SP-GiST for PostgreSQL
User Manual
Version1.0

Disclaimer

While every effort has been made to make this manual as complete and as accurate as possible, no warranty or fitness is implied. The information provided here is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this manual.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. *IIS-0093116.* Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

What is SP-GiST?
SP-GiST is a General Index Framework for Space Partitioning Trees. Check http://www.cs.purdue.edu/spgist for details.

What is PostgreSQL ?
PostgreSQL is a highly extensible open source object-relational database management system. Check http://www.postgresql.org for details.

What is SP-GiST for PostgreSQL ?
SP-GiST for PostgreSQL is an effort to introduce SP-GiST index access method to PostgreSQL.

TABLE OF CONTENTS

INSTALLATION

System Requirements and Prerequisites

SP-GiST for PostgreSQL sources are expected to build successfully on any Unix, or Unix-like system, although it was only tested on Linux-x86 , Linux-amd64 , and SunOS 5.8. PostgreSQL8.0 or later is required. Earlier versions of PostgreSQL are not supported.
Please note that building SP-GiST for PostgreSQL requires PostgreSQL server-side header files to be installed.

Download the source code

SP-GiST for PostgreSQL sources can be downloaded at:
http://www.cs.purdue.edu/spgist/download.html
Since SP-GiST is still under development, it is a good idea to periodically check for new releases of sources.

Building and Installing SP-GiST for PostgreSQL

Quick Installation Instructions

For the impatient, the following Unix commands untars, configure, build, and install SP-GiST for PostgreSQL. (replace xxxx with your distribution version)

```
gzip -cd spgist-xxxx.tgz | tar xvf -
cd spgist-xxxx
./configure --with-postgresql=/path/to/postgresql/installation
make
su
make install
```

If you are using SP-GiST in a production environment (do you really dare?) you may need to use *make install-strip* instead of *make install.*
For detailed description of the installation process please check the next section “Detailed Installation Instructions”.

Detailed Installation Instructions

You may skip this section if you don't want to care about building and installation details or if you are already familiar with the GNU tools.

The `configure' shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a `Makefile' in each directory of the package. It may also create one or more `.h' files containing system-dependent definitions. Finally, it creates a shell script `config.status' that you can run in the future to recreate the current configuration, a file `config.cache' that saves the results of its tests to speed up reconfiguring, and a file `config.log' containing compiler output (useful mainly for debugging `configure').

If you need to do unusual things to compile the package, please try to figure out how `configure' could check whether to do them, and mail diffs or instructions to the address given in the

`README' file, so they can be considered for the next release. If at some point `config.cache' contains results you don't want to keep, you may remove or edit it.

The file `configure.in' is used to create `configure' by a program called `autoconf'. You only need `configure.in' if you want to change it or regenerate `configure' using a newer version of `autoconf'.

The simplest way to compile this package is:

1. `cd' to the directory containing the package's source code and type `./configure' to configure the package for your system. If you're using `csh' on an old version of System V, you might need to type `sh ./configure' instead to prevent `csh' from trying to execute `configure' itself.

Running `configure' takes awhile. While running, it prints some messages telling which features it is checking for.

2. Type `make' to compile the package.

3. Optionally, type `make check' to run any self-tests that come with the package.

4. Type `make install' to install the programs and any data files and documentation.

5. You can remove the program binaries and object files from the source code directory by typing `make clean'. To also remove the files that `configure' created (so you can compile the package for a different kind of computer), type `make distclean'. There is also a `make maintainer-clean' target, but that is intended mainly for the package's developers. If you use it, you may have to get all sorts of other programs in order to regenerate files that came with the distribution.

Some systems require unusual options for compilation or linking that the `configure' script does not know about. You can give `configure' initial values for variables by setting them in the environment. Using a Bourne-compatible shell, you can do that on the command line like this:

CC=c89 CFLAGS=-O2 LIBS=-lposix ./configure

Or on systems that have the `env' program, you can do it like this:

env CPPFLAGS=-I/usr/local/include LDFLAGS=-s ./configure

You can compile the package for more than one kind of computers at the same time, by placing the object files for each architecture in their own directory. To do this, you must use a version of `make' that supports the `VPATH' variable, such as GNU `make'. `cd' to the directory where you want the object files and executables to go and run the `configure' script. `configure' automatically checks for the source code in the directory that `configure' is in and in `..'.

If you have to use a `make' that does not support the `VPATH' variable, you have to compile the package for one architecture at a time in the source code directory. After you have installed the package for one architecture, use `make distclean' before reconfiguring for another architecture.

By default, `make install' will install the modules in `/usr/local/postgresql/lib'. You can modify that path according to your PostgreSQL installation by giving `configure' the option `*--with-*

postgresql=PATH'. Or you can update manually file 'configure.in' and replace path '/usr/local/' with your PostgreSQL path.

You can specify separate installation prefixes for architecture-specific files and architecture-independent files. If you give `configure' the option `--exec-prefix=PATH', the package will use PATH as the prefix for installing programs and libraries. Documentation and other data files will still use the regular prefix.

