DNA SEQUENCE DATABASE EXTENSION

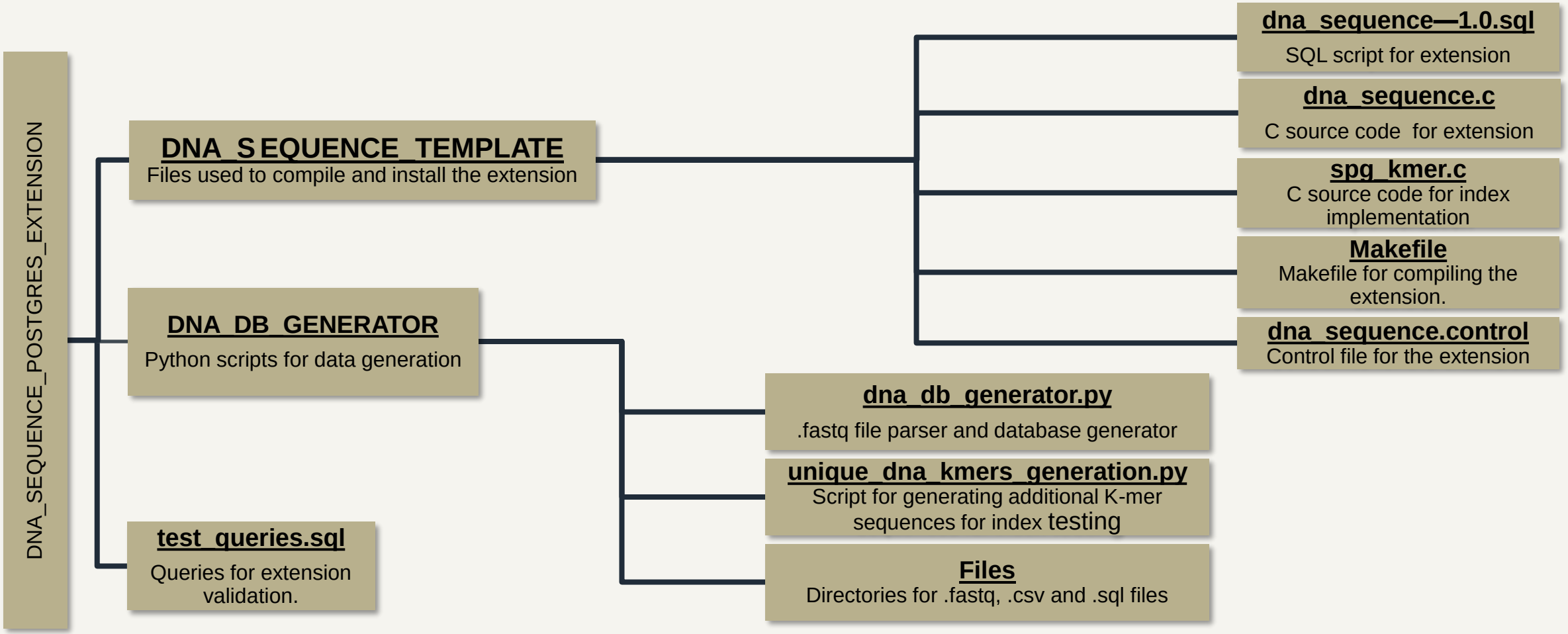
our GitHub repository

Kristóf Balázs - 000612294
Stefanos Kypritidis - 000606810
Otto Wantland - 000607140
Nima Kamali Lassem - 000605572

INFO-H417 Database System Architecture
2024

INTRODUCTION

DNA SEQUENCE REPO STRUCTURE



KMER AND QKMER STRUCTURES

IUPAC NOTATION

DESCRIPTION	SYMBOL	BASES REPRESENTED					COMPLEMENTARY BASES
		NO.	A	C	G	T	
ADENINE	A	1	A				T
CYTOSINE	C			C			G
GUANINE	G				G		C
THYMINE	T					T	A
URACIL	U					U	A
WEAK	W	2	A			T	W
STRONG	S			C	G		S
AMINO	M		A	C			K
KETONE	K				G	T	M
PURINE	R		A		G		Y
PYRIMIDINE	Y			C		T	R
NOT A	B	3		C	G	T	V
NOT C	D		A		G	T	H
NOT G	H		A	C		T	D
NOT T	V		A	C	G		B
ANY ONE BASE	N	4	A	C	G	T	N
GAP	-	0					-

K-MERS FOR TCTAATTAGT

K	K-MERS
1	T, C, A, G
2	TC, CT, TA, AA, AT, TT, TA, AG, GT
3	TCT, CTA, TAA, AAT, ATT, TTA, TAG, AGT
4	TCTA, CTAA, TAAT, AATT, ATTA, TTAG, TAGT
5	TCTAA, CTAAT, TAATT, AATTA, ATTAG, TTAGT
6	TCTAAT, CTAATT, TAATTA, AATTAG, ATTAGT
7	TCTAATT, CTAATTA, TAATTAG, AATTAGT
8	TCTAATTA, CTAATTAG, TAATTAGT
9	TCTAATTAG, CTAATTAGT
10	TCTAATTAGT

DATA TYPES

- *K*-mer: substring of length k within a biological sequence
- *QK*-mer: *K*-mer representation with IUPAC notation

Source: en.wikipedia.org

REAL WORLD APPLICATION: BIOINFORMATICS

Whole Genome Sequencing (WGS) in Bioinformatics

Storage

- WGS requires storing massive amounts of data (~6 billion base pairs genome).

Processing

- Genome reconstruction and annotation require analyzing different K-mers for repetitive sequences and overlaps.

Analysis

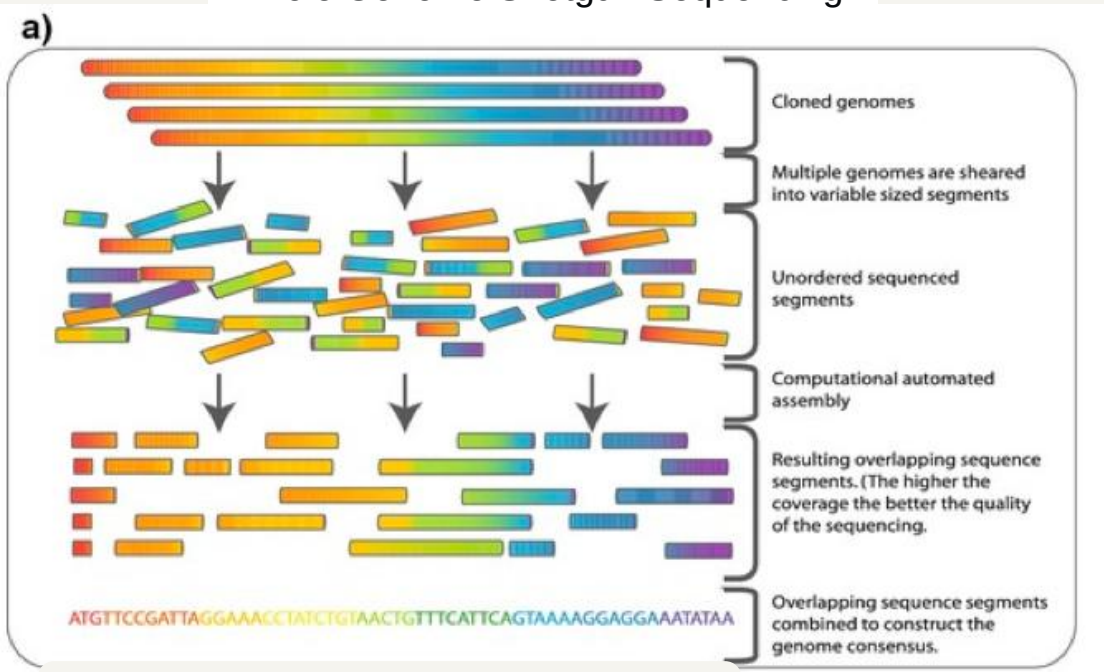
- Mutations can be identified by comparing different strands of the genome.

Sources: pmc.ncbi.nlm.nih.gov, spectrum.ieee.org

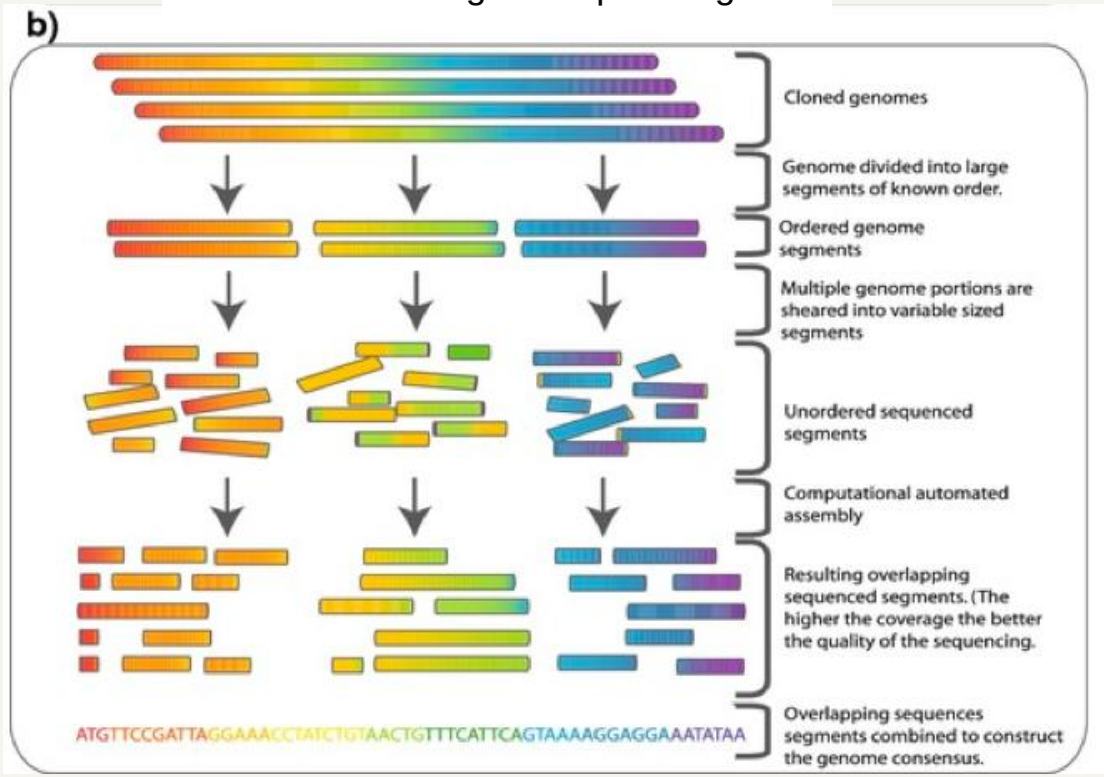
REAL WORLD APPLICATION: EXAMPLE

Example of Whole Genome Sequencing

Whole Genome Shotgun Sequencing



Hierarchical Shotgun Sequencing



Sources: Commins, J., Toft, C., Fares, M. A.

DATA TYPES

DNA SEQUENCE ANALYSIS TYPES

Custom Data Types for DNA

DNA Sequence

- A, C, G, T Nucleotides
- No max length

K-mer

- A, C, G, T Nucleotides
- Max length = 32

Query K-mer

- A, C, G, T Nucleotides
- Ambiguity characters: B, D, H, K, M, N, R, S, V, W, Y
- Max length = 32

KMER TYPE STRUCTURE

length: number of nucleotides of kmer

char_data[FLEXIBLE_ARRAY_MEMBER] :a flexible array member that holds the k-mer sequence

Same logic for other Dna and Q-kmer

Use flexible array member (FAM)

- At least one named member before the FAM.
- FAM should only appear as the last member of the structure.

Text sequences of various lengths are stored efficiently

In dna_sequence.h

```
typedef struct {
    int32 length;
    char data[FLEXIBLE_ARRAY_MEMBER];
} Kmer;
```

KMER TYPE IN FUNCTION

In dna_sequence.c

```c
/* Input function kmer sequence */
PG_FUNCTION_INFO_V1(kmer_in);
Datum kmer_in(PG_FUNCTION_ARGS) {
    char *input = PG_GETARG_CSTRING(0);
    int32 length;
    Kmer *result;

    // Validate the kmer string
    if (!is_valid_kmer_string(input)) {
        ereport(ERROR,
        (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
         errmsg("Invalid input: only 'A', 'C', 'G', 'T' \
         characters are allowed and length must be <= %d", KMER_SIZE)
    }

    length = strlen(input);
    result = (Kmer *) palloc(VARHDRSZ + length);
    SET_VARSIZE(result, VARHDRSZ + length);
    for (int i = 0; i < length; i++) {
        result->data[i] = toupper(input[i]);
    }

    PG_RETURN_POINTER(result);
}
```

Converts a C Sting inputted by the user into a kmer format for storage

PG_FUNCTION_INFO_V1(): registers function with PostgreSQL, marking it as callable from SQL

input: a pointer to a char, thus holds the address of the string

PG_GETARG_CSTRING(0): retrieves the first argument passed to the function as a C-style string

palloc: allocates memory in the PostgreSQL memory context

toupper: converting characters to uppercase

KMER TYPE OUT FUNCTION

Convert the internal representation of the k-mer back into a human-readable string, e.g. for a SELECT query

PG_GETARG_POINTER(0): retrieves the first value (a pointer) passed to a PostgreSQL function

input: initialized with the pointer and now points to the memory location of kmer passed to this function

length + 1: adds one additional byte for the null terminator (\0). Requirement for C-strings

memcy: allows kmer to be copied into the result buffer. Raw binary representation is converted into readable

PG_FREE_IF_COPY(input, 0): releases memory

In dna_sequence.c

```c
/* Output function kmer sequence*/
PG_FUNCTION_INFO_V1(kmer_out);
Datum kmer_out(PG_FUNCTION_ARGS){
    Kmer *input = (Kmer *) PG_GETARG_POINTER(0);
    int32 length = VARSIZE(input) - VARHDRSZ;
    char *result = (char *) palloc(length + 1);

    memcpy(result, input->data, length);
    result[length] = '\0';

    PG_FREE_IF_COPY(input, 0);
    PG_RETURN_CSTRING(result);
}
```

KMER TYPE VALIDATION FUNCTION

In dna_sequence.c

```c
bool is_valid_kmer_string(const char *str){
    int len = strlen(str);
    if (len > KMER_SIZE) {
        ereport(ERROR,
            (errcode(ERRCODE_STRING_DATA_RIGHT_TRUNCATION),
            errmsg("Input exceeds maximum length of %d", KMER_SIZE)));
    }
    for (int i = 0; i < len; i++) {
        if (toupper(str[i]) != 'A' && toupper(str[i]) != 'C'
            && toupper(str[i]) != 'G' && toupper(str[i]) != 'T') {
            return false;
        }
    }
    return true;
}
```

strlen(): calculate String Length

If length exceeds 32,
raises an error with corresponding message

Iterate over each character

Returns true, only if all characters are one of A,C,G,T (case insensitive)

KMER SQL FUNCTIONS

In dna_sequence--1.0.sql

Defines a custom PostgreSQL data type for k-mer sequences

kmer_in: points to the relevant C function also called kmer_in

kmer_out: points to the relevant C function, called kmer_out

internallength = VARIABLE: the internal representation of kmer has a variable size

category = 'S': Classifies the type as string-like. Can help control which implicit cast will be applied in ambiguous situations.

```
-- Function to convert text to kmer
CREATE OR REPLACE FUNCTION kmer_in(cstring)
    RETURNS kmer
    AS 'MODULE_PATHNAME', 'kmer_in'
    LANGUAGE C IMMUTABLE STRICT PARALLEL SAFE;

-- Function to convert kmer to text
CREATE OR REPLACE FUNCTION kmer_out(kmer)
    RETURNS cstring
    AS 'MODULE_PATHNAME', 'kmer_out'
    LANGUAGE C IMMUTABLE STRICT PARALLEL SAFE;

-- Define the new kmer type
CREATE TYPE kmer (
    internallength = VARIABLE,
    input = kmer_in,
    output = kmer_out,
    category = 'S' -- classify as string-like type
);
```

FUNCTIONS AND OPERATORS

BASIC LENGTH FUNCTIONS

In dna_sequence.c

```c
/* Length Function for kmer sequence*/
PG_FUNCTION_INFO_V1(kmer_length);
Datum kmer_length(PG_FUNCTION_ARGS) {
    Kmer *input = (Kmer *) PG_GETARG_POINTER(0);
    int32 length = VARSIZE(input) - VARHDRSZ;
    PG_RETURN_INT32(length);
}
```

VARSIZE(input): total size of the variable-length Kmer object, including the header

VARHDRSZ is subtracted to exclude the size of the header

Same logic for other Dna and Qkmer, **only the types changes**

In dna_sequence--1.0.sql

```
CREATE FUNCTION length(kmer)
    RETURNS integer AS 'MODULE_PATHNAME', 'kmer_length'
    LANGUAGE C IMMUTABLE STRICT PARALLEL SAFE;
```

