need disk space.
- When the first row of a cell is inserted, a new block is allocated.

MDC Tables (5)

- An MDC table always has a composite block index which maps each unique combination of values for the dimension columns (i.e. each non-empty cell) to one or more blocks.

  A block index is smaller than a normal index, because it contains only block addresses, not RIDs of single tuples.
- So when the next row of the cell is inserted, the block is found via the composite block index and the row is inserted into the same block.
- When the block is full, another block is allocated for the same cell (not necessarily nearby).

MDC Tables (6)

- A slice is the set of all cells that has the same value in one dimension column.
- An MDC table always has dimension block indexes for each dimension. These map a dimension value (i.e. a slice) to a set of blocks.
- The intersection and union of slices are efficiently possible.

  Probably the sets of blocks are stored in ordered sequence.
- One can also create normal RID indexes on MDC tables (for non-dimension columns).

MDC Tables (7)

- Dimensions of MDC tables are typically columns that contain not very many distinct values.

  Compared to the number of table rows ("low cardinality columns").
- For each non-empty cell, at least one block is allocated.
- This makes only sense if there are normally enough rows in a cell to make the block sufficiently full.

  E.g. it would be a severe error to choose a key column as dimension.
- As shown in the example, it might be possible to make the dimension granularity coarser ("rollup").