

Basic text indexing with DB2 index extensions

Create customized hash indexes and inverted text indexes for strings

Skill Level: Intermediate

[Knut Stolze, Ph.D. \(stolze@de.ibm.com\)](mailto:stolze@de.ibm.com)

DB2 z/OS Data Warehouse Accelerator, IBM Germany Research and Development
IBM

15 May 2008

All major vendors of database systems provide extensions for text indexing and text search. If the full power of those products is not needed, IBM® DB2® index extensions are a powerful mechanism to implement light-weight text indexes using user-defined index structures. In this article, learn how to use index extensions to implement two index structures, where one creates a hash value for strings and the other indexes strings based on the words in them. Both can be used for many purposes.

Introduction

If you store strings in columns of type VARCHAR or CHAR in a DB2 database, you often have the need to apply content-based predicates on those strings. You can either use LIKE predicates to evaluate the strings or deploy a product like the DB2 Net Search Extender. In this article, explore another approach that takes advantage of index extensions and their capabilities. While index extensions are by no means as powerful as a full-text index for the intended purposes, you may encounter situations where a light-weight text search that is still index-based is desired.

The elements needed to define an index extension (and, with it, extended indexes on your data) are fairly simple: the only requirement is to define how index entries are derived from the values stored in tables, how to apply range searches on those index entries, and under which conditions such an extended index can be exploited.

Implementation details can be found in the [Download](#) section.

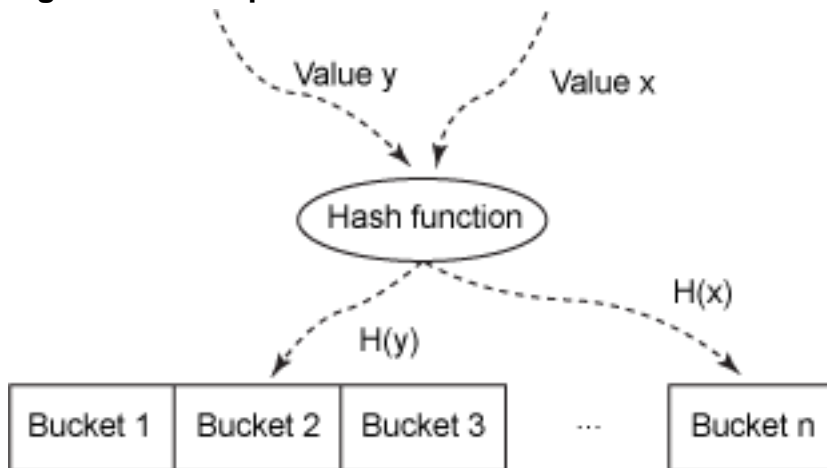
The [first example](#) in this article describes the basic concepts of hash indexes and illustrates how a hash index can easily be integrated in your DB2 database. The [second](#) gives an example of the implementation of an inverted text index.

Indexing strings by hash values

Initially, hash indexing was developed to support fast searches in files in the file system. With a hash index you can find a record in a given file in constant time by generating a hash value for the record and then looking up the record in the file based on that hash value. Thus, the search in a large file using a hash index is faster than scanning the entire file. Many different variations for hash indexes were developed and investigated in the literature (see [Resources](#) section). The various hash index methods are intended to address different aspects of the hashing itself, like the handling of collisions or overflows. Those details are not discussed here; instead this article describes the general idea and its adaptation to integrate a hash index into a database system using the DB2 index extension mechanism.

This article uses a rather basic hash index as an example. A function called `equalString`, which compares two given strings to see if they are equal, is supported by an extended index based on hashing. Java defines the method `hashCode` for all objects. This method computes a hash value for each object. Since I implement the index extension using the Java programming language, I reuse this method as-is and do not implement a new hash function myself. The generated hash value is numeric, and I use this value as single index key for the string being indexed. All numeric hash values are stored using the index extension in the DB2-internal B-Tree index. Searching for a hash value with range predicates is straight-forward. You can explicitly search for a known hash value with point ranges, in other words ranges with identical start and stop keys.