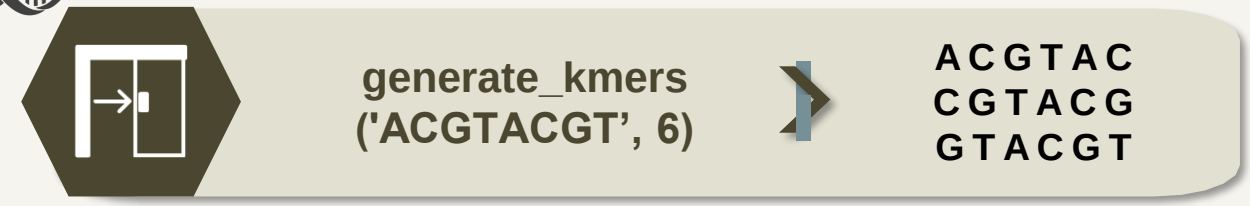
IMMUTABLE: same result for the same input >> caching

STRICT: if any arg. Is NULL >> no function call

PARALLEL SAFE: can be executed in parallel queries

Source: github.com

GENERATE KMERS



DURING THE FIRST CALL

```c
/* Done in only first call of the function */
if (SRF_IS_FIRSTCALL())
{
        /* ... */
        /* Create a function context for cross-call persistence */
        /* Switch to memory context (appropriate for multiple function calls) */
        /* Extract function arguments in the form of: ('ACGTACGT', 6)*/
        dna_input = (Dna_sequence *) PG_GETARG_POINTER(0);
        k = PG_GETARG_INT32(1);

        /* Validate k */
        dna_length = VARSIZE(dna_input) - VARHDRSZ;
        if (k <= 0 || k > dna_length || k > KMER_SIZE) {
            ereport(ERROR,
                (errmsg("Invalid k: must be between 1 and the length of the DNA sequence")));
        }

        /* ... */
        /* Allocate memory for user context */
        /* Copy DNA sequence into the user context */
        /* Store the user context */
        /* Total nr of k-mers */
        funcctx->max_calls = dna_length - k + 1;
        MemoryContextSwitchTo(oldcontext);
}
```

DURING EVERY OTHER CALL

```c
/* Done on every call of the function */
funcctx = SRF_PERCALL_SETUP();
fctx = (generate_kmers_fctx *) funcctx->user_fctx;

/* Check if there are more k-mers to return (also taken from docs) */
if (funcctx->call_cntr < funcctx->max_calls)
{
        /* ... */
        /* Get the next k-mer */
        char *kmer_str = (char *) palloc(fctx->k + 1);
        memcpy(kmer_str, fctx->dna_sequence + current_pos, fctx->k);
        kmer_str[fctx->k] = '\0';  // null-terminate

        /* Create a Kmer object */
        /* ... */
        memcpy(kmer_result->data, kmer_str, fctx->k);
        result = PointerGetDatum(kmer_result);

        /* Clean up temporary memory */
        pfree(kmer_str);

        /* Return the next k-mer */
        SRF_RETURN_NEXT(funcctx, result);
}
else
{
    /* do when there is no more left */
    SRF_RETURN_DONE(funcctx);
}
```


Source: postgresql.org

GENERATE KMERS

generate_kmers ('ACGTACGT', 6)

ACGTAC
CGTACG
GTACGT

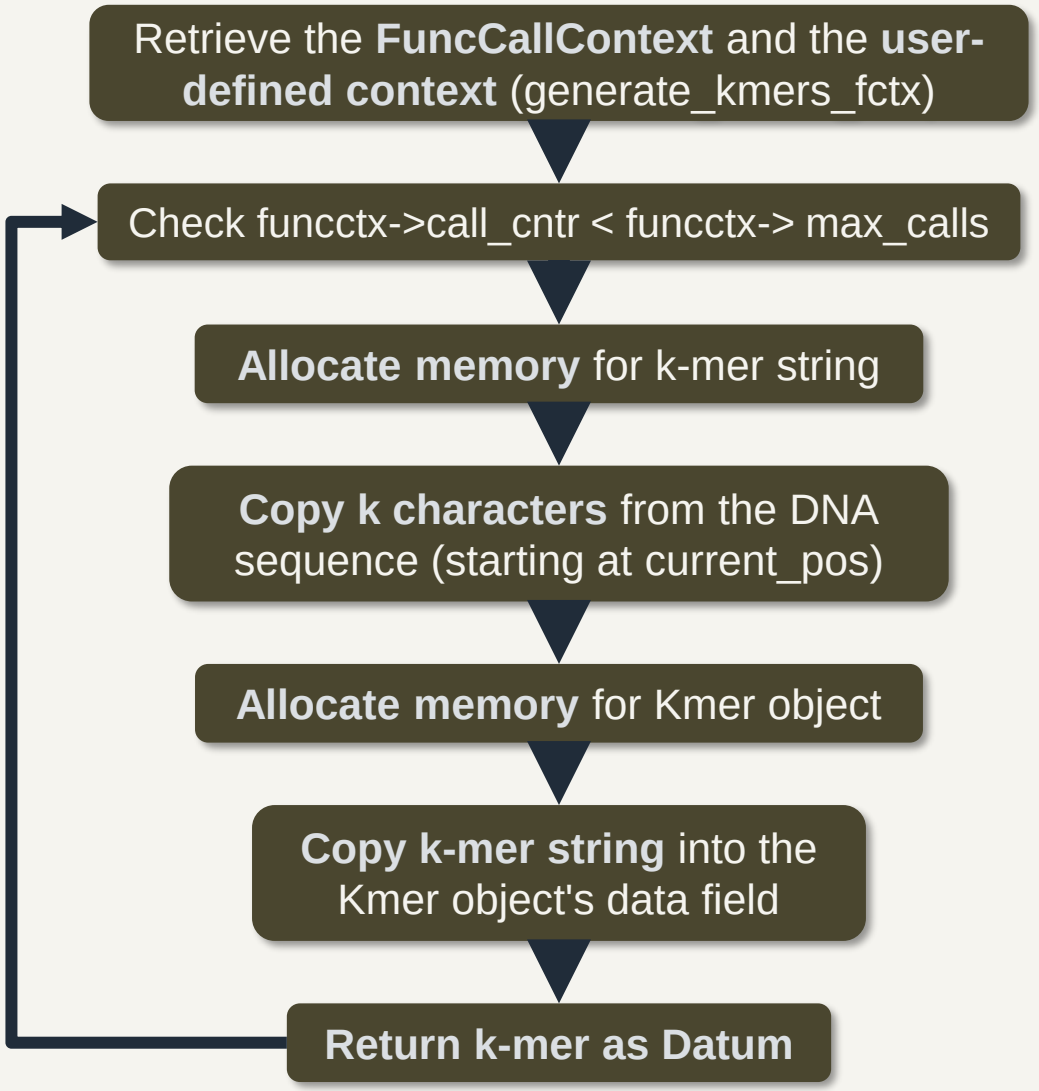
DURING THE FIRST CALL

```c
/* Done in only first call of the function */
if (SRF_IS_FIRSTCALL())
{
        /* ... */
        /* Create a function context for cross-call persistence */
        /* Switch to memory context (appropriate for multiple function calls) */
        /* Extract function arguments in the form of: ('ACGTACGT', 6)*/
        dna_input = (Dna_sequence *) PG_GETARG_POINTER(0);
        k = PG_GETARG_INT32(1);

        /* Validate k */
        dna_length = VARSIZE(dna_input) - VARHDRSZ;
        if (k <= 0 || k > dna_length || k > KMER_SIZE) {
            ereport(ERROR,
                (errmsg("Invalid k: must be between 1 and the length of the DNA sequence")));
        }

        /* ... */
        /* Allocate memory for user context */
        /* Copy DNA sequence into the user context */
        /* Store the user context */
        /* Total nr of k-mers */
        funcctx->max_calls = dna_length - k + 1;
        MemoryContextSwitchTo(oldcontext);
}
```

- Create a **function context** (FuncCallContext)
- Switch to **memory context**
- **Allocate memory for the user-defined context** (generate_kmers_fctx): DNA sequence, its length, and k)
- **Copy the DNA sequence** (excluding the header) into a null-terminated string.
- **SET max_calls** = dna_length - k + 1
- **Save** context
- **Restore** original context

Source: postgresql.org

GENERATE KMERS



DURING EVERY OTHER CALL

```c
/* Done on every call of the function */
funcctx = SRF_PERCALL_SETUP();
fctx = (generate_kmers_fctx *) funcctx->user_fctx;

/* Check if there are more k-mers to return (also taken from docs) */
if (funcctx->call_cntr < funcctx->max_calls)
{
        /* ... */
        /* Get the next k-mer */
        char *kmer_str = (char *) palloc(fctx->k + 1);
        memcpy(kmer_str, fctx->dna_sequence + current_pos, fctx->k);
        kmer_str[fctx->k] = '\0';  // null-terminate

        /* Create a Kmer object */
        /* ... */
        memcpy(kmer_result->data, kmer_str, fctx->k);
        result = PointerGetDatum(kmer_result);

        /* Clean up temporary memory */
        pfree(kmer_str);

        /* Return the next k-mer */
        SRF_RETURN_NEXT(funcctx, result);
}
else
{
    /* do when there is no more left */
    SRF_RETURN_DONE(funcctx);
}
```


Source: postgresql.org

EQUALS FOR KMERS

In dna_sequence.c

```c
PG_FUNCTION_INFO_V1(kmer_equals);
Datum kmer_equals(PG_FUNCTION_ARGS){
    /* ... */
    int32 len1 = VARSIZE(kmer1) - VARHDRSZ;
    int32 len2 = VARSIZE(kmer2) - VARHDRSZ;

    if (len1 != len2) {
        PG_RETURN_BOOL(false);
    }
    // Compares the first num bytes of the block of memory
    pointed by ptr1 to the first num bytes pointed by ptr2
    if (memcmp(kmer1->data, kmer2->data, len1) == 0) {
        PG_RETURN_BOOL(true);
    } else {
        PG_RETURN_BOOL(false);
    }
}
```

If len1!=len2, they cannot be equal, so it **immediately returns false**.

Use memcmp() to **compare** the data regions of the **two k-mers byte by byte**

returns 0 if k-mers are identical

returns false otherwise

Example:

String A: 0110 1101
String B: 0110 1111

dna_sequence--1.0.sql

```
CREATE OPERATOR = (
    LEFTARG = kmer,
    RIGHTARG = kmer,
    PROCEDURE = equals,
    COMMUTATOR = =,
    NEGATOR = <>,
    HASHES,
    RESTRICT = eqsel,
    JOIN = eqjoinsel
);
```

COMMUTATOR: a=b is equivalent to b=a

NEGATOR: added “<>” as mentioned before

HASHES: discussed later in the presentation

RESTRICT: Cost estimation -> WHERE

JOIN: Cost estimation -> JOIN

The same is implemented for kmer_not_equals, with the last condition flipped and the first PG_RETURN_BOOL(true).

Source: postgresql.org; researchgate.net

EQUALS FOR KMERS

In dna_sequence.c

```c
PG_FUNCTION_INFO_V1(kmer_cast_text);
Datum kmer_cast_text(PG_FUNCTION_ARGS) {
    text *txt = PG_GETARG_TEXT_P(0);
    char *str = text_to_cstring(txt);
    int32 length;
    Kmer *result;

    // Validate the kmer string
    if (!is_valid_kmer_string(str)) {
        ereport(ERROR, (errmsg("Invalid input:
only 'A', 'C', 'G', 'T' characters are allowed
and length must be <= %d", KMER_SIZE)));
    }

    length = strlen(str);
    result = (Kmer *) palloc(VARHDRSZ + length);
    SET_VARSIZE(result, VARHDRSZ + length);
    memcpy(result->data, str, length);
    pfree(str);

    PG_RETURN_POINTER(result);
}
```

The **implicit casting works for every operator** (=, <>, ^@, <@) for the kmer data type.

In dna_sequence--1.0.sql

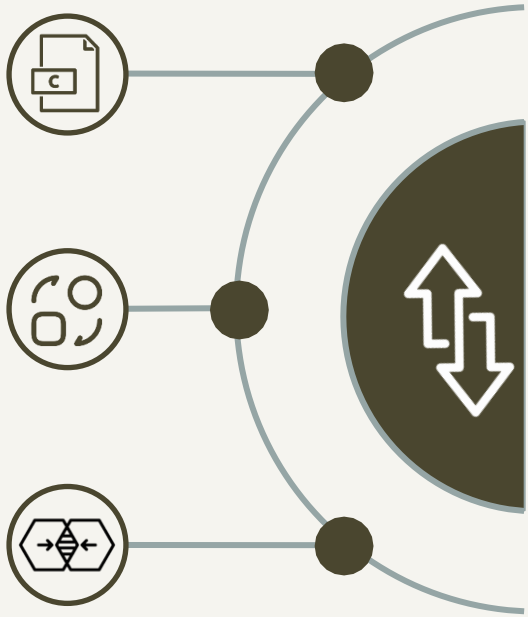
```sql
CREATE FUNCTION kmer(text)
    RETURNS kmer
    AS 'MODULE_PATHNAME', 'kmer_cast_text'
    LANGUAGE C IMMUTABLE STRICT PARALLEL SAFE;
```

```sql
CREATE CAST (text AS kmer) WITH
FUNCTION kmer(text) AS IMPLICIT;
```

The kmer_cast_text function **takes a text input**, validates it (as a valid k-mer string), and **converts it into a Kmer object**.

A **casting rule** is set in the SQL file to **convert text values into the Kmer type** using the function mentioned above.

AS IMPLICIT: the **cast occurs automatically** wherever needed without needing an explicit CAST.



Source: postgresql.org

STARTS_WITH FOR KMERS

In dna_sequence.c

```c
PG_FUNCTION_INFO_V1(kmer_starts_with);
Datum kmer_starts_with(PG_FUNCTION_ARGS) {
    /* ... */
    /* If prefix length is greater than kmer length, return false */
    if (prefix_len > kmer_len) {
        PG_RETURN_BOOL(false);
    }

    /* Compare (prefix_len bytes of) kmer->data and prefix->data */
    if (memcmp(kmer->data, prefix->data, prefix_len) == 0) {
        PG_RETURN_BOOL(true);
    } else {
        PG_RETURN_BOOL(false);
    }
}
```

Two Kmer objects as input:
The **first (kmer)** is the sequence being checked.
The **second (prefix)** is the prefix being compared.

Get the length of both. If **the prefix > k-mer**, it cannot match, so it **immediately returns false**.

Compares the first prefix_len bytes of the k-mer data with the prefix data.

In dna_sequence--1.0.sql

```
CREATE OPERATOR ^@ (
    LEFTARG = kmer,
    RIGHTARG = kmer,
    PROCEDURE = starts_with,
    COMMUTATOR = @^,
    RESTRICT = scalarltsel,
    JOIN = scalarltjoinsel
);
```

COMMUTATOR: a ^@ b is equivalent to b @^ a

RESTRICT: Cost estimation -> WHERE

JOIN: Cost estimation -> JOIN

→ **Estimating selectivity**

Source: postgresql.org

CONTAINS FOR KMERS

In dna_sequence.h

```c
static int iupac_code_to_bits(char c) {
    switch (c) {
        case 'A': return 1; // A
        case 'C': return 2; // C
        case 'G': return 4; // G
        case 'T': return 8; // T
        case 'R': return 1|4; // A or G
        case 'Y': return 2|8; // C or T
        case 'S': return 2|4; // G or C
        case 'W': return 1|8; // A or T
        case 'K': return 4|8; // G or T
        case 'M': return 1|2; // A or C
        case 'B': return 2|4|8; // C or G or T
        case 'D': return 1|4|8; // A or G or T
        case 'H': return 1|2|8; // A or C or T
        case 'V': return 1|2|4; // A or C or G
        case 'N': return 1|2|4|8; // A or C or G or T
        default:
            ereport(ERROR,
                (/*...*/));
            return 0;
    }
}
```

```c
static int nucleotide_to_bits(char c){
    switch (c) {
        case 'A': return 1; // 0001
        case 'C': return 2; // 0010
        case 'G': return 4; // 0100
        case 'T': return 8; // 1000
        default:
            ereport(ERROR,
                (/*...*/);
            return 0;
    }
}
```

A → 0001 (1 in decimal)
C → 0010 (2 in decimal)
G → 0100 (4 in decimal)
T → 1000 (8 in decimal)

nucleotide_to_bits

Converts standard nucleotide characters (A, C, G, T) into a **unique bitwise representation**.

iupac_code_to_bits

Converts IUPAC nucleotide codes into a **bitwise representation**.

It function encodes the set using a **4-bit representation**.

Example: **R (A or G) → 1 | 4 = 5**

Compression implementation for the whole extension: **for a sequence of length n, the storage requirement would be n * 4 bits** (50% reduction compared to 8-bit ASCII). This idea has not been implemented.

