Transbase® Publishing Guide

Transbase Publishing Guide

Version V8.4

Publication date 2022-11-30
Copyright © 2022 Transaction Software GmbH

ALL RIGHTS RESERVED.

While every precaution has been taken in the preparation of this document, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Table of Contents

Introduction

This manual contains the description of the Transbase Publishing facilities.

They are based upon database files that are platform independent and read-only, so-called *ROM-files*. These files can then be distributed for all platforms on a single read-only medium, e.g. CD, DVD, Blu-ray or from a single download source.

In the following we explain

- the process of creating the *master database* for publication,
- the creation *ROM-files* for distribution of the data on read-only media like CDs, DVDs or Blu-ray disks,
- the commands for creating a *retrieval database* whose data are stored in the ROM-files,
- the *tuning facilities* for these retrieval databases,
- the *update facilities* for publishing differential updates to these bulk data that can easily be transmitted over the internet.

1. Overview of Database Publishing Concepts

1.1. Standard Databases and Retrieval Databases

Transbase supports databases on hard disks as well as databases on read-only media as CD and DVD.

It may be that the license conditions restrict the capabilities of your installed Transbase system to the standard harddisk environment or to the CD/DVD databases.

In the sequel, a database is termed either *Standard Database* or *Retrieval Database* (the latter also *CD/DVD database*).

Standard Databases normally are stored on harddisk only,

A Retrieval Database is a database whose underlying data containers are read only files (also called "romfiles") which may reside on read only media like CD or DVD.

Given a set of romfiles, a Retrieval Database is created via a special CREATE DATABASE command. This is a process which lasts several seconds only.

Romfiles are binary compatible among different machines and thus are suited to be distributed on a variety of platforms.

Note that the term Retrieval Database is used despite the fact that updates on *Retrieval Databases are possible with full SQL functionality*.

1.2. Data Containers of Retrieval Databases

In a Standard Database, all data is stored in read-write files which by default are placed in the subdirectory *disks* and are named *disk#####.###* where # are digits.

A Retrieval Database has one or several romfiles as the containers of permanent data. Their names are *rfileddddd.ddd*. In a Retrieval Database these romfiles may reside on physical CD/DVDs which are mounted read-only to the file system (of course, it is also possible to have the romfiles on physical disk).

For each romfile there exists one corresponding file *compddddd.ddd* in the directory *roms/cd* which contains address information for the compressed page. Compression is described in a separate chapter.

A Retrieval Database also has read-write files in the subdirectory *disks* which however initially are empty. They are used to cache data from romfiles and to store updates. This will be explained in detail below.

1.3. Creation and Distribution of a Retrieval Database

1.3.1. PUBLISH

This chapter describes the development of a database suited for physical distribution on one or several CDs or DVDs.

This process is done in 3 steps:

- **Step (1):** *Creation of a Standard Database* <db>. This is done by the usual CREATE DATABASE .. command. In the sequel, for all commands listed below, it is assumed that the user is inside a tbi session issued by "tbi admin".

  ```
  "CREATE DATABASE masterdb"
  ```

  The CREATE DATABASE command also might contain additional database parameters as is described in the syntax for CREATE DATABASE. The command creates a logically empty database "masterdb", i.e. only the catalogue tables are created. In the sequel, masterdb might be filled with user data via SQL commands.

- **Step (2):** *Preparation of CD Romfiles* This phase is done with the *PUBLISH* command. In its simplest form it looks like:

  ```
  "PUBLISH DATABASE masterdb TO DIRECTORY romfiledir"
  ```

  This command creates romfiles in a directory romfiledir. The romfiles contain the data of "masterdb". The romfiles thus produced then are suited to be physically transferred to read-only media such as CD/DVDs.

- **Step (3):** *Creation of a Retrieval Database* The creation of a database <*dbR*> to work upon the romfiles data is performed with the command

  ```
  CREATE DATABASE dbR FROM PUBLICATION romfiledir"
  ```

  The command creates a database which initially contains all data of "masterdb" The supplied path "romfiledir" must be a directory which contains the romfiles. "romfiledir" also might be a directory on a DVD or CD. This database then is called a Retrieval Database (although this name hides the fact that updates are possible). The process of creating a Retrieval Database is often called "Attaching to romfiles" or similar in the sequel.