Figure 1. Concept of hash functions



A hash index is often implemented using an array where each element in the array is a hash bucket with a fixed size, as is depicted in Figure 1. The number of buckets is typically much smaller than the domain of input values for the hash function. The hash value identifies the respective hash bucket. Each hash function can produce collisions; in other words, two different input values can have the same hash code $H(x)$. Given that and the restricted size of the hash buckets, a mechanism is necessary to handle overflows, that is, hash values that belong to a certain hash bucket but do not fit any more because the bucket is already full. Various techniques have been suggested to handle such overflows. Fortunately, the whole discussion of overflows does not apply to the hash index that I implemented. The reason is that the underlying DB2 internal data structure stores the hash values in a tree, and duplicates are allowed in such a tree. So I can completely rely on DB2 handling collisions.

Combining this short analysis with the description of index extensions (see [Resources](#) section), I can now summarize the mechanism to find equal strings using hashing:

1. The key generator computes the hash code for the given string, and that hash code is stored in the DB2-internal B-Tree.
2. The range producer computes the hash code for the given string to search for and returns a point range where the start and the end of the range is the hash code itself. Thus, all indexed strings with the same hash code are returned from the index scan as possible candidates.
3. In the final step, the user-defined function (UDF) itself is evaluated to filter out all false positive candidates, in other words, remove all non-equal strings.

One thing to be aware of regarding DB2 index extensions is the requirement for user-defined types — either distinct types or structured types. For this article scenario, I created a distinct type named HashString, whose source type is a VARCHAR(3500).

Key generator

The implementation of the key generator is very simple. The code in [Listing 1](#) looks a bit more complex because the function needs to be implemented as a table function, thus the handling for the different call types and the end-of-table condition is required. The main logic, however, is just the single line that is set in ***bold italics*** in Listing 1, below:

Listing 1. Implementation of the hash index key generator

```

public void generateIndexKey(String str, int hashKey) throws Exception
{
    switch (getCallType()) {
        case SQLUDF_TF_OPEN:
            indicateEOF = false;
            break;

        case SQLUDF_TF_FETCH:
            if (indicateEOF == true) {
                setSQLstate("02000");
                break;
            }

            // compute hash value and return it as index entry

                set(2, str.hashCode());

            indicateEOF = true;
            break;
    }
}

```

Registering this function in your database is also straight-forward. Two requirements from the index extensions are that the key generator is a deterministic and does not use SQL. Both are fulfilled, so the SQL statement in Listing 2 can be used to create the UDF:

Listing 2. Registering the hash index key generator UDF

```

CREATE FUNCTION HashKeyGenerator ( string HashString )
  RETURNS TABLE ( hashKey INTEGER )
  SPECIFIC hash_key_gen
  EXTERNAL NAME 'HashIndex.generateIndexKey' LANGUAGE JAVA
  PARAMETER STYLE DB2GENERAL
  DETERMINISTIC RETURNS NULL ON NULL INPUT NOT FENCED NO SQL
  NO EXTERNAL ACTION NO SCRATCHPAD NO FINAL CALL DISALLOW PARALLEL
  NO DBINFO@

```

Range producer

The range producer is nearly as simple as the key generator, except that it needs to set two output parameters, which will define the range, as indicated in ***bold italics*** in Listing 3. Otherwise, the code for the function is identical, as Listing 3 shows:

Listing 3. Implementation of the Hash Index Range Producer

```

public void produceSearchRanges(String str, int startKey,
    int stopKey) throws Exception
{
    switch (getCallType()) {
        case SQLUDF_TF_OPEN:
            indicateEOF = false;
            break;

        case SQLUDF_TF_FETCH:
            if (indicateEOF == true) {
                setSQLstate("02000");

```

```

        break;
    }

        set(2, str.hashCode());

        set(3, str.hashCode());

    indicateEOF = true;
    break;
}
}
}

```

The `CREATE FUNCTION` statement is, as you would expect, very similar to the statement for the key generator, except that the resulting table consists now of two columns. Listing 4 shows the SQL statement:

Listing 4. Registering the Hash Index Range Producer

```

CREATE FUNCTION HashRangeProducer(string HashString)
  RETURNS TABLE ( start INTEGER, stop INTEGER )
  SPECIFIC hash_range_prod
  EXTERNAL NAME 'HashIndex.produceSearchRanges' LANGUAGE JAVA
  PARAMETER STYLE DB2GENERAL
  DETERMINISTIC RETURNS NULL ON NULL INPUT NOT FENCED
  NO SQL
  NO EXTERNAL ACTION NO SCRATCHPAD NO FINAL CALL
  DISALLOW PARALLEL
  NO DBINFO@

```

The index extension and user-defined predicate

The final step is to define the index extension itself and the user-defined predicates that can make use of an extended index.