CONTAINS FOR KMERS

In dna_sequence.c

```c
PG_FUNCTION_INFO_V1(contained_qkmer_kmer);
Datum contained_qkmer_kmer(PG_FUNCTION_ARGS) {
    /*...*/
    if (pattern_len != kmer_len) {
        PG_RETURN_BOOL(false);
    }
    /* For each pos., compare acc. to IUPAC codes */
    for (int i =0; i < pattern_len; i++) {
        char qc = toupper(pattern->data[i]);
        char kc = toupper(kmer->data[i]);

        int q_bits = iupac_code_to_bits(qc);
        int k_bits = nucleotide_to_bits(kc);

        if ((q_bits & k_bits) == 0) {
            PG_RETURN_BOOL(false);
        }
    }

    PG_RETURN_BOOL(true);
}
```

In dna_sequence--1.0.sql

```
CREATE OPERATOR @> (
    LEFTARG = qkmer,
    RIGHTARG = kmer,
    PROCEDURE = contains,
    COMMUTATOR = <@,
    RESTRICT = contsel,
    JOIN = contjoinsel
);
```

EXAMPLE

qkmer = 'RYG' **kmer = 'ACG'**

At **position 1**:
q_bits = 1 | 4 = 5 (A or G)
k_bits = 1 (A)
q_bits & k_bits = 5 & 1 = 1 (match)
…(pos 2 and pos 3)
The function **returns true** (all match).

contained_qkmer_kmer

If the **qkmer and kmer lengths don't match**, the function **immediately returns false**.

Every character **converted to uppercase**, same as in the other functions.

iupac_code_to_bits(qc) for qkmer chars
nucleotide_to_bits(kc) for kmer chars

Bitwise AND (&) between q_bits and k_bits

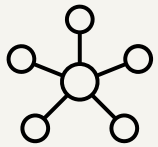
Both the **contains_qkmer_kmer(qkmer, kmer)** and the **contained_qkmer_kmer(kmer, qkmer)** functions are present in dna_sequence.c, that is why we can have **a commutator (<@)** for the operator. They are both also present in the operator class.

K-MER COUNTING SUPPORT

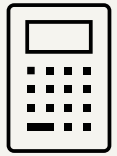
K-MER COUNTING SUPPORT

Custom Hash Function for Group By Operation of k-mer using equality operator

GROUP BY



COUNT



DISTINCT

CANONICAL

CANONICAL KMER

The canonical DNA of a k-mer is the **lexicographically smaller representation** between a DNA k-mer and its reverse complement (alphabetically, the one that comes first)

Reverse Complement:
> Reverse the sequence
> Replace each k-mer character with its complement: A ↔ T, C ↔ G

Example:
K-mer: GAT
Reverse complement: ATC
Canonical: ATC

DISTINCT OPERATION EXAMPLE

Simple SELECT query:

DISTINCT query:

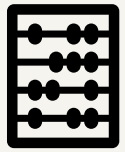
```
dna=# SELECT k.kmer
FROM generate_kmers('ATCGATCAC', 3) AS k(kmer);
 kmer
------
 ATC
 TCG
 CGA
 GAT
 ATC
 TCA
 CAC
(7 rows)
```

```
dna=# SELECT DISTINCT(k.kmer)
FROM generate_kmers('ATCGATCAC', 3) AS k(kmer);
 kmer
------
 CGA
 TCG
 TCA
 CAC
 ATC
 GAT
(6 rows)
```

EXAMPLE NORMAL & CANONICAL GROUP BY

GROUP-BY query:

```
dna=# SELECT k.kmer, count(*)
FROM generate_kmers('ATCGATCAC', 3) AS k(kmer)
GROUP BY k.kmer
ORDER BY count(*) DESC;
 kmer | count
------+-------
 ATC  |     2
 CGA  |     1
 TCG  |     1
 TCA  |     1
 CAC  |     1
 GAT  |     1
(6 rows)
```



Canonical GROUP-BY query:

```
dna=# SELECT canonical(k.kmer), count(*)
FROM generate_kmers('ATCGATCAC', 3) AS k(kmer
GROUP BY canonical(k.kmer)
ORDER BY count(*) DESC;
 canonical | count
-----------+-------
 ATC       |     3
 CGA       |     2
 TCA       |     1
 CAC       |     1
(4 rows)
```

HASH FUNCTION IN C

hash_any: a PostgreSQL utility function that computes a hash for a given array of characters.

unsigned char *: casting kmer sequence so that each **byte of a char** takes values of **0 to 255** instead of potentially negative numbers (-128 to 127).

PG_RETURN_UINT32(hash): Returns computed hash value as an integer to PostgreSQL

In dna_sequence.c

```c
/* Compute a hash value based on the contents
    of the Kmer data type */
PG_FUNCTION_INFO_V1(kmer_hash);
Datum kmer_hash(PG_FUNCTION_ARGS) {
    Kmer *kmer = (Kmer *) PG_GETARG_POINTER(0);
    int32 length = VARSIZE(kmer) - VARHDRSZ;

    /* Compute hash using PostgreSQL's hash_any function */
    uint32 hash = hash_any((unsigned char *)
                           kmer->data, length);

    PG_RETURN_UINT32(hash);
}
```

HASH OPERATOR

In dna_sequence—1.0.sql

```
-- Create hash function for kmer
CREATE FUNCTION kmer_hash(kmer) RETURNS integer
    AS 'MODULE_PATHNAME', 'kmer_hash'
    LANGUAGE C IMMUTABLE STRICT;

-- Register the kmer type as hashable
CREATE OPERATOR CLASS kmer_hash_ops DEFAULT FOR TYPE kmer
USING hash AS
    OPERATOR 1 = (kmer, kmer),
    FUNCTION 1 kmer_hash(kmer);
```

kmer_hash: defines a PostgreSQL function written in C, also called kmer_hash

kmer_hash_ops: registers kmer type as a hashable type in PostgreSQL

DEFAULT: set as the default hash operator class for kmer

OPERATOR 1 = (kmer, kmer): equality operator for hash-based indexing

FUNCTION 1 kmer_hash(kmer): Specifies kmer_hash function to compute the hash value.

CANONICAL FUNCTION LOOP

In dna_sequence.c

VARDATA(kmer): points to the actual k-mer string data

For loop: iterates over the input string in reverse order, calculating the reverse complement

reverse_complement[VARHDRSZ + i]: reverse complement string is stored starting from VARHDRSZ to accommodate the header, thus the header is skipped

```c
PG_FUNCTION_INFO_V1(canonical_kmer);
Datum canonical_kmer(PG_FUNCTION_ARGS) {
    Kmer *kmer = (Kmer *) PG_GETARG_POINTER(0);
    int32 len = VARSIZE(kmer) - VARHDRSZ;
    // Gets the actual string data
    char *input = VARDATA(kmer);
    // Allocate space for the reverse complement
    char *reverse_complement = palloc(len + VARHDRSZ);

    // iterates over the input string in reverse order
    for (int i = 0; i < len; i++) {
        char ch = input[len - 1 - i];
        switch (ch) {
            // Maps each kmer character to its complement A<->T, C<->G
            case 'A': reverse_complement[VARHDRSZ + i] = 'T'; break;
            case 'T': reverse_complement[VARHDRSZ + i] = 'A'; break;
            case 'C': reverse_complement[VARHDRSZ + i] = 'G'; break;
            case 'G': reverse_complement[VARHDRSZ + i] = 'C'; break;
            default:
            // if there is a character other than 'A', 'T',
            // 'C', or 'G', the function throws an error
                ereport(ERROR,
                        (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                         errmsg("Invalid character '%c' in kmer string", ch)));
        }
    }
```

INDEX STRUCTURE

SP-GiST

SP-GIST Core Functions (Insertions)

spg_kmer_config()

Initializes the SP-GiST index configuration.

spg_kmer_choose()

Decides where to insert a tuple.

spg_kmer_picksplit()

Groups leaf tuples into inner tuples and generates node labels.

Sample Set

ACG
ACGT
ACGC
ACGG
ACT
ATG

SP-GIST Core Functions (Insertions)

Sample Set

ACG
ACGT
ACGC
ACGG
ACT
ATG

SP-GiST

SP-GIST Core Functions (Insertions)

Sample Set

~~ACG~~
ACGT
ACGC
ACGG
ACT
ATG

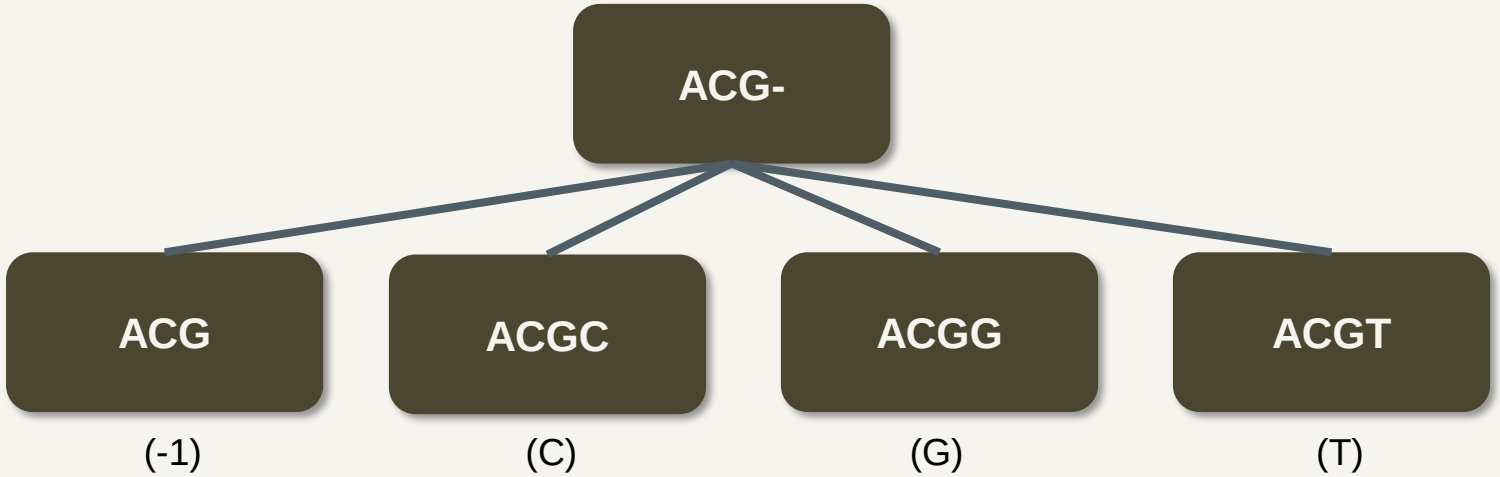
ACG-

SP-GiST

SP-GIST Core Functions (Insertions)

Sample Set

~~ACG~~
~~ACGT~~
~~ACGC~~
~~ACGG~~
ACT
ATG

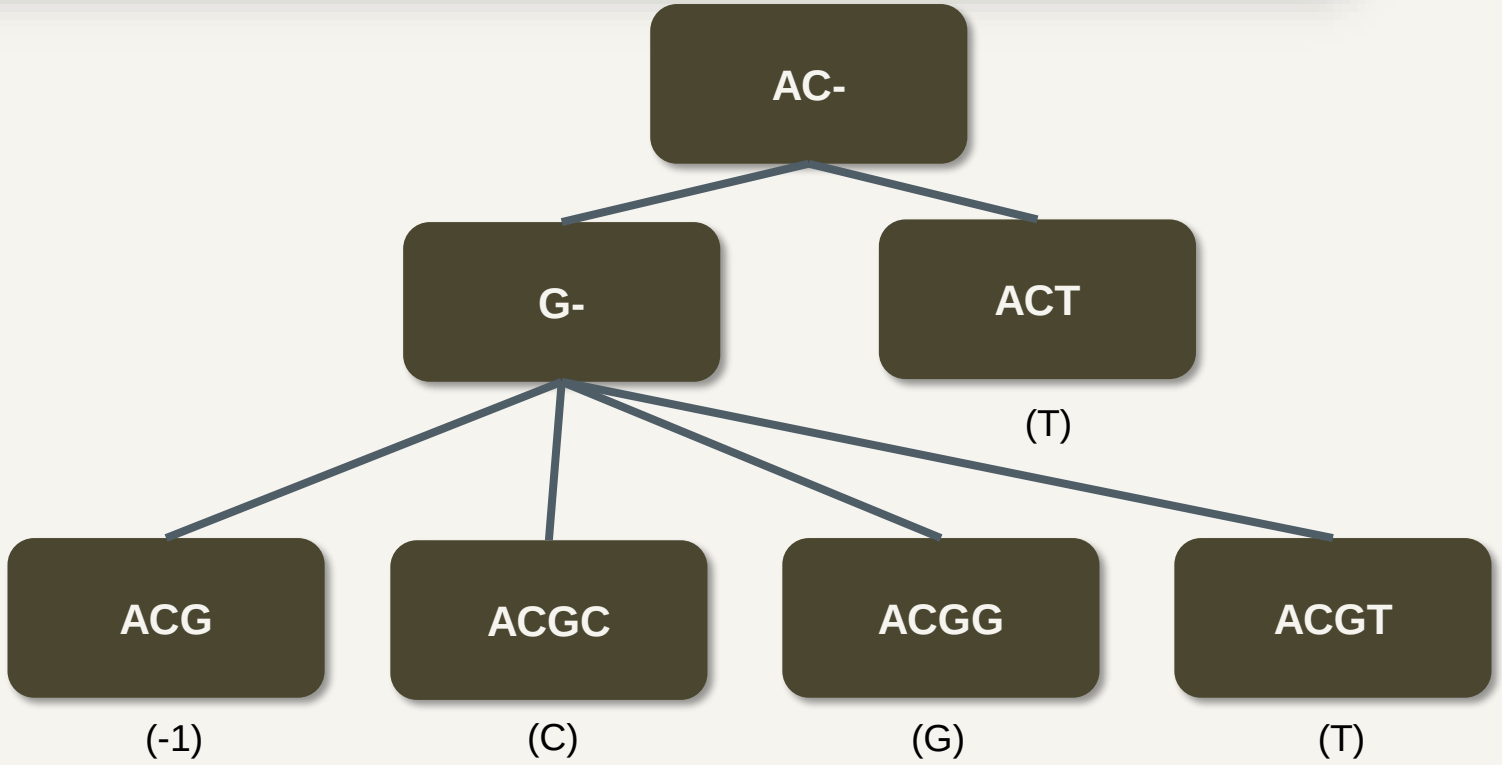


SP-GiST

SP-GIST Core Functions (Insertions)

Sample Set

~~ACG~~
~~ACGT~~
~~ACGC~~
~~ACGG~~
~~ACT~~
ATG

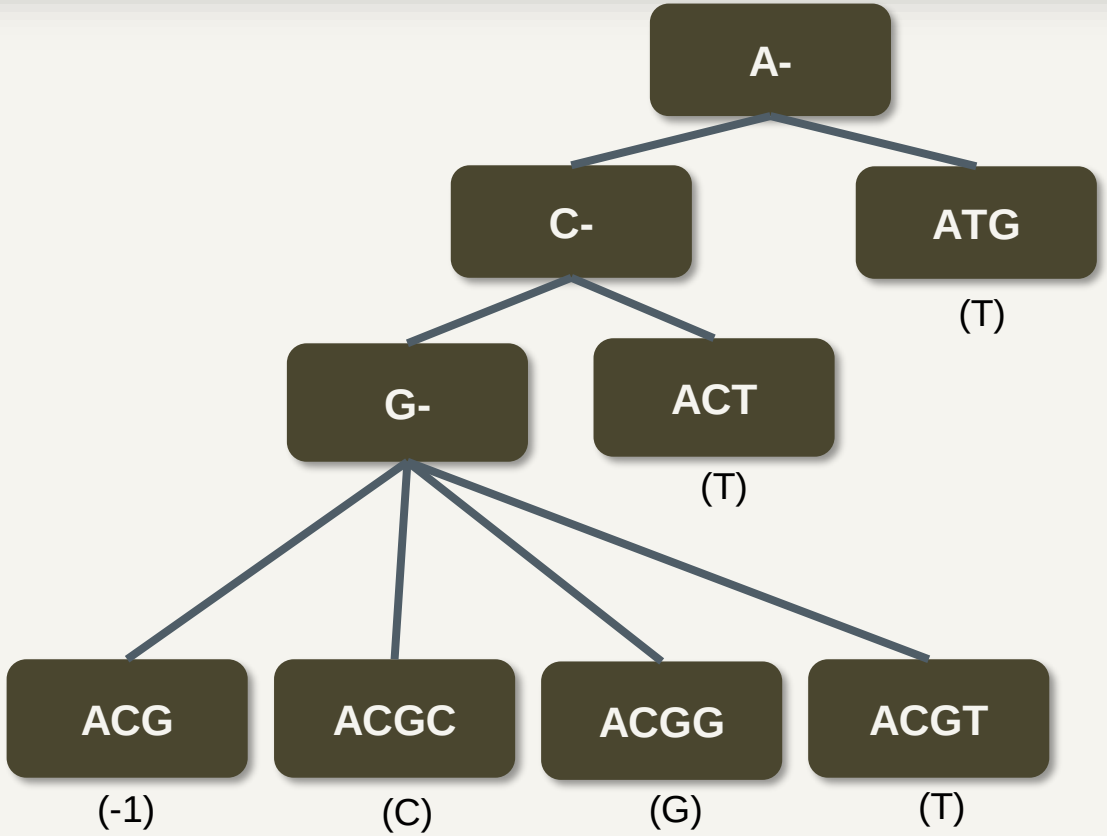


SP-GiST

SP-GIST Core Functions (Insertions)