Note

The PUBLISH command does not change the state of the specified DB but just produces romfiles in the specified output directory. When a PUBLISH has been run on the DB, one can proceed to update data and generate new data and then run the PUBLISH command again. This enables the correction of semantic data errors or deficiencies which are detected after step (2) has been completed.

1.3.2. Binary Compatibility of Retrieval Databases

Romfiles produced by PUBLISH are binary compatible on all machines supported by Transbase.

1.4. Dataspaces and Romfiles

1.4.1. Separation of Data into Several Romfiles

If no user-defined Dataspace exists in the source database then the PUBLISH command produces a single romfile rfile00000.000 which contains all data.

For each user-defined dataspace a separate romfile is produced. E.g. if two user-defined dataspaces exist then the result of the PUBLISH command will consist of the romfiles rfile00000.000, rfile000001.000 and rfile00002.000. The first romfile contains the database catalogue and all tables (and their indexes and LOB containers) which have been created without specified dataspace.

The romfile with number 00001 contains all tables which have been created in the first user-defined dataspace. The name of the romfile does not contain the explicit name of the dataspace but its number (dspaceno) which can be looked up in the *sysdataspace* system view.

If a dataspace consists of more than one file then several romfiles result with the same dataspace number and a running number in the suffix.

1.5. Caching Data on a Retrieval Database

1.5.1. Shadow Files

When a Retrieval Database has been created by attaching to romfiles, all data resides on the romfiles.

For each attached romfile, another file with a corresponding number and prefix name "shad" (for shadow) is provided in the disks directory. For example, for rfile00003.005, a shadow file shad00003.005 exists,

The shadow files are read-write. They can be used for caching romfile data and also to store updates on the Retrieval DB (as explained later).

For the maintenance of the shadow files as a cache, a LOAD statement and a UNLOAD statement are provided. They serve to transfer data on the basis of tables, secondary indexes and lob indexes to the disk cache and to remove them again.

Note that loading and unloading only influences *performance* of query processing but not availability of data, i.e. all data which initially is on the romfiles can be accessed without any load statements. The load and unload statements are described in detail in chapter *Tuning a Retrieval Application* .

In the sequel, the term *diskcache* is often used in the context of a Retrieval Database.

1.5.2. SQL Updates on a Retrieval Database

All data on a Retrieval Database also can be updated. All SQL update statements are supported, i.e. SPOOL new data, INSERT/UPDATE/DELETE queries, CREATE/DROP/TABLE etc. The romfiles, of course, remain

unchanged and all changes and new data accumulate on the shadow files. New data on the shadow files hides older data on the romfile.

In summary, the purpose of the shadow files on a Retrieval Database is twofold: it servers as a data cache for performance tuning (load/unload statements) and it serves as an update accumulation container.

1.5.3. The View loadinfo

The set of LOADed and changed pages (thus residing in the shadowfiles) can be observed via the catalog view *loadinfo*.

Field	**Type**
segment	INTEGER
rompage	INTEGER
diskpage	INTEGER
flag	INTEGER

Primary key: (segment, rompage)

- *segment:* Contains the segment number of the page. It identifies the table, index or LOB container the page belongs to.
- *rompage:* Contains the page number of the page in the ROM address space.
- *diskpage:* Contains the page number of the page in the diskfile address space.
- *flag:* Contains the value 1 if the page has just been cached via a LOAD STATEMENT and the value 2 if the page has been changed due to a INSERT or UPDATE or DELETE STATEMENT .

The view loadinfo contains one row for each database page which resides on one of the shadow files.

For example, the number of pages on the diskcache can be evaluated by :

```
SELECT COUNT(*) FROM loadinfo
```

1.5.4. Update Behaviour of Retrieval Databases in Multi User Mode

It has to be noted that a Retrieval DB exhibits limited update concurrency in multi user mode. Beside the unavoidable normal lock behaviour which can lead to contention on tables (rejected queries or transaction deadlocks) additional contention on the virtual table *loadinfo* may occur.

This table is in effect a data structure which holds the information which pages have been changed (or have been loaded) onto the shadow files.

Update conflicts on loadinfo can lead to transaction abort.

If one needs absolute certainty that an update transaction does not fail, one must assure that only one transaction drives updates at a time.

Note in this context that the Transbase tool *tbdiff* is a useful tool to automate the process of transferring updates to Retrieval databases. tbdiff reads two databases and generates a SQL script (and spoolfiles) which transforms

one database to the other. In this way, tbdiff can be used to generate a script which incorporates data changes on a Retrieval DB and the script can then be distributed and applied on the corresponding Retrieval databases.