The article "[Using DB2 index extensions by example and in detail](#)" (developerWorks, December 2003) discusses how the different pieces of index extensions play together. Let's now focus on the important items from [Listing 5](#). We need to specify which function should be used to generate the index entries from the given input value. This is the `HashKeyGenerator` UDF in this case, and it is marked in ***bold italics***. The second portion is the definition of the search methods, which includes the specification of the function that is to be used to produce the search ranges. I have only one search method, and the UDF `HashRangeProducer` is designated for this task. The relevant piece in the SQL statement is marked in **bold**. The user-defined predicate is associated to the function `equalString`, as shown in *italics* in Listing 5. It says that the function must be used in an expression involving the comparison operation and that the search method "equals" of the index extension `hash_index` is referred to.

Listing 5. Defining the index extension and user-defined predicate

```

CREATE INDEX EXTENSION hash_index
  FROM SOURCE KEY ( string HashString )

          GENERATE KEY USING HashKeyGenerator(string)

WITH TARGET KEY ( hashCode INTEGER )
SEARCH METHODS
WHEN equals(searchString HashString)
          RANGE THROUGH HashRangeProducer(searchString)@

CREATE FUNCTION equalString(
  str1 HashString, str2 HashString)
  RETURNS INTEGER
  SPECIFIC equal_str_udp
  LANGUAGE SQL DETERMINISTIC
  NO EXTERNAL ACTION CONTAINS SQL
  PREDICATES (
    WHEN = 1
      SEARCH BY INDEX EXTENSION hash_index
      WHEN KEY (str1) USE equals(str2)
      WHEN KEY (str2) USE equals(str1) )
  RETURN CASE WHEN str1 IS NULL OR str2 IS NULL THEN NULL
    WHEN str1 = str2 THEN 1 ELSE 0 END@

```

With all the definitions in place, you can test the function `equalString` and the index extension. As you could see in Listing 5, the function just compares the two given strings and returns 1 (one) if they are equal. The search through the index extension might be added up front if such an extended index was created on a column that is one of the parameters of the function and if the DB2 optimizer chooses to exploit the index. So I build a scenario, as shown in Listing 6. A table is properly created, an extended hash index is created on the table, data is inserted, and the key generator and range producer functions are tested to see which index entries are stored in the internal B-Tree, and which ranges are used to query those index entries. The final query is the query that exploits the extended index.

Listing 6. Testing the hash index

```

CREATE TABLE strings ( id INTEGER NOT NULL PRIMARY KEY, str HashString NOT NULL )@
CREATE INDEX hash_idx ON strings(str) EXTEND USING hash_index@

INSERT INTO strings
VALUES ( 1, HashString('abc def ghi') ),
      ( 2, HashString('abc') ),
      ( 3, HashString('def') ),
      ( 4, HashString('hello') ),
      ( 5, HashString('xyz') ),
      ( 6, HashString('abcdefghijklmnopqrstuvwxyz') )@

SELECT s.id, idx.* FROM strings AS s, TABLE ( HashKeyGenerator(str) ) AS idx@

```

ID	HASHKEY
1	-1655199921
2	96354
3	99333
4	99162322
5	119193
6	958031277

```
6 record(s) selected.
SELECT * FROM TABLE ( HashRangeProducer(HashString('def')) ) AS rng@
RANGESTART  RANGESTOP
-----
          99333          99333

1 record(s) selected.
SELECT id FROM strings AS s WHERE equalString(str, HashString('def')) = 1@
ID
-----
          3

1 record(s) selected.
```

If you run the final query from Listing 6, as it is shown above, you should see in the access plan that DB2 decides to use the extended index by the EISCAN operator. The respective access plan is visualized in [Figure 2](#). For comparison, [Figure 3](#) illustrates the access plan where I forced the DB2 optimizer to ignore the index by adding the clause `SELECTIVITY 1` to the predicate in the `WHERE` clause.