Sample Set

~~ACG~~
~~ACGT~~
~~ACGC~~
~~ACGG~~
~~ACT~~
~~ATG~~



SP-GiST

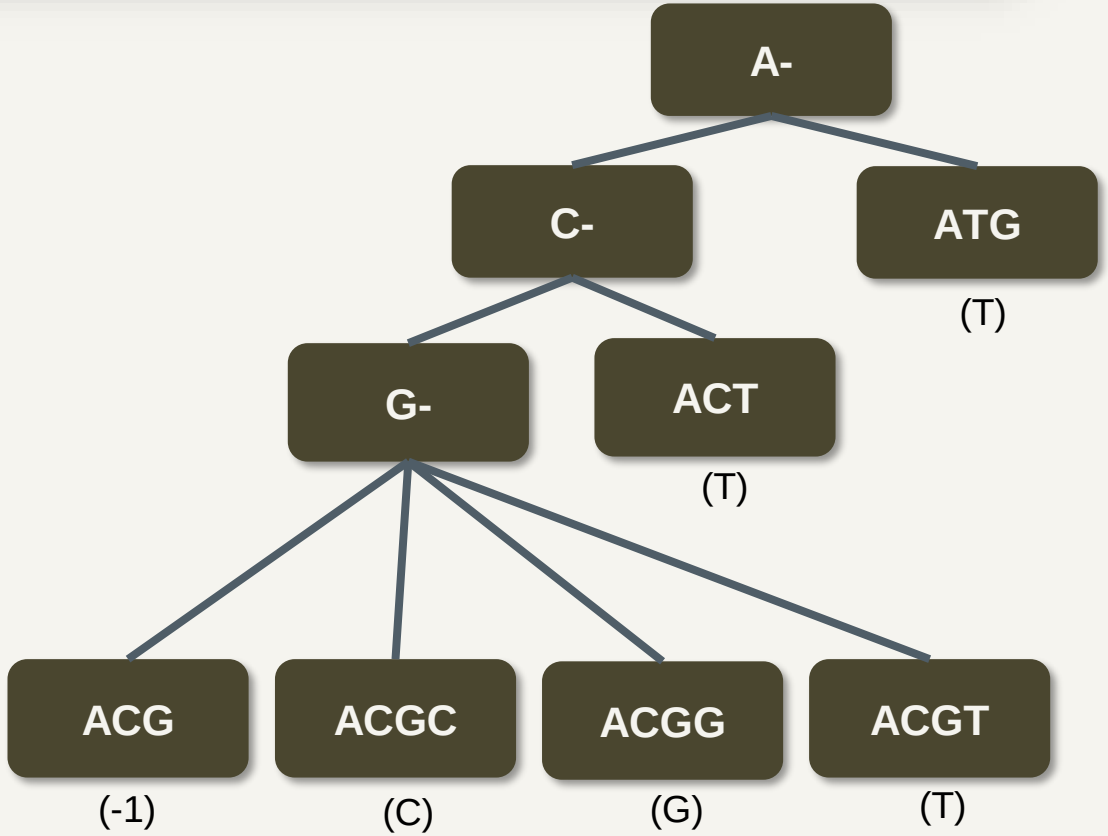
SP-GIST Core Functions (Query)

spg_kmer_inner_consistent()

Identifies the branches to follow during traversal.

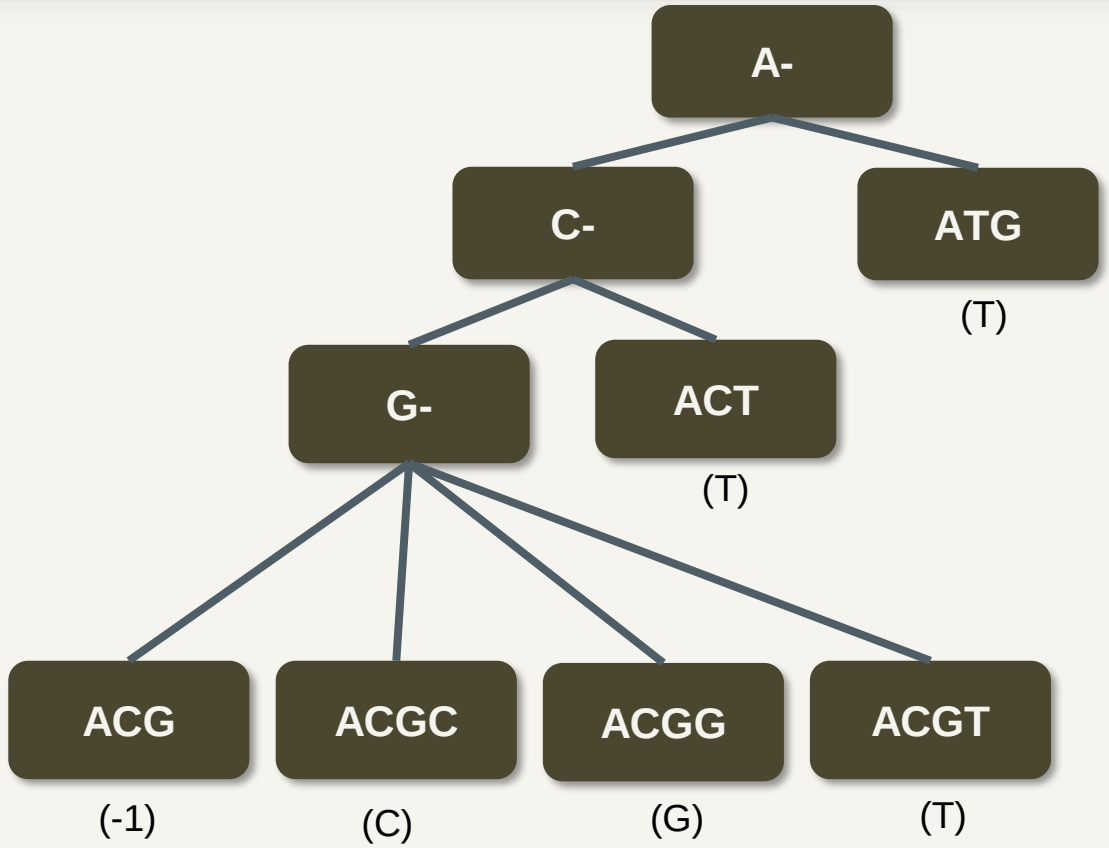
spg_kmer_leaf_consistent()

Verifies if a leaf tuple satisfies the query.



SP-GiST

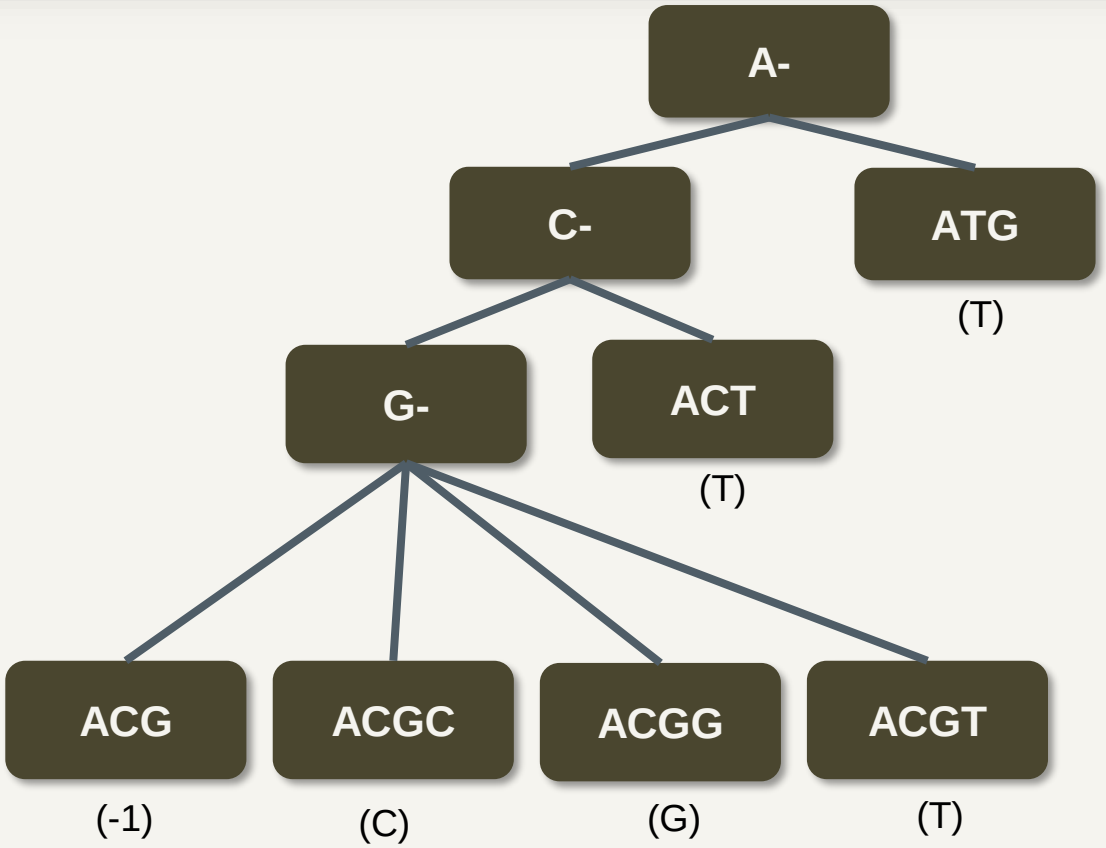
SP-GIST Core Functions (Query)



SP-GiST

SP-GIST Core Functions (Query)

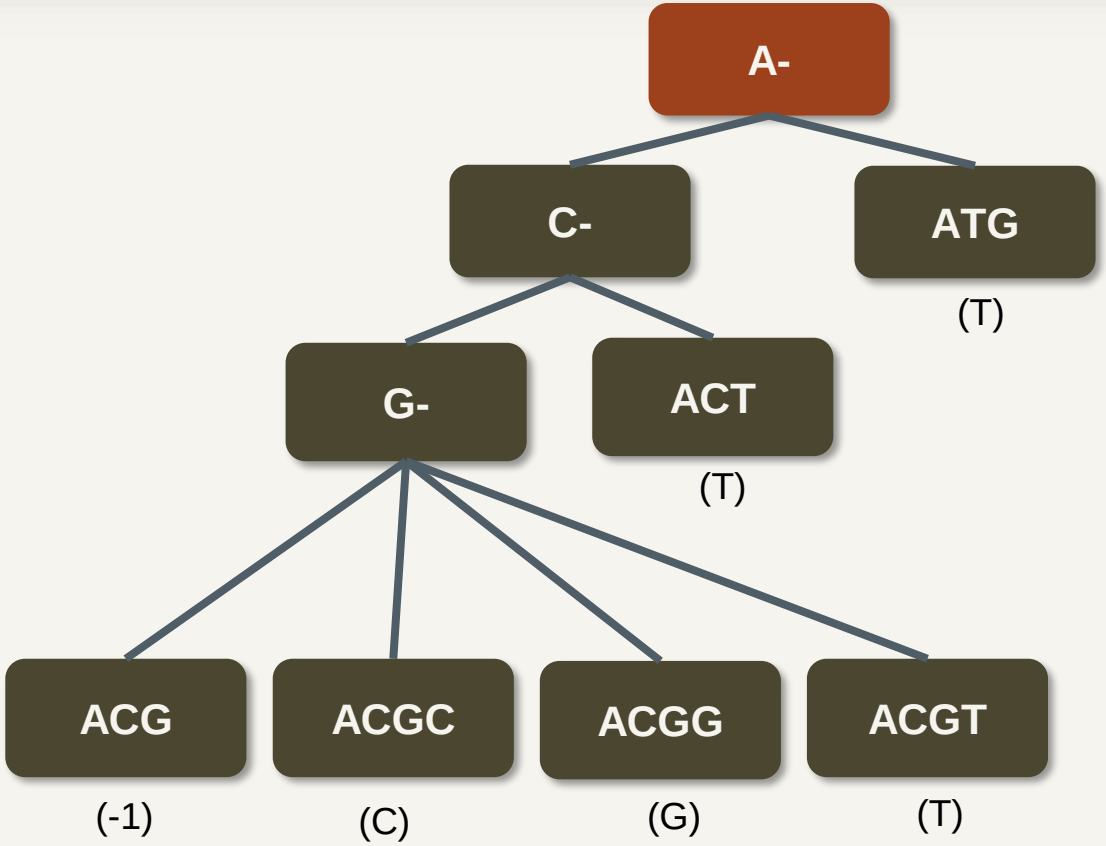
Equality Search for **ACT**



SP-GiST

SP-GIST Core Functions (Query)

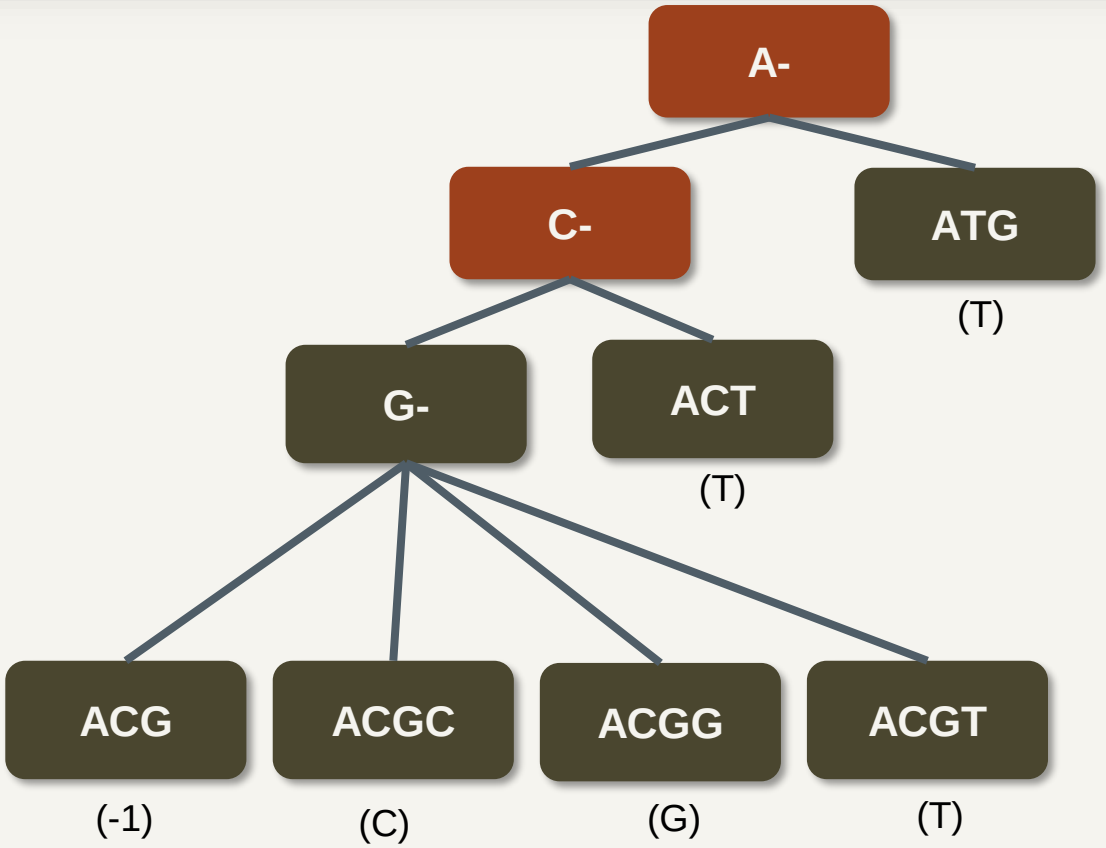
Equality Search for **ACT**



SP-GiST

SP-GIST Core Functions (Query)