Another possibility to distribute updates is the concept of delta romfiles (see *Database_Deltas*).

1.6. Compression of Romfiles

All romfiles are automatically compressed.

Romfile compression is completely transparent to the applications. Of course, compression is also invisible to all application programs.

Romfile compression is made on the basis of single pages. In the attached database, each page accessed in the romfile is decompressed before it is worked upon. For the sake of very fast access, pages in the shadow files are all stored non-compressed.

To access a page in a compressed romfile its offset must be looked up in a corresponding compression information file (compfile00000.000, etc.). After fetching the page into main memory, it is decompressed if necessary (not all pages really are compressed in a compressed romfile, only those where the space saving reached a threshold value).

The compression rate for Transbase tables depends on their structure. For a random mix of tables, a space reduction in the order of 40% can be expected. This means that a CD of 600 MB can store 1 GB Transbase tables.

For Binary Large Objects (LOBs), the compression aspects are a bit more complicated. On the one hand, there are LOBs which compress excellently (e.g. ASCII text). On the other hand, for certain LOB types, compression should be done by the application because it can process the whole LOB instead of single pages as Transbase does and can take into account the internal structure of the LOB (which is unknown to Transbase). If an application stores well compressed LOBs, Transbase normally achieves no further reduction. In this case, the LOBs are stored as delivered by the application and thus also are redelivered with no decompression overhead.

1.6.1. Performance Impacts of Compression

The compression facility slows down the PUBLISH command which however is not done frequently. On the Retrieval DB, each page read from the romfile must be decompressed which however is rather fast. Performance can still be enhanced by using the LOAD command in a suitable manner (see special chapter). This command caches parts of the DB onto the shadow files (uncompressed) where they can be accessed without the decompression overhead.

1.7. Distribution of Database Updates with Delta Romfiles

The issue of this section is, to describe a mechanism to redistribute an updated version of a formerly distributed DB to the client.

Assume, that the initially distributed version of the database is the version V0. The database is updated on the server site via SQL commands finally leading to a version V1. Instead of redistribution of a complete new romfile

set, there is a mechanism to redistribute some smaller files to the client which contain the updates from V0 to V1 and which are considerably smaller.

At server site, the database in version V1 may be continously updated leading to newer versions V2,...,Vn. Each version Vi may be propagated to clients by submitting a set of files which are as small as possible.

To this end, two additional filetypes are used. So called "gendelta files" hold updates of the DB on a after image page basis. A variant which minimizes transport volume uses an XOR technique on pages and is used in so called "tfiles".

1.7.1. Working with Gendelta Files

Recall, that the proposed directory "romfiledir" holds the romfiles of an initial version V0 of the master database.

For newer versions V1, V2 etc. of the master database, subdirectories "delta00001", "delta00002" etc. are used which contain the corresponding updates of the newer versions w.r.t. the older versions.

In each subdirectory, a set of files with names gendelta00000.000, gendelta00001.000, etc. will be placed.

A gendelta file with name *gendelta*<*x*>.<*y*> shadows or adds data contained in romfile with name *rfile*<*x*>.<*y*> where x and y are 5-digit and 3-digit numbers, resp. (starting with 0).

1.7.1.1. Producing Gendelta Files

Initially, an original romfile set R0 inside a father directory "romfiledir" (which represents the database "DBE" in a first version V0) has been produced (as already described).

Assume that on client site also a directory "romfiledir" exists, and the client's DB has used this directory for creating its Retrieval DB.

For producing gendelta files, a retrieval database in version V0 is also made at server site:

```
"CREATE DATABASE DBV0 FROM PUBLICATION  romfiledir"
```

The command creates a (new) retrieval database DBV0 by attaching to the produced romfiles. This is exactly the same mechanism as at client site.

DBV0 is the starting point for the development of the new database version V1.

The development is done via arbitrary SQL commands possibly including DDL commands.

When the development of the DB version V1 is finished, gendelta files can be produced using the command

"PUBLISH DATABASE DBV0 INCREMENTAL TO romfiledir"

This command uses the existing directory "romfiledir" which contains the original romfiles. Inside "romfiledir" a new directory "delta000001" is produced, inside delta00001 a set D1 of gendelta files is stored.