Figure 2. Access plan for query accessing a hash Index

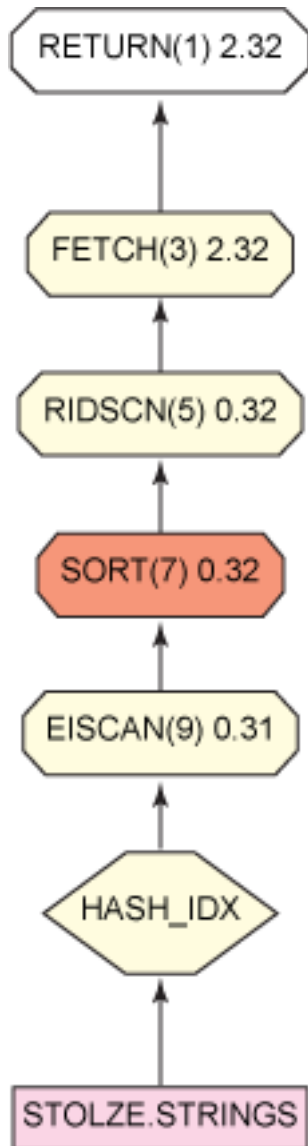
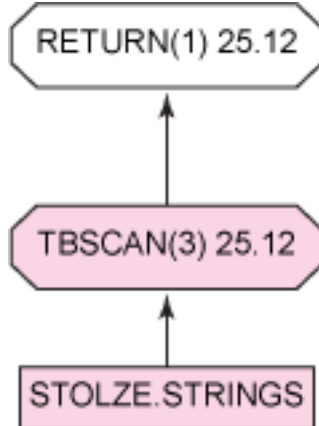


Figure 3. Access plan for query without hash Index



Inverted text index

This section describes a simple mechanism for creating an inverted text index by using index extensions. The functionality described here is by no means a replacement for dedicated full-text indexing products like the DB2 Net Search Extender. Such products are highly specialized, and can deliver a much richer feature set and even better performance for more complex text searches. However, the concepts and mechanisms described here show you how powerful index extensions can be, and if the limited functionality is already sufficient for your applications, this may be an option that you could pursue further.

Let's first discuss the scope of the functionality that the inverted text index provides. All words in text, stored as value of type `Text`, are extracted. Each word is an index entry for the document it originates from, and the index entries are sorted by those words. For example, a text "abc def ghi" consists of three words, and that results in three index entries (all pointing to the same text document) being generated. Thus, if you search for all texts that contain a specific word, DB2 only has to look up all the index entries for this word, and it will know exactly which texts qualify that way. [Listing 7](#) demonstrates how this works with a user-defined function named `contains`. Each row in the temporary table is comprised of a text (for example, "abcd-def ghi") and a word to search for (for example, "def"). The UDF returns 1 (one) if the word (honoring word boundaries) occurs in the text; otherwise, 0 (zero) is the result.

Listing 7. Using a UDF to search texts containing a specific word

```
CREATE DISTINCT TYPE Text AS VARCHAR(2000) WITH COMPARISONS@

SELECT text.contains(text, word) AS contains
FROM   TABLE ( VALUES ( Text('abc def ghi'), 'abc' ),
                        ( Text('abc-def ghi'), 'abc' ),
                        ( Text('abcd-def ghi'), 'abc' ),
                        ( Text('abcd-def ghi'), 'def' ),
                        ( Text('abc def ghi'), 'ghi' ),
                        ( Text('abc def ghi'), 'abcd' ) ) AS t(text, word)@

CONTAINS
-----
      1
      1
      0
      1
      1
      0

6 record(s) selected.
```

Defining the index extension

As before with the hash index, you need to define a key generator UDF, a range producer UDF, the index extension, and finally a UDF that exploits the index

extension if possible. The Java implementation of the key generator and the range producer UDFs is summarized in [Listing 8](#). The main task is to extract the words from the texts in the key generator. The Java method `String.split` is used for that (set in ***bold italics*** in Listing 8). Then all words are placed in a mathematical set, and the set operations will automatically eliminate duplicate words. All this is done in the `OPEN` call to the table function; in the `FETCH` calls, the next word is retrieved from the set and returned as index entry. That is done until all words are processed and, thus, the text is completely indexed.