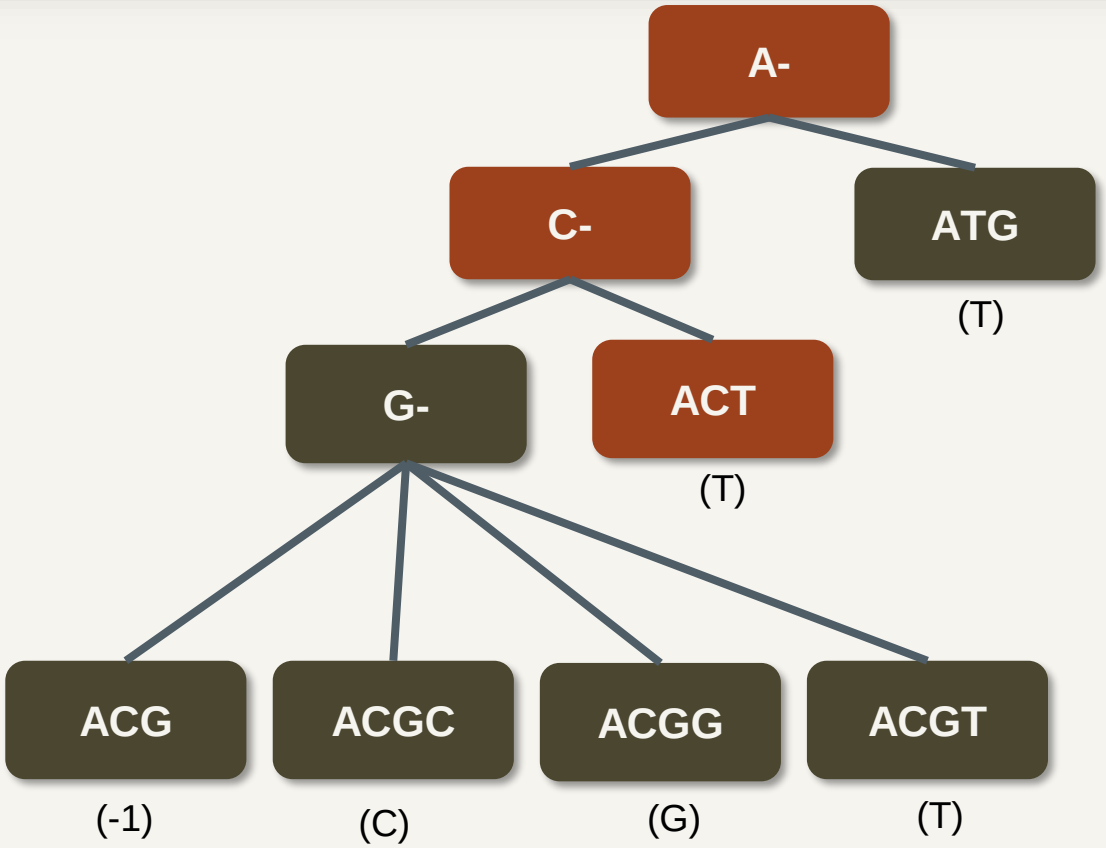
Equality Search for **ACT**



SP-GiST

SP-GIST Core Functions (Query)

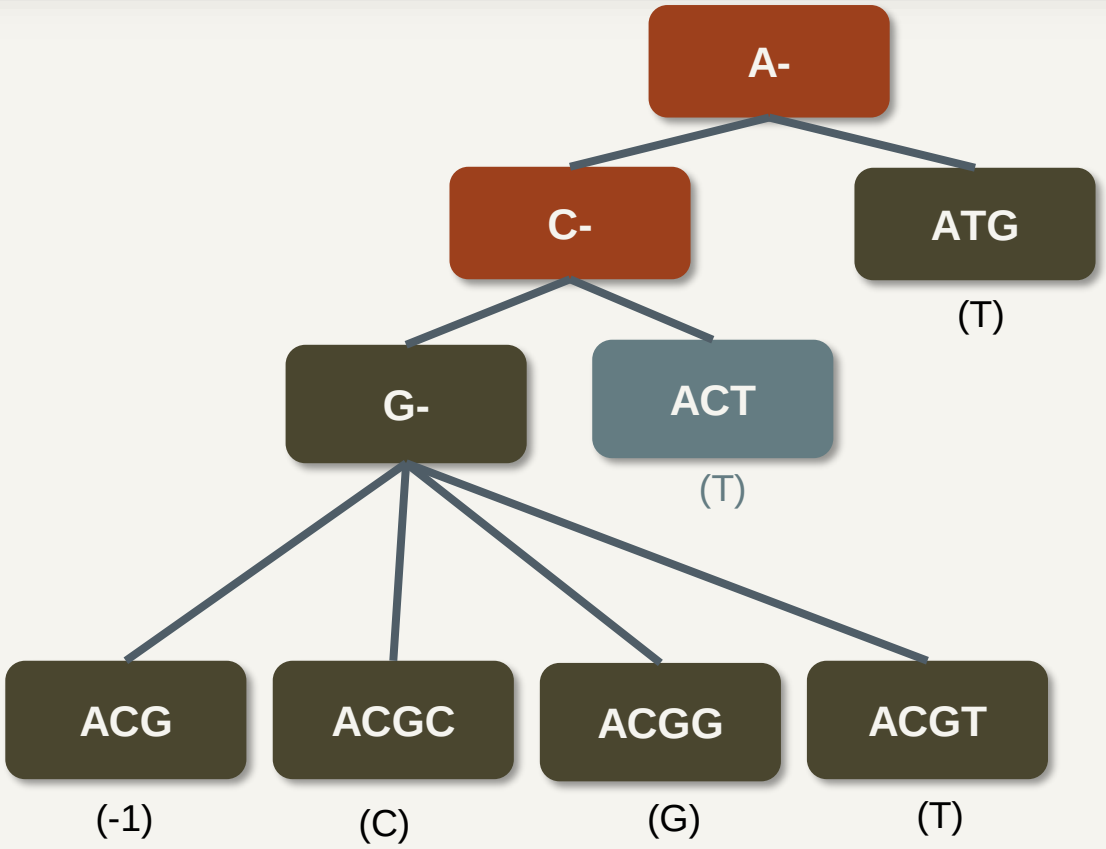
Equality Search for **ACT**



SP-GiST

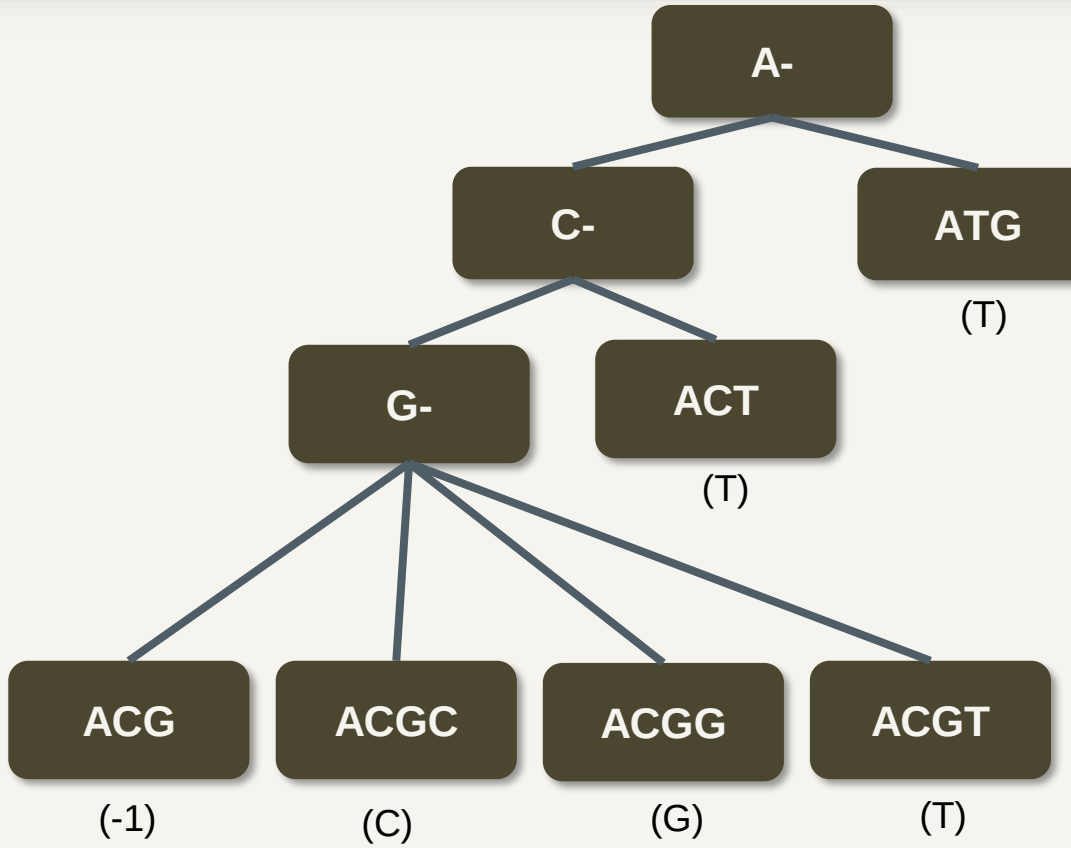
SP-GIST Core Functions (Query)

Equality Search for ACT



SP-GiST

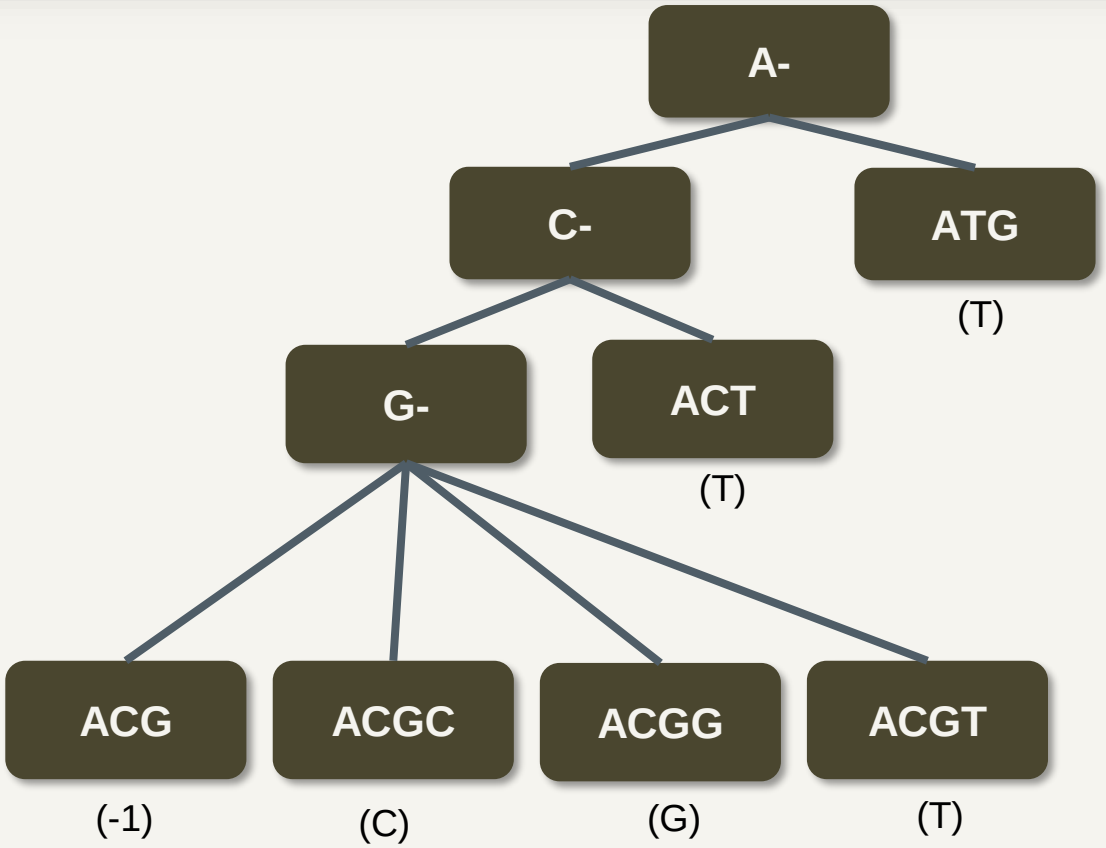
SP-GIST Core Functions (Query)



SP-GiST

SP-GIST Core Functions (Query)

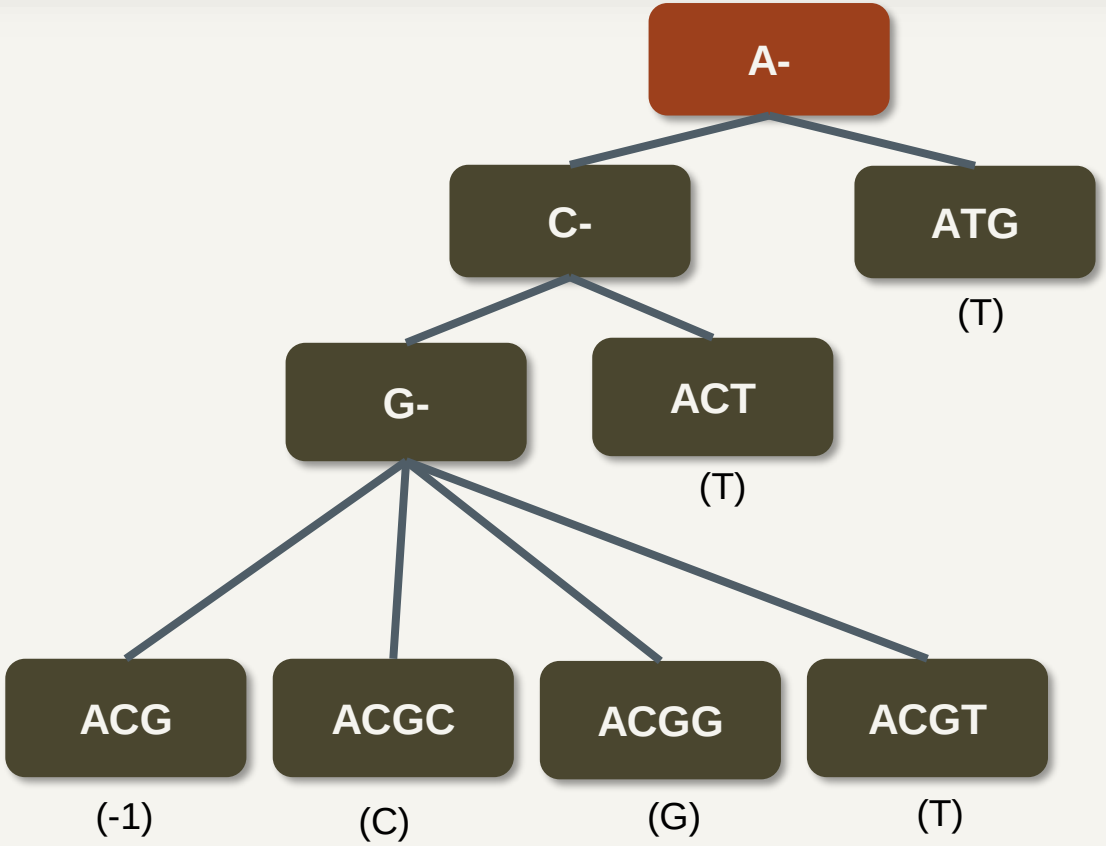
Prefix Search
for
ACG



SP-GiST

SP-GIST Core Functions (Query)

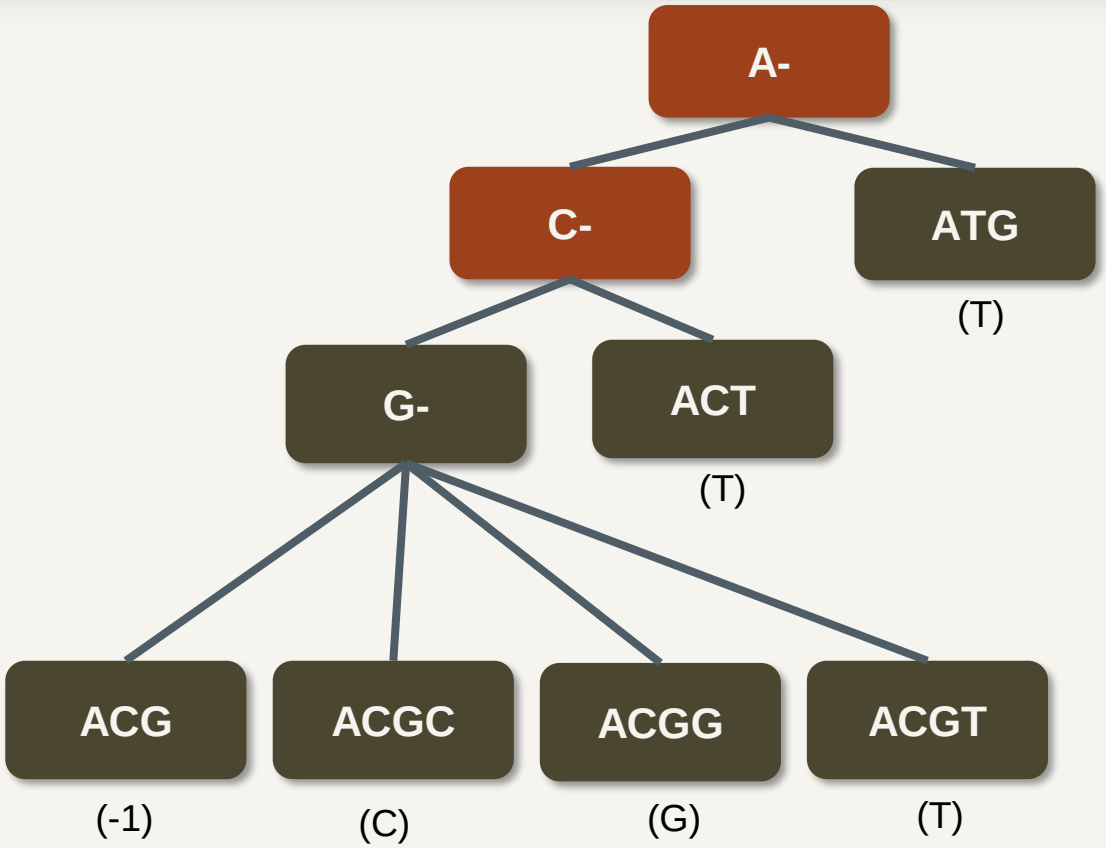
Prefix Search
for
ACG



SP-GiST

SP-GIST Core Functions (Query)