For each attached romfile, a corresponding gendelta file is produced.

Now "delta000001" can be shipped to the client's database DB which - up to now - works on the original romfiles only. The installation procedure now must link the directory "delta000001" under the original main directory romfiledir as on server site.

The existing RETRIEVAL database (for example also named DBV0) at client site may now be deleted. A new database DBV1 now can be created by using "romfiledir" again exactly as in the first CRESTE DATABASE command.

"CREATE DATABASE DBV1 FROM PUBLICATION romfiledir".

The former database DBV0 would remain operational (and unchanged) if this is desired. Alternatively, DBV0 could have been deleted before via

"DROP DATABASE DBV0".

and the new DB could be created with the same name as the old.

1.7.1.2. Iteration of DB Updates with Gendelta Files

It is now described how a further update round is made.

At server site, first the same procedure is made as at client site:

"CREATE DATABASE DBV1 FROM PUBLICATION romfiledir".

DBV1 contains all changes of the first update. The property "publish_revision" in the catalog table "sysdatabase" has the numerical value 1, because one delta subdirectory has been attached.

Further updates now can be applied to DBV1 resulting in a newer version V2 of the database,

Then a new set of gendelta files can be produced by a further invocation of:

"PUBLISH DATABASE DBV1 INCREMENTAL TO romfiledir"

Note that the same output directory as for the first update must be used. A further subdirectory delta00002 then is added which again contains file gendelta00000 and so on.

A new DB of the generation 2 can then be made. This is again performed by the call:

"CREATE DATABASE DBV2 FROM PUBLICATION romfiledir".

Of course, at client site, the structure of the current deltadir must be installed by linking the new delta00002 directory under its "romfiledir".

1.7.2. Working with Tfiles

Tfiles are an alternative representation of gendelta files. They are based on an XOR representation of the page updates. In many cases the size of tfiles is smaller than the size of gendelta files.

At server site, tfiles are produced with

"PUBLISH DATABASE DB INCREMENTAL XOR TO romfiledir"

Note the usage of the XOR clause placed behind INCREMENTAL.

Like with the gendelta variant, a subdirectory *delta<x>* is generated under romfiledir. For the first update generation, it would be delta00001.

delta00001 contains files named tfile0000.000 and so on.

The delta00001 directory now is shipped to the client.

The next 2 steps are identical to the gendelta variant:

The old DB can be dropped: *"DROP DATABASE DBV0"*.

The new DB can be attached: *"CREATE DATABASE DBV1 FROM PUBLICATION romfiledir"*.

2. Development on an Independent Database

2.1. Building the Initial Standard Database

The performance of a Retrieval Database is highly influenced by the degree of fragmentation in the address space of the database. For example, LOB objects should lie adjacently in the address space. Local access to a table should also physically behave locally.

The address space becomes scattered if records of different tables are deleted and inserted randomly. In contrast, if one table is spooled in completely before the next table is spooled, the address space is normally best suited for good performance.

So it is recommended to develop the first version of the database on a separate standard database "developdb". Instead of PUBLISHing developdb, a new database masterdb should be built up, e.g. with the tool "tbarc" .

tbarc -w archive developdb

tbarc -r archive masterdb

The building process guarantees a perfect address space organization on masterdb. The same then holds for the produced romfiles.

2.2. Minimizing Page Updates in Gendelta Files

As described in the section about *database deltas* , the database DBVi at server site may be continuously maintained leading to new versions which are to be shipped to clients via gendelta files. It is of high interest to keep their sizes as small as possible.

To this end, changes on the retrieval database at server site should be planned carefully. The sizes of gendelta files depends on the number of changed pages in the B-trees of tables and secondary indexes.

As a drastic example, assume that 1000 records of a table T are to be changed on a field F. An integrated UPDATE statement would change the corresponding pages in the B-tree and thus would be a reasonable way to keep page changes minimal. In contrast, a DELETE of the 1000 records followed by an INSERT statement of the 1000 changed records would probably first cause many B-tree merges followed by split operations. In addition, all secondary indexes would also be touched with DELETE and INSERT operations even if they do not contain field F. All these page changes would cause new page versions on the shadow files. The pages of the secondary indexes not containing field F would logically have the same contents as before but not necessarily be physically identical. So these pages would increase the transport volume considerably.