Listing 8. Implementation of the key generator and range producer

```
public void generateIndexKey(String str, String word) throws Exception
{
    switch (getCallType()) {
        case SQLUDF_TF_OPEN:
            // split the given text at all non-word characters

            String words[] = str.split("\\W+");

            // construct a (mathematical) set of all the words
            wordList = new TreeSet();
            for (int i = 0; i < words.length; i++) {
                wordList.add(words[i]); // ignore duplicates
            }
            wordListIter = wordList.iterator();
            break;

        case SQLUDF_TF_FETCH:
            while (wordListIter.hasNext() == true) {
                String w = (String)wordListIter.next();
                if (w == null || w.length() == 0) { // ignore empty strings
                    continue;
                }
                set(2, w);
                return;
            }
            // we reached the end of the word list; signal end-of-table
            setSQLstate("02000");
            break;

        case SQLUDF_TF_CLOSE:
            // cleanup
            wordListIter = null;
            wordList.clear();
            wordList = null;
            break;
    }
}

public void produceSearchRanges(String word, String startWord, String stopWord)
throws Exception
{
    // test that the given word does not contain any special characters
    for (int i = 0; i < word.length(); i++) {
        if (isWordChar(word.charAt(i)) == false) {
            setSQLstate("38T01");
            setSQLmessage("Invalid word '" + word + "'.");
            return;
        }
    }
}

// depending on the current call type to the table function, do the
```

```

// initialization or return the next element from the wordList
switch (getCallType()) {
  case SQLUDF_TF_OPEN:
    // the first FETCH call needs to return a result
    indicateEOF = false;
    break;

  case SQLUDF_TF_FETCH:
    if (indicateEOF == true) {
      // signal end-of-table
      setSQLstate("02000");
      break;
    }

    // set return value
    set(2, word);
    set(3, word);

    // indicate that the next FETCH call needs to indicate the
    // end-of-table condition
    indicateEOF = true;
    break;
}
}
}

```

Registering both functions is very similar to the SQL statements in [Listing 2](#) and [Listing 4](#). The same applies to the definition of the index extension and the UDF `text.contains`, which has the necessary user-defined predicates attached. You can find those details in the sample code accompanying this article (see [Download](#)). The remaining task is to verify the proper functioning of the key generator and range producer functions. [Listing 9](#) demonstrates how to do this by explicitly calling the respective functions. The ID column in the result set establishes the relationship between index entry and the text value.

Listing 9. Testing the key generator and range producer functions

```

CREATE TABLE texts (
  id    INTEGER NOT NULL PRIMARY KEY,
  text  Text    NOT NULL
)@
CREATE INDEX text_idx ON texts(text) EXTEND USING text_index@

INSERT INTO texts
VALUES ( 1, 'abc def ghi' ),
      ( 2, 'some text' ),
      ( 3, 'and now for something completely different..., the larch' )@

SELECT texts.id, idx.*
FROM   texts, TABLE ( TextKeyGenerator(text) ) AS idx@

```

ID	WORD
1	abc
1	def
1	ghi
2	some
2	text
3	and
3	completely
3	different
3	for

```

      3 larch
      3 now
      3 something
      3 the

13 record(s) selected.

SELECT *
FROM   TABLE ( TextRangeProducer('and') ) AS t@

RANGESTART          RANGESTOP
-----
and                 and

1 record(s) selected.

SELECT id
FROM   texts
WHERE  text.contains(text, 'some') = 1@

ID
-----
      2

1 record(s) selected.

```

Comparing the access plans in [Figure 4](#) and [Figure 5](#) shows again with the EISCAN operator that an extended index is used by the query. That means, the range producer function is invoked to create the start/stop keys for a range scan on the DB2-internal B-Tree, which equates to point ranges for the search argument. In the query above, the search range is defined by the word "some". While it may seem that the access plan in [Figure 5](#) is simpler, it comes with a higher estimated execution cost because no index could be exploited to filter out non-qualifying rows.