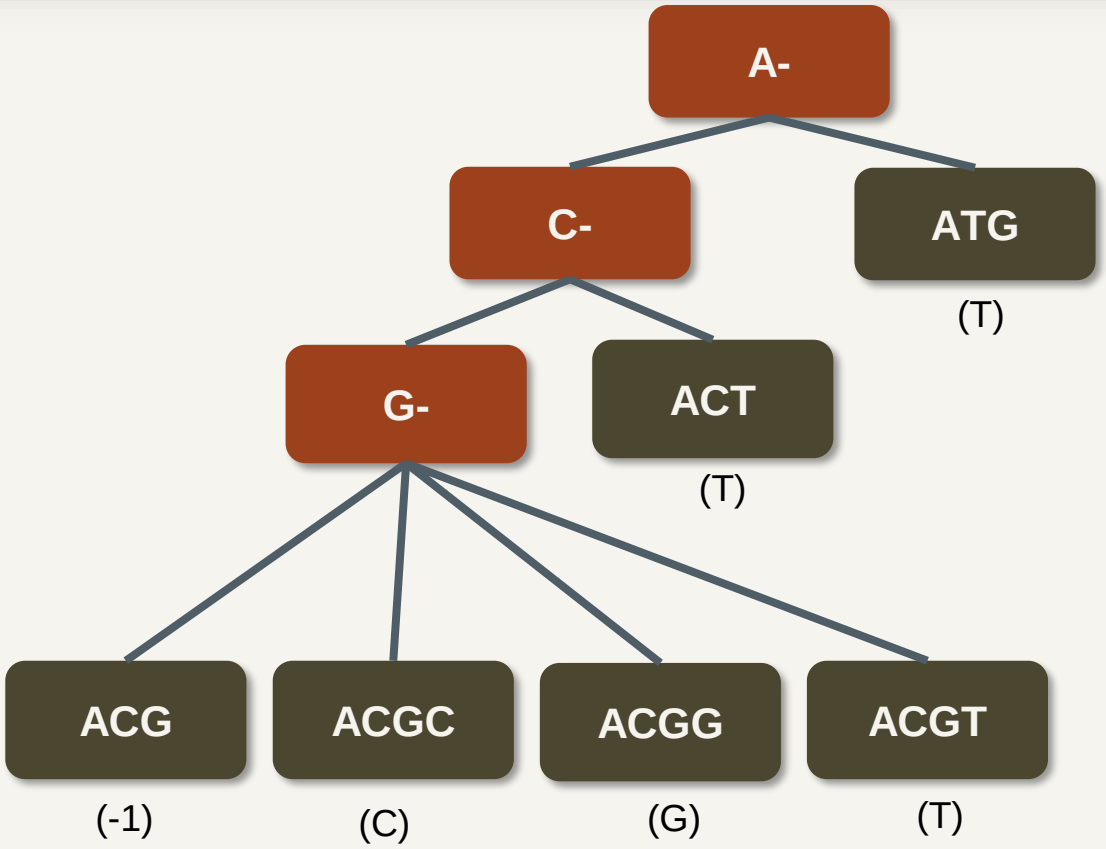
Prefix Search
for
ACG



SP-GiST

SP-GIST Core Functions (Query)

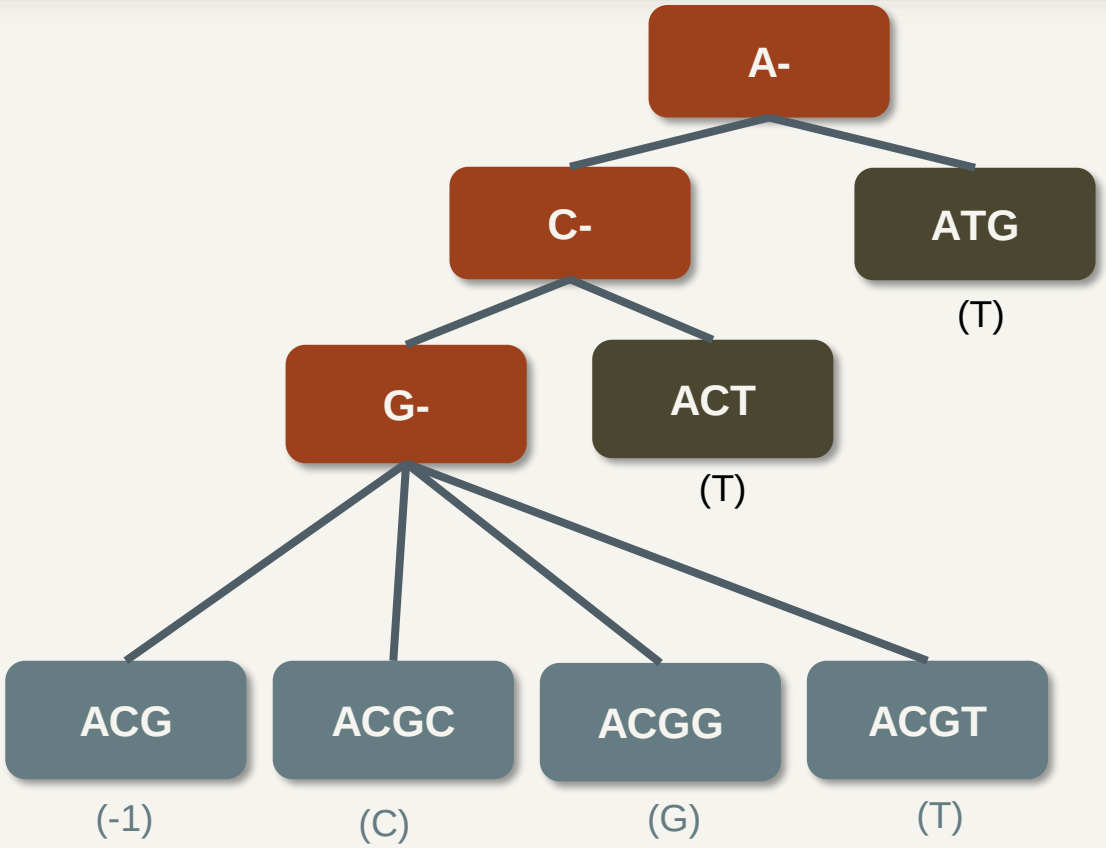
Prefix Search
for
ACG



SP-GiST

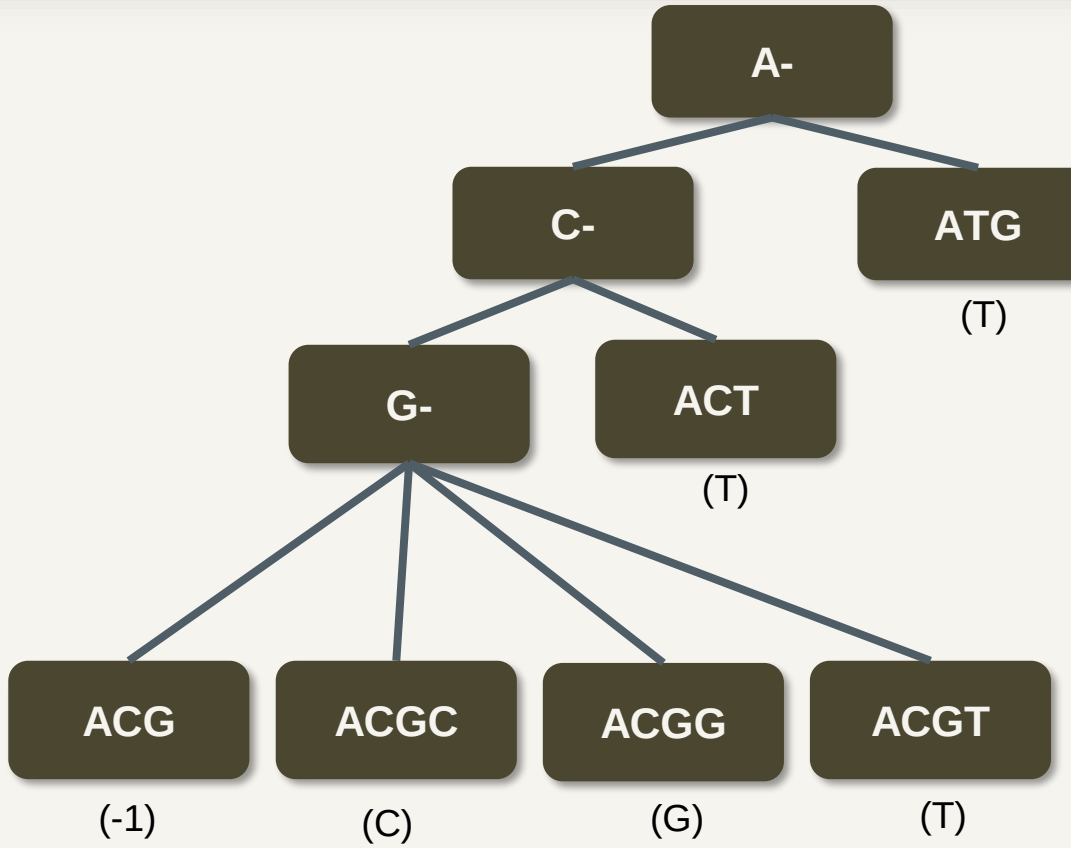
SP-GIST Core Functions (Query)

Prefix Search
for
ACG



SP-GiST

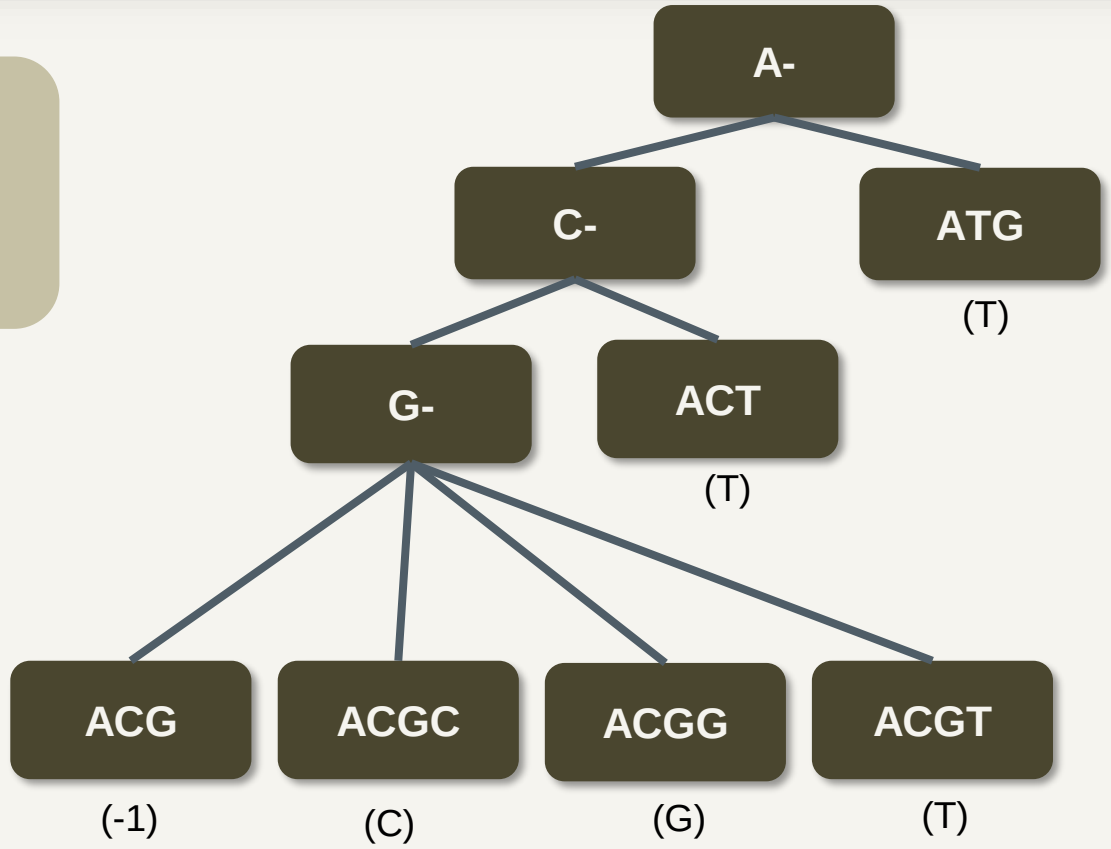
SP-GIST Core Functions (Query)



SP-GiST

SP-GIST Core Functions (Query)

Contains Search
Using IUPAC
for
qkmer = 'ACGS'

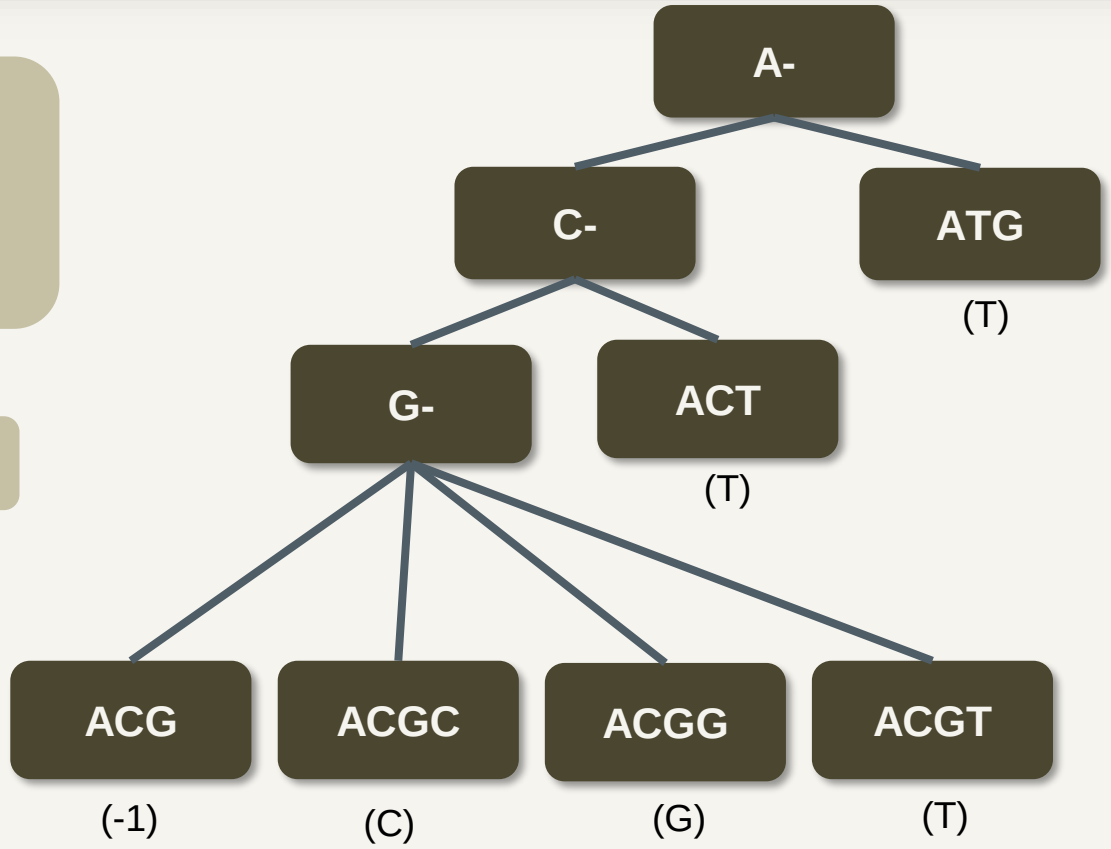


SP-GiST

SP-GIST Core Functions (Query)

Contains Search
Using IUPAC
for
qkmer = 'ACGS'

S means (C or G)