Whenever a table T on the Retrieval DB is to be changed in a nontrivial manner, a special variant of the SQL command UPDATE may be useful to keep page changes minimal. This is described in the next section.

2.2.1. The "UPDATE FROM" statement

This statement is a variant of the standard SQL UPDATE statement.

UPDATE T FROM <source>

<source> may be a table name in a local or remote DB, or an arbitrary SELECT block referencing tables in a distributed Transbase D query.

Instead of performing changes on a table T directly, one could work arbitrarily on a copy of T (table "Tcopy", perhaps in another database "dbdevelop"). Finally the following statement is performed:

UPDATE T FROM Tcopy@dbdevelop

The effect of this statement is that T is updated to contain the same data set as the source *Tcopy@dbdevelop*. Logically it is an assignment operation. It is performed in a way that keeps page changes minimal:

Source and target are merged in key order and compared on a data based level. Matching data records do not cause an update, equal keys with not matching nonkey fields cause a record update in T, records missing on T side are inserted, records in T not contained in the source are deleted in T.

Note that there are is a semantic difference to the standard SQL UPDATE statement because referential constraints as well as triggers are ignored, i.e. the statement is designed to work on a table basis. However, secondary indexes of course are maintained.

2.3. Transferring Database Updates with tbdiff

If the database is developped on an independent database dbdwvelop, then the tool `tbdiff` may be used to trace and apply the updates to the current version of the retrieval database DBVi at server site.

tbdiff is applied to DBVi and MASTERDB. For each table in both databases, tbdiff checks for existence of the corresponding counterpart, identifies the difference in their data sets, and finally produces SQL statements and spoolfiles which are suited to update DBVi to the state of MASTERDB.

There are two variants of tbdiff, namely an online variant and an offline variant. The online variant directly performs updates on DBVi and thus produces the state of MASTERDB. The offline variant produces spoolfiles and a script S. The script S then can be applied using tbi on the DBVi and also produces the same target state.

Both variants are described below.

2.3.1. Tbdiff in Online Variant

The online variant of tbdiff has the following syntax:

tbdiff [options1] [-i] -o[a] db1 db2 [options2]

The option -o indicates the online variant of tbdiff. The option -i outputs some statistical information about the changes performed on db1. The option -oa triggers a rollback on db1 after the updates have been performed and

thus leaves db1 unchanged. In combination with the -i option, it gives an overview about the changes necessary to push db1 to the state of db2 (without performing any changes).

The online variant of tbdiff is optimal with respect to the size of gendelta files.

2.3.2. Tbdiff in Offline Variant

The offline variant of tbdiff has the following syntax:

tbdiff [options1] [-i] scriptdir [-uf] db1 db2 [options2]

The option -i is as in the online variant.

If the directory scriptdir does not exist it is created otherwise it must be empty.

tbdiff writes a script diff.sql into scriptdir which consists of SQL commands and DDL commands using spoolfiles which are also placed in the target directory.

If the script diff.sql is applied onto db1 with the tool tbi, it produces the state of db2 on db1. It is, however, possible to modify the script before it is applied.

With the -uf option, tbdiff produces a script which uses the UPDATE FROM statement. This may decrease the size of gendelta files but can only be used with Transbase versions implementating this statement.

3. Tuning the CD/DVD Application - Load and Unload

After creation of a Retrieval Database ("Attach"), initially all data is read from the romfiles residing on read-only media.

To speed up access to often used data, the shadow files can be used as a data cache.

This can be achieved by the statement LOAD and UNLOAD. They can either be run interactively with programs like TBI or can be run in application programs against the programming interface.

LOAD and UNLOAD are subject to the transaction concept, thus they have to be committed. If the enclosing transaction is aborted, all LOAD and UNLOAD effects of that transaction are made undone.

Note

Loading and unloading data is a pure performance tuning feature. They have no effects on the results of SQL queries.

The set of actually loaded pages can be observed by the catalog view *loadinfo*.

3.1. The Load Table Statement and Load Index Statement

The Load Table Statement and Load Index Statement serve to load data from a specified table or index from romfile or delta romfile to the shadow files.

Loadable objects are: some or all B-tree levels of a specified table or secondary index, the IK-part of a table, and the index part of a LOB container.