Figure 4. Access plan for query accessing an inverted text index

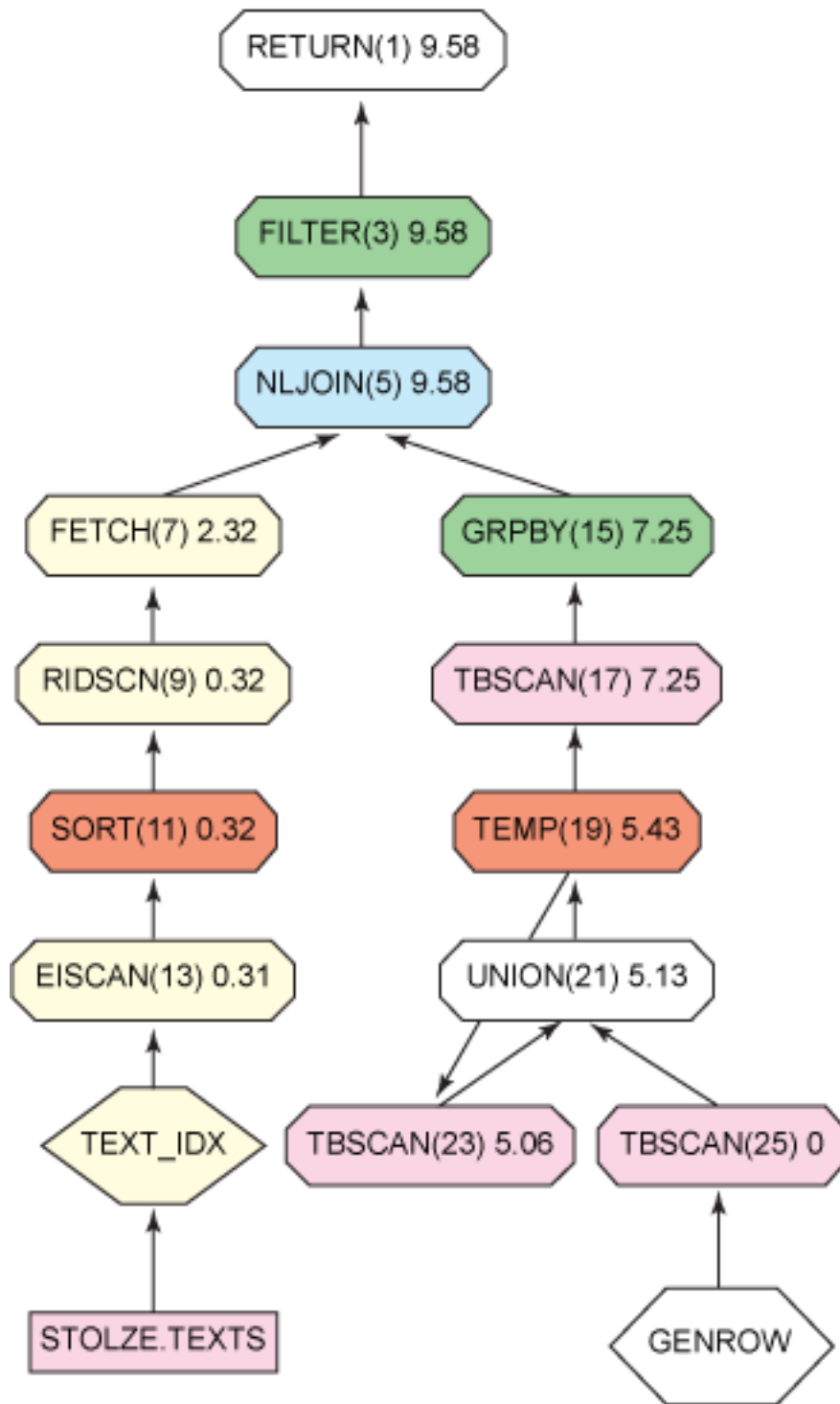
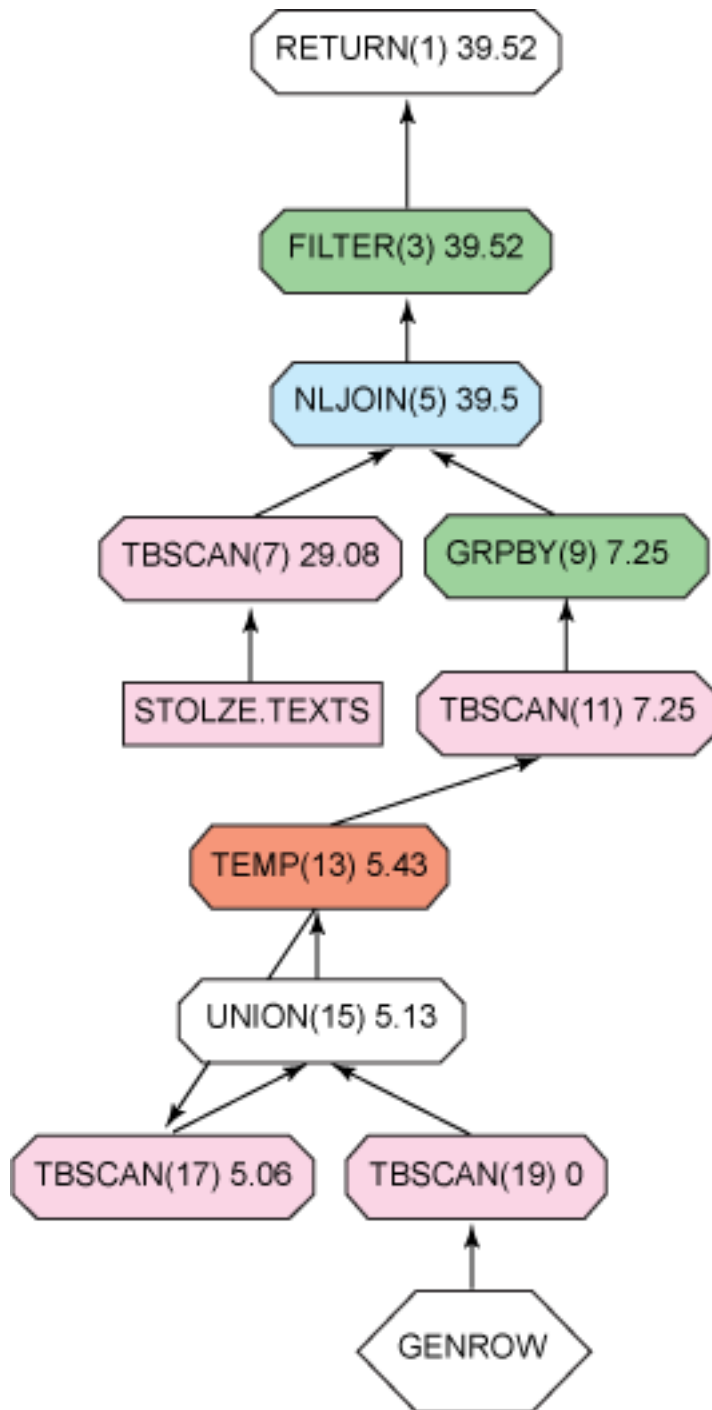