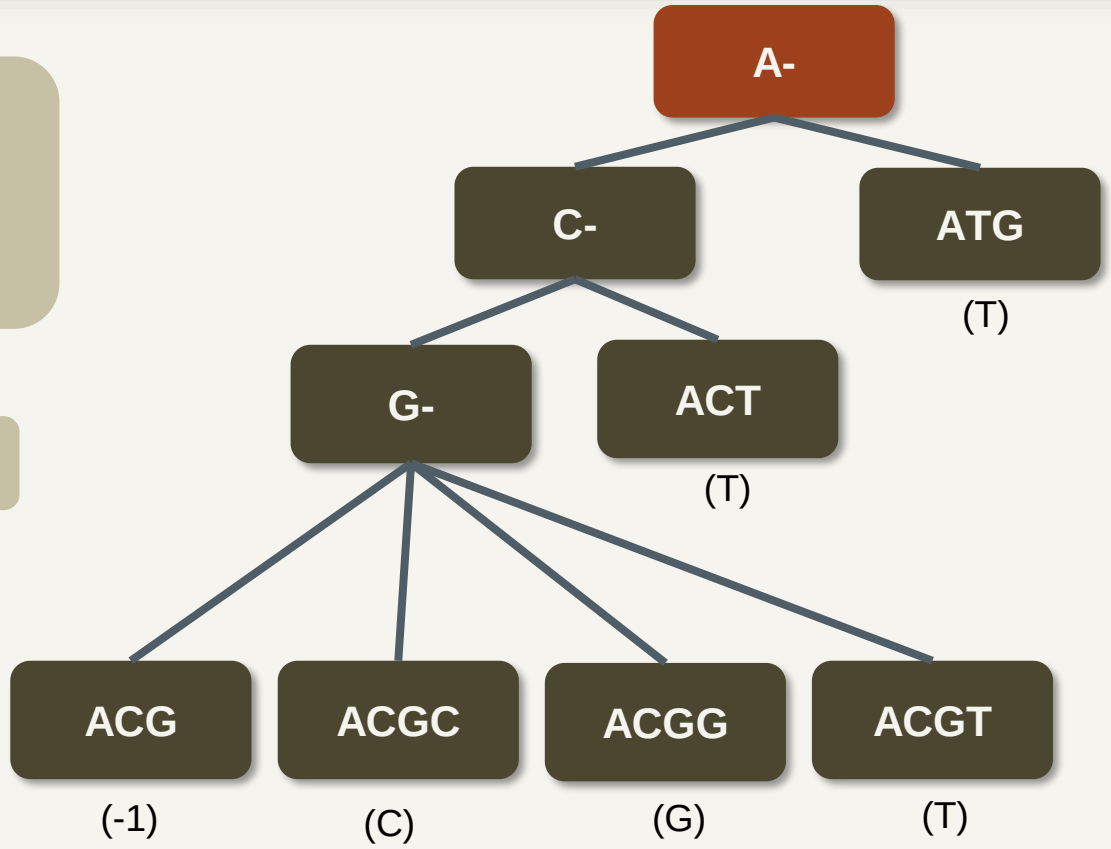


SP-GiST

SP-GIST Core Functions (Query)

Contains Search
Using IUPAC
for
qkmer = 'ACGS'

S means (C or G)

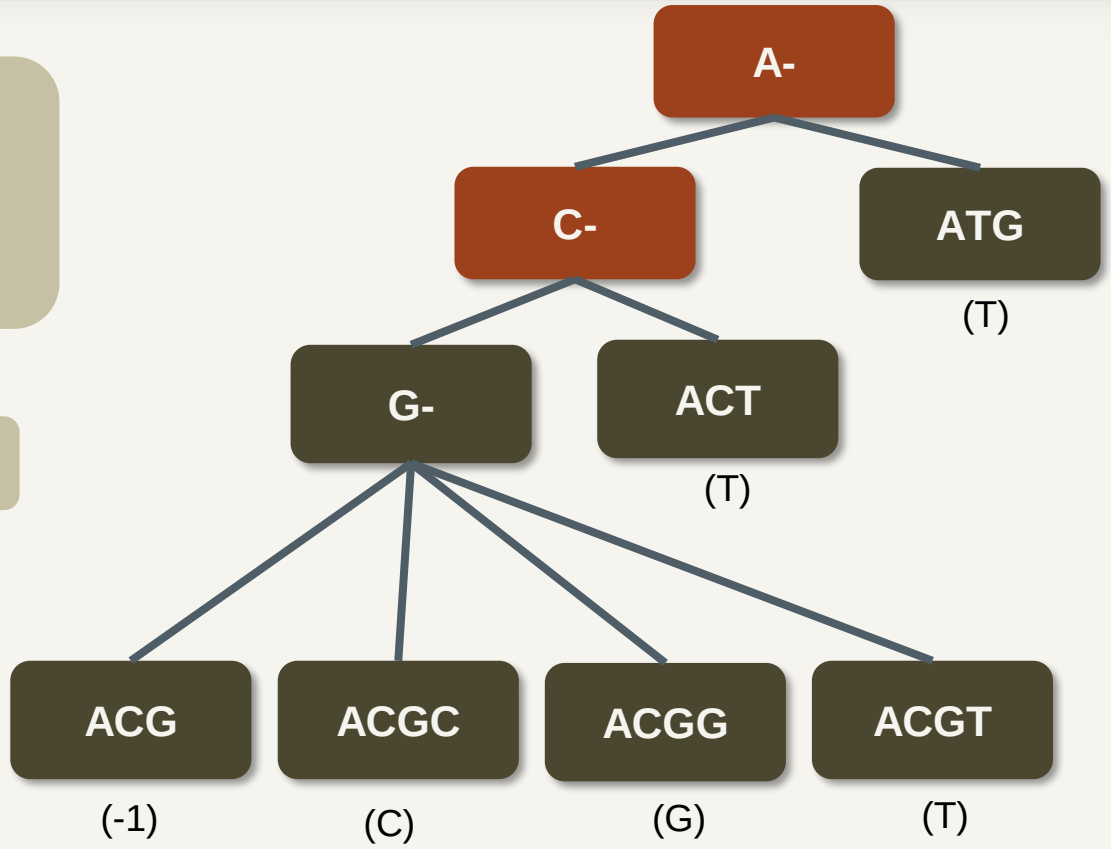


SP-GiST

SP-GIST Core Functions (Query)

Contains Search
Using IUPAC
for
qkmer = 'ACGS'

S means (C or G)

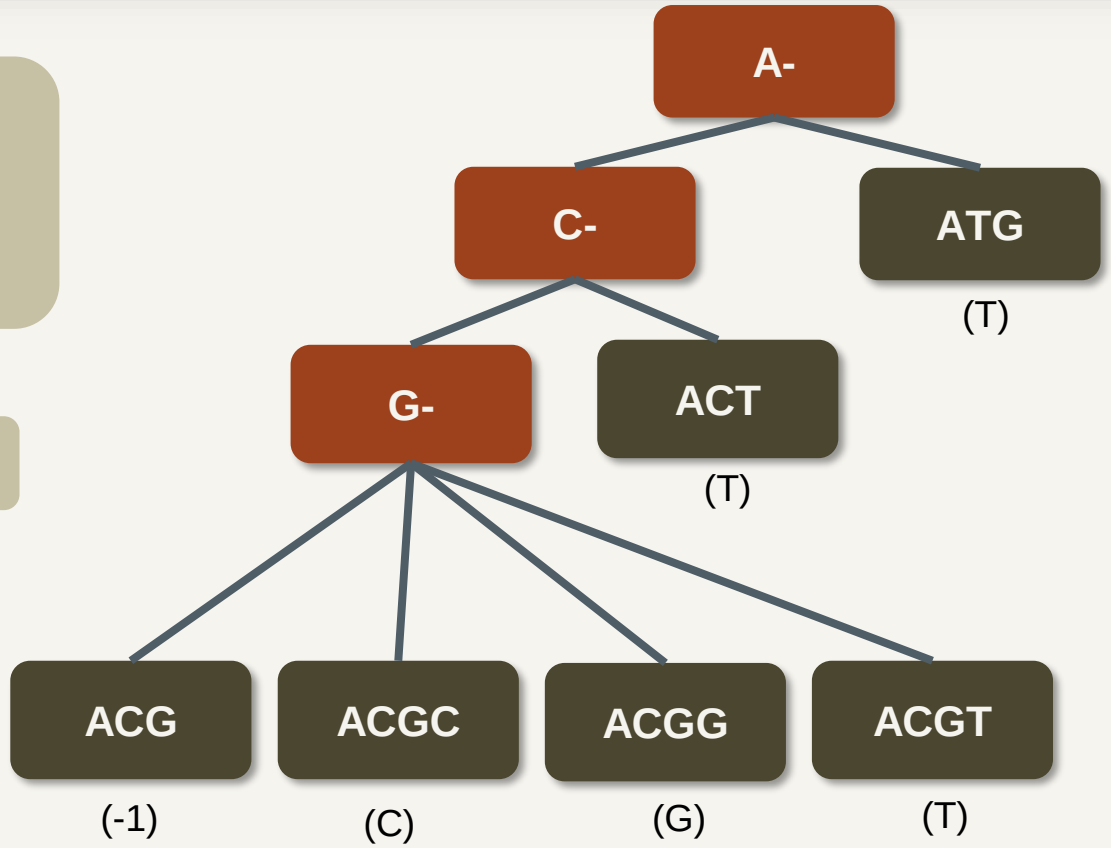


SP-GiST

SP-GIST Core Functions (Query)

Contains Search Using IUPAC for **qkmer = 'ACGS'**

S means (C or G)

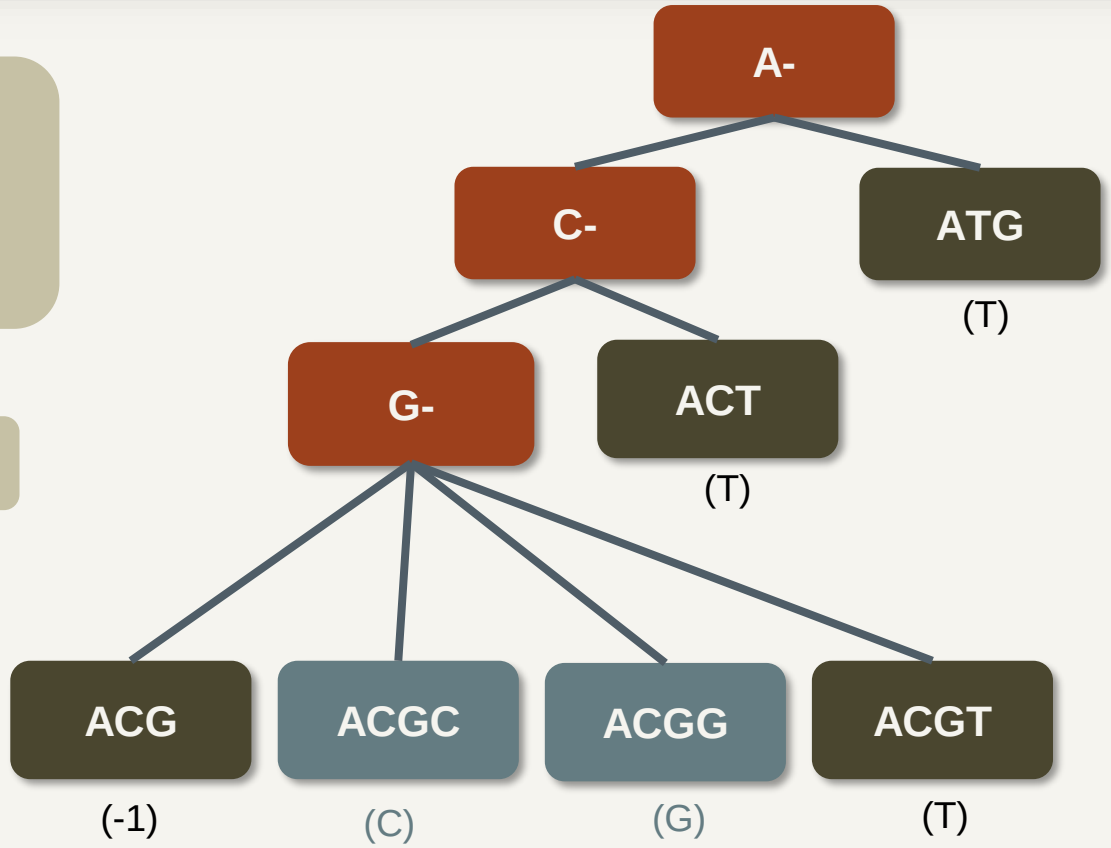


SP-GiST

SP-GIST Core Functions (Query)

Contains Search
Using IUPAC
for
qkmer = 'ACGS'

S means (C or G)



CHANGES IN SP-GiST FOR KMER (CONFIG)

```
Datum
spg_text_config(PG_FUNCTION_ARGS)
{
    /* spgConfigIn *cfgin = (spgConfigIn *) PG_GETARG_POINTER(0); */
    spgConfigOut *cfg = (spgConfigOut *) PG_GETARG_POINTER(1);

    cfg->prefixType = TEXTOID;
    cfg->labelType = INT2OID;
    cfg->canReturnData = true;
    cfg->longValuesOK = true;   /* suffixing will shorten long values */
    PG_RETURN_VOID();
}
 return go(f, seed, [])
}
```

```
Datum
spg_kmer_config(PG_FUNCTION_ARGS)
{
    spgConfigOut *cfg = (spgConfigOut *) PG_GETARG_POINTER(1);

    cfg->prefixType = get_kmer_oid();
    cfg->labelType = INT2OID;
    cfg->canReturnData = true; /* allow for reconstructon of original kmers */
    cfg->longValuesOK = false;   /* suffixing will shorten long values */

    /* picksplit can be applied to a single leaf tuple only in the case that the config function
     * set longValuesOK to true and a larger-than-a-page input value has been supplied. */
    PG_RETURN_VOID();
}
```

CHANGES IN SP-GiST FOR KMER (DATA TYPE)

Kmer data type is used instead of text

config() was accordingly updated to use custom kmer type as the prefix type.

```
Datum
spg_text_config(PG_FUNCTION_ARGS)
{
    /* spgConfigIn *cfgin = (spgConfigIn *) PG_GETARG_POINTER(0); */
    spgConfigOut *cfg = (spgConfigOut *) PG_GETARG_POINTER(1);

    cfg->prefixType = TEXTOID;
    cfg->labelType = INT2OID;
    cfg->canReturnData = true;
    cfg->longValuesOK = true;   /* suffixing will shorten long values */
    PG_RETURN_VOID();
}
 return go(f, seed, [])
}
```

```
Datum
spg_kmer_config(PG_FUNCTION_ARGS)
{
    spgConfigOut *cfg = (spgConfigOut *) PG_GETARG_POINTER(1);

    cfg->prefixType = get_kmer_oid();
    cfg->labelType = INT2OID;
    cfg->canReturnData = true; /* allow for reconstructon of original kmers */
    cfg->longValuesOK = false;   /* suffixing will shorten long values */

    /* picksplit can be applied to a single leaf tuple only in the case that the config function
     * set longValuesOK to true and a larger-than-a-page input value has been supplied. */
    PG_RETURN_VOID();
}
```

CHANGES IN SP-GiST FOR KMER (DATA TYPE)

```
Datum
spg_text_choose(PG_FUNCTION_ARGS)
{
  ...

  /* Check for prefix match, set nodeChar to first byte after prefix */
    if (in->hasPrefix)
    {
        text       *prefixText = DatumGetTextPP(in->prefixDatum);

        prefixStr = VARDATA_ANY(prefixText);
        prefixSize = VARSIZE_ANY_EXHDR(prefixText);

        commonLen = commonPrefix(inStr + in->level,
                                 prefixStr,
                                 inSize - in->level,
                                 prefixSize);

  ...
 }
```

```
Datum
spg_kmer_choose(PG_FUNCTION_ARGS)
{
  ...

   /* Check for prefix match, set nodeChar to first byte after prefix */
   if (in->hasPrefix)
   {
       Kmer       *prefixKmer = (Kmer *) DatumGetPointer(in->prefixDatum);

       prefixStr = VARDATA_ANY(prefixKmer);
       prefixSize = VARSIZE_ANY_EXHDR(prefixKmer);

       commonLen = commonPrefix(inStr + in->level,
                                prefixStr,
                                inSize - in->level,
                                prefixSize);

  ...

 }
```

CHANGES IN SP-GiST FOR KMER (DATA TYPE)

```
Datum
spg_text_choose(PG_FUNCTION_ARGS)
{
  ...

  /* Check for prefix match, set nodeChar to first byte after prefix */
    if (in->hasPrefix)
    {
        text       *prefixText = DatumGetTextPP(in->prefixDatum)

        prefixStr = VARDATA_ANY(prefixText);
        prefixSize = VARSIZE_ANY_EXHDR(prefixText);

        commonLen = commonPrefix(inStr + in->level,
                                 prefixStr,
                                 inSize - in->level,
                                 prefixSize);


  ...
 }
```

```
Datum
spg_kmer_choose(PG_FUNCTION_ARGS)
{
  ...

   /* Check for prefix match, set nodeChar to first byte after prefix */
   if (in->hasPrefix)
   {
       Kmer       *prefixKmer = (Kmer *) DatumGetPointer(in->prefixDatum);

       prefixStr = VARDATA_ANY(prefixKmer);
       prefixSize = VARSIZE_ANY_EXHDR(prefixKmer);

       commonLen = commonPrefix(inStr + in->level,
                                prefixStr,
                                inSize - in->level,
                                prefixSize);


  ...

 }
```

- The datum is now cast to Kmer * instead of text *.
- Subsequent lines that reference inText or prefixText now reference inKmer or prefixKmer.