Syntax:

```
Load_Default_Statement ::=
        LOAD [ DISK CACHE ] DEFAULT

Load_Statement ::=
        LOAD [ DISK CACHE ] Load_spec
Load_spec ::=
        TABLE Table_name [ Tload_spec ]
    |     INDEX Index_name [ Xload_spec ]
    |     INDEX FULLTEXT Special_Ftxt_Table
                [OF]  Fulltext_Index_name [ Xload_spec ]
Tload_spec ::=
        Level_spec
    |     IKACCESS
    |     LOBACCESS
Xload_spec ::=
        Level_spec
Level_spec ::=
        LEVELS Levels
Levels ::=
        ALL
    |     IntegerLiteral
    |     ALL - IntegerLiteral
Special_Ftxt_Table ::=
```

```
      WORDINDEX
|      WORDLIST
|      CHARMAP
```

Explanation:

The *Load_Default_Statement* is a useful combination of a series of Load Statements for all existing tables and indexes. It is described in the next Chapter.

In *LOAD TABLE T [TloadSpec]*, an unspecified *Tloadspec* means loading all B-tree levels of table T, its IK-part (if any) and also its LOBACCESS index if the table has LOB columns, but not its secondary indexes..

Note that loading the IK-part (if any) is only useful if queries are evaluated via a secondary index which does not include all needed attributes such that the result records must additionally be materialized using the IK value of the records.

Analogously, in *LOAD INDEX I [XloadSpec]*, an unspecified *Xload_spec* means loading all B-tree levels of index I.

A fulltext index has some auxiliary tables which are explained in the TB/SQL Reference Manual. Some of them can also be loaded. A reasonable strategy is explained in the *LoadDefaultStatement*. With the specification of *Tloadspec and Xloadspec*, resp., one has finer control over the loaded parts of a table or an index. With the *LEVELS* clause it is possible to restrict the loading to the upper levels of the B-tree of a table or index. The level numbering starts at 1 with the root.

- Specification *LEVELS n (n >=1)* loads the n upper levels of the tree.
- Specification *ALL* loads all levels.
- Specification *ALL-n* loads all levels except the n lowest levels.

Specifically, the specification *ALL-1* loads all levels except the leaf level (which is by far the largest) and mostly is the most reasonable variant.

If *IKACCESS* is specified in the *Tload_spec* then only the IK-part of the table is loaded.

If *LOBACCESS* is specified in the *Tload_spec* then the index part of the LOB container of the table is loaded (not the LOB objects themselves!). If the table has no LOB attribute then an error occurs.

Loading parts which are already loaded is not an error but has no effect. For example, to load additionally level 3 when 2 levels are already loaded, one need not unload and reload, but can directly load with LEVEL 3.

Note

For large tables, a good compromise between disk space and performance gain is to specify:

```
LOAD  TABLE  T  LEVELS  ALL-1
```

The disk space needed is in most cases about 1%-2% of the table which is very space economical. [SK1]access.

For system tables (needed for query compilation) it is convenient to specify:

```
LOAD  TABLE  T
```

If only stored queries are used, then this is not necessary.

For tables with LOB fields, *LOADing LOBACCESS* is recommended.

- **LOAD TABLE suppliers.** loads all tree levels and the IK-part of the table suppliers

- **LOAD TABLE suppliers LEVELS ALL-1 .** loads all levels except the leaf level of the table suppliers
- **LOAD INDEX sindex LEVELS ALL .** loads the secondary index sindex
- **LOAD TABLE blobtable LOBACCESS .** loads the indexpart of the LOB container of blobtable provided the table has at least one LOB attribute.

Locks:

The *LoadStatement* sets an R-Lock on the underlying table. Concurrent read and write transactions may access the data and also already benefit from the loaded data.

3.2. The Load Default Statement

The Load Default Statement is a useful combination of a series of *LoadStatements* for all existing tables and indexes. For many Retrieval Databases, it offers a good default strategy.

Syntax:

(repeated)

```
Load_Default_Statement ::=
        LOAD [ DISK CACHE ] DEFAULT
```

Explanation:

The *LoadDefaultStatement* implicitly performs the following statements:

```
    For all catalog tables S (systable, syscolumn, etc.):
        LOAD  TABLE  <S>  LEVELS  ALL
    For all user defined tables T :
        LOAD  TABLE  <T>  LEVELS  ALL - 1
    For all secondary indexes I :
        LOAD  INDEX  <I>  LEVELS  ALL - 1
    For all fulltext secondary indexes FI :
        LOAD  INDEX  FULLTEXT WORDLIST <FI>  LEVELS
            ALL - 1
        LOAD  INDEX  FULLTEXT WORDINDEX <FI>  LEVELS
            ALL - 1
        LOAD  INDEX  FULLTEXT CHARMAP <FI>
    For all user defined tables T which have LOB fields:
        LOAD  TABLE  <T>  LOBACCESS
```

In summary, the *LoadDefaultStatement* is very economic in disk space - the disk space needed is in most cases a few percent of the Retrieval Database size.