In addition, if you use an unusual directory layout you can give options like `--bindir=PATH' to specify different values for particular kinds of files. Run `configure --help' for a list of the directories you can set and what kinds of files go in them.

If the package supports it, you can cause programs to be installed with an extra prefix or suffix on their names by giving `configure' the option `--program-prefix=PREFIX' or `--program-suffix=SUFFIX'.

Optional Features
==========
Some packages pay attention to `--enable-FEATURE' options to `configure', where FEATURE indicates an optional part of the package. They may also pay attention to `--with-PACKAGE' options, where PACKAGE is something like `gnu-as' or `x' (for the X Window System). The `README' should mention any `--enable-' and `--with-' options that the package recognizes.
For packages that use the X Window System, `configure' can usually find the X include and library files automatically, but if it doesn't, you can use the `configure' options `--x-includes=DIR' and `--x-libraries=DIR' to specify their locations.

Specifying the System Type
==========
There may be some features `configure' can not figure out automatically, but needs to determine by the type of host the package will run on. Usually `configure' can figure that out, but if it prints a message saying it can not guess the host type, give it the `--host=TYPE' option. TYPE can either be a short name for the system type, such as `sun4', or a canonical name with three fields:

CPU-COMPANY-SYSTEM

See the file `config.sub' for the possible values of each field. If `config.sub' isn't included in this package, then this package doesn't need to know the host type. If you are building compiler tools for cross-compiling, you can also use the `--target=TYPE' option to select the type of system they will produce code for and the `--build=TYPE' option to select the type of system on which you are compiling the package.

Sharing Defaults
==========
If you want to set default values for `configure' scripts to share, you can create a site shell script called `config.site' that gives default values for variables like `CC', `cache_file', and `prefix'. `configure' looks for `PREFIX/share/config.site' if it exists, then `PREFIX/etc/config.site' if it exists. Or, you can set the `CONFIG_SITE' environment variable to the location of the site script. A warning: not all `configure' scripts look for a site script.

Operation Controls
==========
`configure' recognizes the following options to control how it operates.

`--cache-file=FILE'
Use and save the results of the tests in FILE instead of `./config.cache'. Set FILE to `/dev/null' to disable caching, for debugging `configure'.

`--help'
Print a summary of the options to `configure', and exit.

`--quiet'
`--silent'
`-q'
Do not print messages saying which checks are being made. To suppress all normal output, redirect it to `/dev/null' (any error messages will still be shown).

`--srcdir=DIR'
Look for the package's source code in directory DIR. Usually `configure' can determine that directory automatically.

`--version'
Print the version of Autoconf used to generate the `configure' script, and exit. `configure' also accepts some other, not widely useful, options.

GETTING STARTED

Loading SP-GiST core and extensions modules

The SP-GiST core module and extension modules are installed by default under *$pgdir/lib* , where *$pgdir* is the PostgreSQL installation prefix. SP-GiST for PostgreSQL distribution is shipped with some support scripts to simplify the process of loading and configuring the modules.

To load SP-GiST core module all you need to is to invoke the following command:

psql < spgist-load.sql.

The spgist-load.sql script loads the module, and defines the spgist access method to the PostgreSQL engine.

Loading an SP-GiST extension module is done similarly by invoking the load script located in the extension directory of the distribution. For example, to load the spgist-trie extension, make sure that spgist core is loaded first then invoke the following commands:

cd spgist-trie
psql < spgist-trie-load.sql

Creating indexes using SP-GiST access method

Before you can create or use an SP-GiST access method, the SP-GiST core module should be loaded and at least one spgist extension is used to build an operator class that supports the data type being indexed. Refer to the extension load script for details about the operators and operator classes defined by the extension.

The create statement syntax is:

CREATE INDEX <index name> ON <relation name> USING spgist (<key column> <operator class>);

Example:

CREATE INDEX example_idx ON example_tbl USING spgist(str spgist_trieword_ops);

WRITING SP-GIST EXTENSIONS

SP-GiST extension is a set of functions that realize certain index structure and define the exact behavior of the index. We provide four extensions with the release of version 1 which are, spgist-kd, spgist-pquad, spgist-pmrquad, and spgist-trie which release indexes, kd-tree, point quadtree, PMR quadtree, and the trie, respectively.

The main functions of an SP-GiST extension are:

- **Consistent()**
- **NN_Consistent()**
- **Penalty()**
- **Split()**

These four functions are written by the developer to specify how the index works. Each of the functions has a set of interface parameters through which the SP-GiST internal functions (SP-GiST core) and the external functions communicate and exchange data. In the next subsection, we describe in more details the role of each function.

SP-GiST Extension Functions

• Consistent()

Consistent() function is called from the internal function *Search()* to guide the searching in the index tree. The input parameters to Consistent() are:

1. ***spgist_query *q:*** q is a pointer to the query information. The query information contains ***strategy*** element which specifies the passed operator type, and ***key*** element which specifies the search key passed through the query.