CHANGES IN SP-GiST FOR KMER (STRATEGIES)

```
switch (strategy)
{
    case BTLessStrategyNumber:
    case BTLessEqualStrategyNumber:
    case BTEqualStrategyNumber:
    case BTGreaterEqualStrategyNumber:
    case BTGreaterStrategyNumber:
    case RTPrefixStrategyNumber:
        ...
}
```

```
switch (strategy)
{
    case KMER_EQUAL_STRATEGY:
        // Compare reconstructed kmer equality using memcmp
        break;
    case KMER_PREFIX_STRATEGY:
        // Check prefix condition (memcmp or kmer_starts_with)
        break;
    case KMER_CONTAINS_STRATEGY:
        // Check qkmer pattern match using iupac_code_to_bits() and nucleotide_to_bits()
        break;
    default:
        elog(ERROR, "unrecognized strategy number: %d", in->scankeys[j].sk_strategy);
}
```

CHANGES IN SP-GiST FOR KMER (STRATEGIES)

```
switch (strategy)
{
    case BTLessStrategyNumber:
    case BTLessEqualStrategyNumber:
    case BTEqualStrategyNumber:
    case BTGreaterEqualStrategyNumber:
    case BTGreaterStrategyNumber:
    case RTPrefixStrategyNumber:
        ...
}
```

```
switch (strategy)
{
    case KMER_EQUAL_STRATEGY:
        // Compare reconstructed kmer equality using memcmp
        break;
    case KMER_PREFIX_STRATEGY:
        // Check prefix condition (memcmp or kmer_starts_with)
        break;
    case KMER_CONTAINS_STRATEGY:
        // Check qkmer pattern match using iupac_code_to_bits() and nucleotide_to_bits()
        break;
    default:
        elog(ERROR, "unrecognized strategy number: %d", in->scankeys[j].sk_strategy);
}
```

- Strategies are changed from BT to SP-GiST, custom made for Kmer.
- In addition, bitwise operations are now implemented to determine if a given nucleotide matches a more general IUPAC-coded pattern character.

CHANGES IN SP-GiST FOR KMER (STRATEGIES)

```
switch (strategy)
{
    case BTLessStrategyNumber:
    case BTLessEqualStrategyNumber:
    case BTEqualStrategyNumber:
    case BTGreaterEqualStrategyNumber:
    case BTGreaterStrategyNumber:
    case RTPrefixStrategyNumber:
        ...
}
```

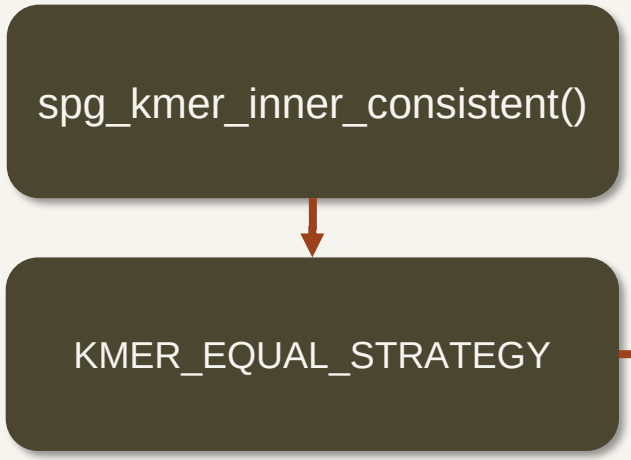
```
switch (strategy)
{
    case KMER_EQUAL_STRATEGY:
        // Compare reconstructed kmer equality using memcmp
        break;
    case KMER_PREFIX_STRATEGY:
        // Check prefix condition (memcmp or kmer_starts_with)
        break;
    case KMER_CONTAINS_STRATEGY:
        // Check qkmer pattern match using iupac_code_to_bits() and nucleotide_to_bits()
        break;
    default:
        elog(ERROR, "unrecognized strategy number: %d", in->scankeys[j].sk_strategy);
}
```

- Strategies are changed from BT to SP-GiST, custom made for Kmer.
- In addition, bitwise operations are now implemented to determine if a given nucleotide matches a more general IUPAC-coded pattern character.

CHANGES IN SP-GiST FOR KMER (STRATEGIES)

spg_kmer_inner_consistent()

CHANGES IN SP-GiST FOR KMER (STRATEGIES)




```c
/* KMER_EQUAL_STRATEGY:
 * Verifies that the reconstructed kmer segment exactly matches the query kmer.
 * Steps:
 * 1. Retrieve the query kmer and its length.
 * 2. Compare the reconstructed segment and the query kmer character-by-character.
 * 3. If there's any mismatch or the reconstructed segment is shorter than the query segment,
 *    mark this node as inconsistent (res = false).
 */

Kmer *inKmer = (Kmer *) DatumGetPointer(in->scankeys[j].sk_argument);
int   inSize = VARSIZE_ANY_EXHDR(inKmer);

int r = memcmp(VARDATA(reconstrKmer), VARDATA_ANY(inKmer), Min(inSize, thisLen));
if (r != 0 || inSize < thisLen)
    res = false;
```

In the inner consistency check, the **KMER_EQUAL_STRATEGY** ensures that the partially reconstructed kmer at the current node is not only a prefix but entirely equal to the query kmer up to this level. If the reconstructed segment doesn't match or is too short, we exclude this node from the search path.

CHANGES IN SP-GiST FOR KMER (STRATEGIES)

spg_kmer_inner_consistent()

KMER_PREFIX_STRATEGY

```
/* KMER_PREFIX_STRATEGY:
 * Ensures the reconstructed kmer segment is a valid prefix of the query kmer.
 * Steps:
 * 1. Retrieve the query kmer and its length.
 * 2. Compare the reconstructed segment to the start of the query kmer.
 * 3. If any character differs, this node is not consistent (res = false).
 */

Kmer *inKmer = (Kmer *) DatumGetPointer(in->scankeys[j].sk_argument);
int   inSize = VARSIZE_ANY_EXHDR(inKmer);

int r = memcmp(VARDATA(reconstrKmer), VARDATA_ANY(inKmer), Min(inSize, thisLen));
if (r != 0)
    res = false;
```

For the KMER_PREFIX_STRATEGY, we check if the reconstructed kmer segment so far can serve as a prefix of the query kmer. Any mismatch disqualifies this node from further search. If the partial sequence aligns perfectly with the start of the query kmer, the node is considered consistent.

CHANGES IN SP-GiST FOR KMER (STRATEGIES)

```c
/* KMER_CONTAINS_STRATEGY:
 * Checks the reconstructed kmer segment against a Qkmer pattern with ambiguous nucleotides.
 * Steps:
 * 1. Retrieve the Qkmer and ensure the reconstructed segment isn't longer than the pattern.
 * 2. For each character:
 *    - Convert both the qkmer character and the kmer character into bit patterns using
 *      iupac_code_to_bits() and nucleotide_to_bits().
 *    - If the bitwise AND is zero, it means no valid nucleotide overlap, so res = false.
 */

Qkmer *inQkmer = (Qkmer *) DatumGetPointer(in->scankeys[j].sk_argument);
int    qSize   = VARSIZE_ANY_EXHDR(inQkmer);

if (qSize < thisLen)
{
    res = false;
}
else
{
    char *qkmer_str = VARDATA_ANY(inQkmer);
    char *kmer_str  = VARDATA(reconstrKmer);

    for (int k = 0; k < thisLen; k++)
    {
        int q_bits = iupac_code_to_bits(qkmer_str[k]);
        int k_bits = nucleotide_to_bits(kmer_str[k]);

        if ((q_bits & k_bits) == 0)
        {
            res = false;
            break;
        }
    }
}
```

spg_kmer_inner_consistent()

KMER_CONTAINS_STRATEGY

The KMER_CONTAINS_STRATEGY involves matching against a Qkmer pattern that may represent multiple possible nucleotides at each position (via IUPAC codes). Here, we ensure that the reconstructed partial kmer is compatible with the qkmer pattern. Each position is checked bitwise, and if any position fails to match the allowed set of nucleotides, we discard this node.

INDEXING DEMO - OUTLINE

```
"Table Name"    "Index Name"    "Total Size"    "Total Size of all Indexes" "Table Size"    "Index Size"    "Estimated table row count"
"kmers" "kmers_spgist_idx"  "2840 MB"   "1253 MB"   "1586 MB"   "1253 MB"   29,355,896
```

Over **29 million unique kmers** generated using DNA data from the NCBI Sequence Read Archive (SRA).

Script using set with a **sliding_window (sequence, random(1, min(32, dna_length)))**, for over 1 million sequences.

Commands before each run

```
VACUUM ANALYZE kmers_big;
SET enable_bitmapscan = OFF;
SET max_parallel_workers_per_gather = 0;
```

query1.sql

```
EXPLAIN ANALYZE VERBOSE
SELECT * FROM kmers WHERE kmer <@ 'nttnn';
```

query2.sql

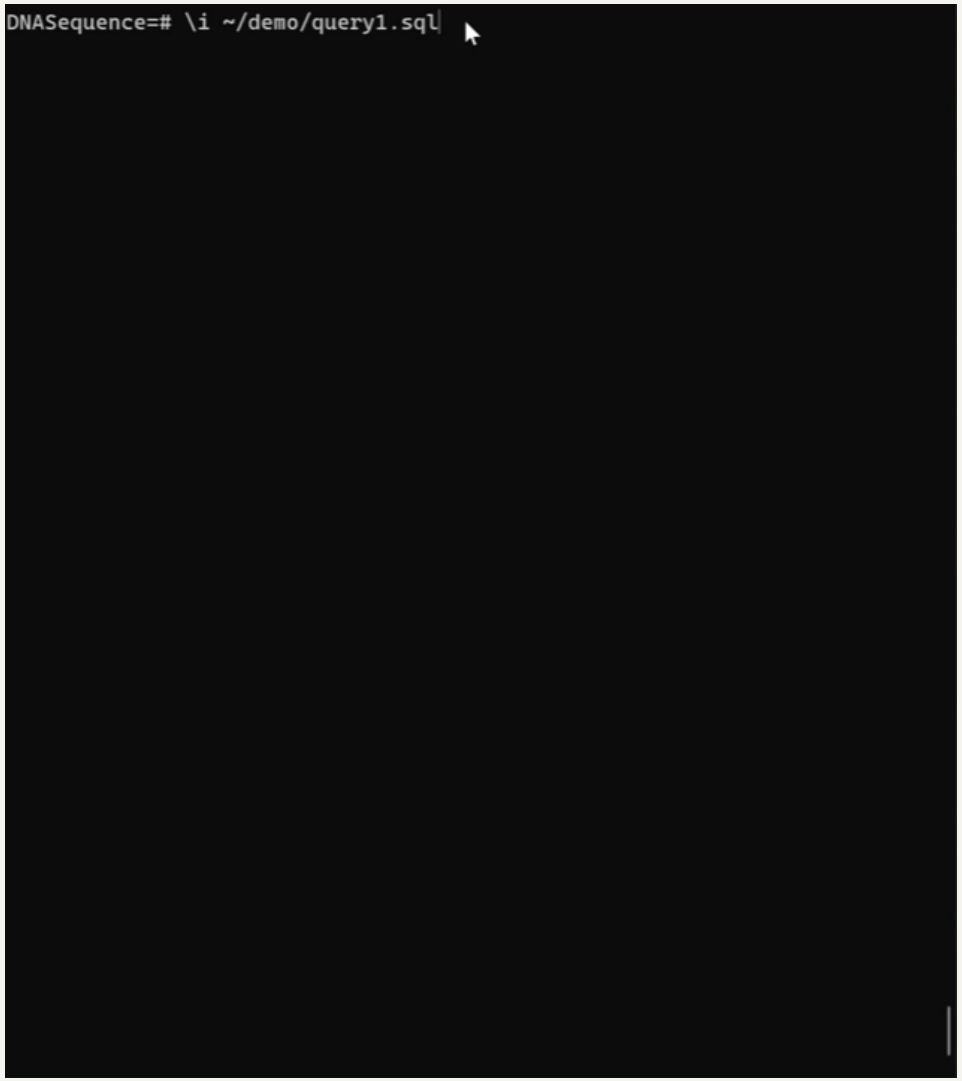
```
SELECT kmer FROM kmers WHERE kmer ^@ 'GCTAGCTAAGCT'
UNION
SELECT kmer FROM kmers WHERE kmer <@ 'CGTTAAGCTTGACCGT'
UNION
SELECT kmer FROM kmers WHERE kmer = 'GATTACAGCTAGCTTGA'
UNION
SELECT kmer FROM kmers WHERE kmer ^@ 'ACGTAGCTTGAACCGT' AND kmer <@ 'TGACCGTTGACTTGA'
UNION
SELECT kmer FROM kmers WHERE kmer ^@ 'GTACCGTTGACTGAA' AND kmer = 'CAGTGACTTGACGT'
UNION
SELECT kmer FROM kmers WHERE kmer <@ 'TACGTTGACTGAACTGACC' AND kmer = 'GCTACGTACGTTGAAC'
UNION
SELECT kmer FROM kmers WHERE kmer ^@ 'GCTAGCTTGAACCTGACT' AND kmer <@ 'GATTACAGCTAGCTAAGCTTGACCGT'
UNION
SELECT kmer FROM kmers WHERE kmer ^@ 'ACGTAGCTAAGCTTGAACCGTAC' AND kmer = 'TGACCGTTGACTTGAACCTGACG';
```


Source: postgresql.org; stackoverflow.com

INDEXING DEMO

SET enable_indexscan = ON;

SET enable_indexscan = OFF;



CONCLUSION

EXTENSION IMPLEMENTATION

Successful Implementation of the DNA Sequence Extension

Data Types

- All three required data types with their respective conditions regarding length and accepted characters.

Functions

- Basic data type functions (length, equivalence, etc.).
- K-mer counting and generation support
- QK-mer contain function

Index

- SPGiST for K-mer queries

Bonus

- **Canonical Function**
- **Negator for equality operator**

REAL WORLD APPLICATION: EXTENSION POSSIBILITIES

Using our extension IRL

- Can the extension provide what's needed for actual analysis?

Storage

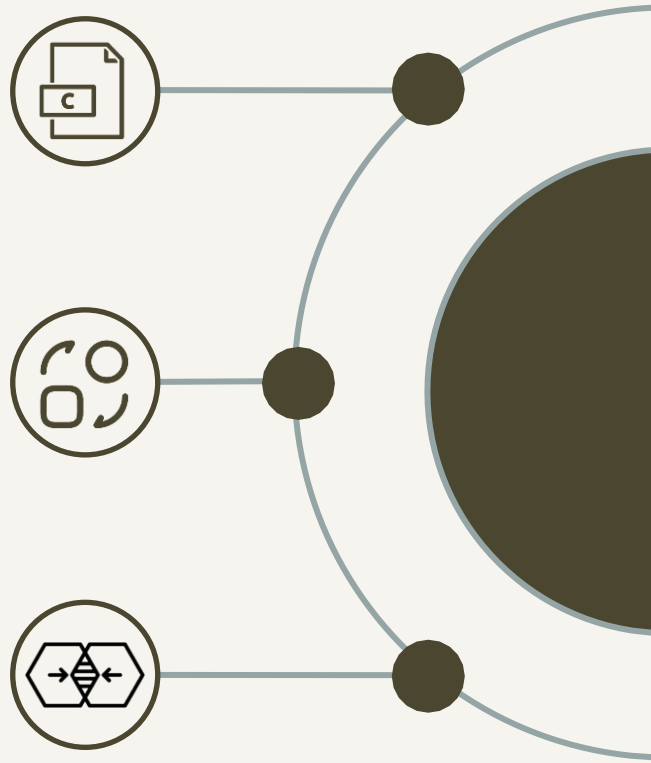
- Data types capable of handling long sequences.
- Compression was not used to avoid problems with indexing.

Processing

- Functional support for subsequence generation and comparison.
- K-mer generation function is key in shotgun sequencing applications.

Analysis

- Index implementation allows for data analysis of large amounts of subsequences.
- Can be utilized for sequence mining tasks yielding important insights.



Source: pmc.ncbi.nlm.nih.gov

DNA SEQUENCE DATABASE EXTENSION

our GitHub repository

Kristóf Balázs,
Stefanos Kypritidis
Otto Wantland
Nima Kamali Lassem

INFO-H417 Database System Architecture
2024