Of course, the desired load profile can be achieved by a combination of the *LoadDefaultStatement* with other Load and/or *UnloadStatements*.

A load script could contain the following statements:

```
LOAD  DEFAULT
LOAD  TABLE  T31
LOAD  TABLE  T32  IKACCESS
```

Note that *LOAD DEFAULT* does not load the *IKACCESS* part of user tables (because the space needed can be considerable for very large tables: 4 byte per record).

3.3. The Selective Load Statement

The Selective Load Statement loads data which is specified by a *SELECT* query. The *SELECT* query is internally evaluated (no data is delivered to the user) and all pages needed to evaluate the query are loaded. The loading of B-tree leaf pages can be suppressed.

Syntax:

```
Load_Select_Statement ::=
    LOAD [ DISK CACHE ] [ Leaf_Spec ] Select_Statement
Leaf_spec ::=
     { WITH | WITHOUT } LEAF
```

Explanation:

The specified *SelectStatement* is internally evaluated. All pages which are read during the evaluation of the query are loaded onto the disk cache (unless they are already loaded).

Pages needed for evaluation may be B-tree pages or IK pages. B-tree pages come from primary B-trees or from B-trees for secondary indexes. IK pages are needed whenever a secondary index is used for evaluation of a search condition but the secondary index does not include all needed record fields such that the record additionally must be materialized via the internal key (IK).

If no *Leafspec* is specified then the effect is as if a *WITH LEAF* had been specified.

If a *WITHOUT LEAF* has been specified then all pages needed for evaluation except the leaf pages of B-trees are loaded.

Note that LOB objects are not materialized by a *SelectStatement*. This means that no LOB pages can be loaded by the *LoadSelectStatement* (but see the Load Switch Statement).

Let 'name' be the key of the table 'suppliers'; then a search condition name = 'xyz' is evaluated by a B-tree traversal:

```
LOAD SELECT suppno FROM suppliers
WHERE name = 'xyz'
    Loads the B-tree path from the root to the data page
    (including the data leaf page) where supplier
    record 'xyz' resides.

LOAD WITHOUT LEAF SELECT suppno FROM suppliers
WHERE name = 'xyz'
    Loads the B-tree path from the root to the data
    page (not including the data leaf page) where
    supplier record 'xyz' resides.

LOAD SELECT * FROM suppliers
    Loads all B-tree leaf pages.
```

Locks:

The Selective *LoadStatement* sets R-Locks via the specified *SelectStatement*.

3.4. The Switch Load Statements

With the Switch Load Statements an internal load descriptor can be turned *ON* and *OFF*. The load descriptor is transaction specific. Whenever a page is read for the evaluation of a query on behalf of a transaction (TA), the transaction specific load descriptor is inspected. If its state is *ON*, the page is loaded. At begin of the TA, the descriptor state is *OFF*. With this technique, page loading is done as a side effect of query processing.

Syntax:

```
Switch_Load_Statement ::=
        LOAD [ DISK CACHE ] [ Switch_Spec ] SET ON
    |     LOAD [ DISK CACHE ] SET OFF
Switch_spec ::=
        [ Leaf_Spec ] [ Lob_Spec ]
    |     [ Lob_Spec ] [ Leaf_Spec ]
Leaf_spec ::=
        { WITH | WITHOUT } LEAF
Lob_spec ::=
        { WITH | WITHOUT } LOB
```

Explanation:

If *SET OFF* is specified, then the load descriptor is set to the state *OFF*. This is also the initial state at begin of a TA. In this state no page loading is done during query processing.