2. ***predicate *entry_pred:*** entry_pred is a pointer to the currently passed node entry (corresponds to a partition in the node). If the current node is a leaf node, then entry_pred points to a data key. Otherwise, entry_pred is a predicate for the subtree descendant from this entry. For example, the predicate in the case of the trie is a character, while the predicate in the case of kd-tree is a point.

3. ***int predicate_len:*** predicate_len is the length of the passed entry_pred. It is useful in the cases in which the length of the entry predicate is not fixed. For example, in the case of the trie, the data key is a string of variable length.

4. ***int depth:*** depth is the depth of the currently processed node in the index tree. The depth is important in index structures such as kd-tree. The root node of the index tree has a depth equals 0.

5. ***int type:*** type specifies the type of the currently processed node. If the node is a leaf node (contains data keys), then type has value 0. Otherwise type has value 1.

6. ***predicate *node_pred:*** node_pred is a pointer to the predicate of the currently processed node. For example, the node predicate of a leaf node in the *point quadtree* is the space covered by the node.

Consistent() returns a Boolean value. TRUE means that the passed node entry satisfies the query, while FLASE means that the passed node entry does not satisfy the query. This Boolean value guides the *search()* function while traversing the index tree.

- **NN_Consistent()**

NN_Consistent() function is called from the NN_search() internal function to search for the nearest neighbors of a given query object. The parameters for the NN_Consistent() are the same as the parameters for Consistent() function plus two more parameters:

1. ***float * distance:*** distance is an output parameter in which NN_Consistent() returns the minimum possible distance between the query object and the passed node entry. If the node entry contains a data key, then the returned distance is the actual distance between the query object and the data key.

2. ***float *parent_dist:*** parent_dist is an input parameter through which the NN_search() passes the minimum distance of the parent node. This is important in some index structures such as the trie in which computing the minimum distance of the current node needs the minimum distance of the parent node to be available.

- **Penalty()**

Penalty() function is called from the insert() internal function to guide the tree's traversal until we reach a leaf node in which the given key should be inserted. Penalty() returns value 0 for the entries that need to be traversed further, and returns Max_Penalty otherwise. The parameters to Penalty() function are:

1. ***predicate *entry_pred:*** entry_pred is a predicate of the passed node entry. It specifies the space covered by the subtree of this entry.

2. ***int pred_len:*** pred_len is the length of the node predicate.

3. ***keyType *key:*** key is the key to be inserted in the index.

4. ***int keylen:*** keylen is the length of the passed key.

5. ***int depth:*** depth is the depth of the passed node in the index tree. The root node has depth 0.

6. ***int type:*** type specifies the type of the currently processed node. If the node is a leaf node (contains data keys), then type has value 0. Otherwise type has value 1.

7. ***predicate *node_pred:*** node_pred is a predicate of the passed node.

8. ***penalty *p:*** p is an output parameter that is assigned value 0 if the entry needs to be traversed further, and value Max_Penalty otherwise.

9. ***void *Entry:*** Entry is an array of predicates. Entry will contain the predicate that matches the inserted key. In the index structure allows Node_Shrink, i.e., creating partitions only if they contain keys, it is possible that Penalty() returns Max_penalty for

all entries of a given node, i.e., no partition is available for the given key. In this case, Insert() function uses Entry array to create the appropriate entry for this key.

10. ***int entry_len:*** entry_len is the length of the predicate created in Entry array.

- **Split()**

Split() function is called from the Insert() internal function to split an overflow leaf node. Split() converts the passed leaf node into a nonleaf node and creates the required leaf nodes. The parameters to Split() function are:

1. ***cursor *p_cursor:*** p_cousor contains all the data keys that will be distrusted over the new leaf nodes.

2. ***int **Entries:*** Entries is a pointer to an array of two dimensions, columns represent the new child nodes (leaf nodes), while rows represent the id's of the data keys stored in each child node. Entries is an output parameter.

3. ***int *num_entries:*** num_entries is an array that stores the number of keys in each new leaf node.

4. ***int *num_of_child:*** num_of_child stores how many children nodes (leaf nodes) are created. If the index structure allows Node_Shrink option, i.e., creating partitions only if they contain keys, then num_of_child can be less than the number of the possible space partitions.

5. ***void **old_pred:*** old_pred is the predicate of the node to be split. old_pred specifies the space covered by this node.

6. ***int *old_pred_len:*** old_pred_len is the length of the old_pred.

7. ***void **Entries_pred:*** this is an array that contains a predicate for each newly created leaf node. Entries_pred is an output parameter.

8. ***int *Entries_pred_len:*** Entries_pred_len is an array that contains the length of each predicate in Entries_pred. Entries_pred_len is an output array.

9. ***int *depth:*** depth is the depth of the node to be split.

10. ***int *type:*** type is the type of the node to be split. In most cases it has value 0 (leaf node).

CONTRIBUTION TO THE PROJECT

Please send your patches to rhassan@cs.purdue.edu