Figure 5. Access plan for query without inverted text index



Summary

This article demonstrated how DB2 index extensions can be used to index texts or strings. In one example, the search for exact strings can be implemented by generating a hash value for the string, and then indexing this hash value. While you

can realize such a function-based index with DB2's generated columns, the approach described in this article does not impose the requirement to add another column to the table. The second scenario creates an inverted text index from strings. The strings are split at word-boundaries into separate words, and those words form the index entries. This article has shown that this can be accomplished with rather negligible coding effort. You may build on the described functionality, and adapt and extend it to your own needs. For example, the index extension for the inverted text index can be parameterized to also support case-sensitive and case-insensitive indexes.

Downloads

Description	Name	Size	Download method
Sample code for this article	samples.zip	11KB	HTTP

[Information about download methods](#)

Resources

Learn

- [DB2 Net Search Extender manual](#): Understand the full range of text search functionality that this product provides.
- ["DB2 Index Extensions by example and in detail"](#) (developerWorks, December 2003): Find further detail about the concepts of DB2 Index Extensions.
- ["Hash Table Methods"](#) (ACM, 1978): Get an extensive introduction to hash functions
- [developerWorks Information Management zone](#): Learn more about Information Management. Find technical documentation, how-to articles, education, downloads, product information, and more.
- Stay current with [developerWorks technical events and webcasts](#).
- [Technology bookstore](#): Browse for books on these and other technical topics.

Get products and technologies

- Build your next development project with [IBM trial software](#), available for download directly from developerWorks.

Discuss

- [Participate in the discussion forum for this content](#).
- Participate in [developerWorks blogs](#) and get involved in the developerWorks community.

About the author

Knut Stolze, Ph.D.

Knut Stolze started his work with DB2 when he joined IBM as a visiting scientist at the Silicon Valley Lab, USA, in 1999 where he worked on the DB2 Image Extender. He moved on to the DB2 Spatial Extender development team and was responsible for several enhancements to improve usability, performance, and standard-conformance of the Extender. From 2002 through 2006 he was a PhD student and teaching assistant at the University of Jena, Germany. At the same time he continued his work for IBM in the area of the Information Integrator development. Today, Knut Stolze is a member of the development team responsible for DB2 z/OS Utilities and DB2 z/OS Data Warehouse Accelerator (BLINK). He can be reached through the newsgroups [comp.databases.ibm-db2](#) and [ibm.software.db2.udb.spatial](#) or at stolze@de.ibm.com.