If *SET ON* is specified, then the load descriptor is set to the state *ON*. Depending on the *LeafSpec* and the *LobSpec* there are 4 different *ON* states. A missing *LeafSpec* is equivalent to *WITH LEAF*. A missing *LobSpec* is equivalent to *WITHOUT LOB*. Whenever a page is read and the load descriptor has been *SET ON*, then the following is performed: If the page is a B-tree leaf page and *WITH LEAF* has been specified then the page is loaded. If the page is a LOB page and *WITH LOB* has been specified then the page is loaded. If the page is neither a B-tree leaf page nor a LOB page, then it is loaded. In all other cases the page is not loaded.

Note

LOB page reading (and loading) may only occur as side effect of a *GETBLOB* or *GETCLOB* call.

```
LOAD SET ON
    Sets load descriptor ON, defaults are WITH LEAF
    and WITHOUT LOB.

LOAD WITH LOB SET ON
    Sets load descriptor ON, leaf is default ON,
    LOB is explicitly ON.

LOAD SET OFF
    Sets load descriptor OFF.
```

3.5. The Unload Statement

The Unload Statement is the inverse statement of the Load Statement. The same objects as with the Load Statement can be specified. In addition, in its simplest form all loaded objects are unloaded from the disk cache.

Syntax:

```
Unload_Statement ::=
        UNLOAD [ DISK CACHE ] Unload_spec
Unload_spec ::=
        ALL
```

```
    |     TABLE Table_name
    |     INDEX Index_name
```

Explanation:

If the *Unloadspec* is *ALL*, then everything that has been loaded is unloaded, i.e. the disk cache is freed from all loaded objects.

If *TABLE Tablename* is specified then all loaded pages which belong to the specified table (B-tree pages, IK pages, LOB pages) are removed from the disk cache. It makes no difference whether a page has been loaded via the Load Table Statement or the Selective Load Statement or as side effect after a Switch Load Statement.

If *INDEX Indexname* is specified then all loaded pages which belong to the specified index (B-tree pages) are removed from the disk cache. It makes no difference whether a page has been loaded via the Load Table Statement or the Selective Load Statement or as side effect after a Switch Load Statement.

```
UNLOAD ALL
    unloads all loaded objects from the disk cache
UNLOAD TABLE suppliers
    unloads all loaded pages of table suppliers.
UNLOAD INDEX sindex
    unloads the secondary index sindex from
    the disk cache
```

Locks:

The *UnloadStatement* sets an R-Lock on the underlying table(s).

3.6. Disk Caching and Main Memory Caching

Beside *disk caching* on Retrieval Databases, Transbase also employs *main memory caching*. To give a better overview about the relationship and differences, in the following the two mechanisms are compared to each other.

Transbase *automatically* caches often used data in its main memory (shared memory in multi user databases). The size of the main memory cache can be configured at database creation time (or *attach time* in case of CD/DVD) and later on with the *tbadmin -a* ..command. This kind of caching is done for each database type. The data is replaced based on a LRU (least recently used) algorithm. An exception are LOB data pages which are not cached but tossed immediately, since LOBs typically are large and would very quickly replace often used data pages.

Disk caching for Retrieval Databases is *not done automatically*. It is explicitly controlled by the described load and unload statements.

4. A Sample Development Process

In the following, a sample scenario is described for distribution of gendelta updates for a retrieval database

```
// Server Site
// Production of romfile set RS and distribution to client

// Server Site and Client Site
CREATE DATABASE DBR0 FROM PUBLICATION <romfiledir>

// +++ Begin Update iteration1:
// Server Site
// SQL updates performed on DBR0, leading to a DB state D1
PUBLISH DATABASE DBR0 INCREMENTAL TO   <romfiledir>
// creates subdirectory delta00001
// Distribution of fileset delta000001 of <romfiledir> to client
// Client Site
// Linking delta00001 under <romfilepath> of client
// Server Site and Client Site
DROP DATABASE DBR0
CREATE DATABASE DBR1 FROM PUBLICATION <romfiledir>
// +++ End Update iteration1:

// +++ Begin Update iteration2: this time tfiles are used instead of gendeltas
// Server Site
// SQL updates performed on DBR1, leading to a DB state D2
PUBLISH DATABASE DBR1 INCREMENTAL XOR TO   <romfiledir>
// creates subdirectory delta00002
// Distribution of fileset delta000002 of <romfiledir> to client
// Client Site
// Linking delta00002 under <romfilepath> of client
// Server Site and Client Site
DROP DATABASE DBR1
CREATE DATABASE DBR2 FROM PUBLICATION <romfiledir>
// +++ End Update iteration2:
```