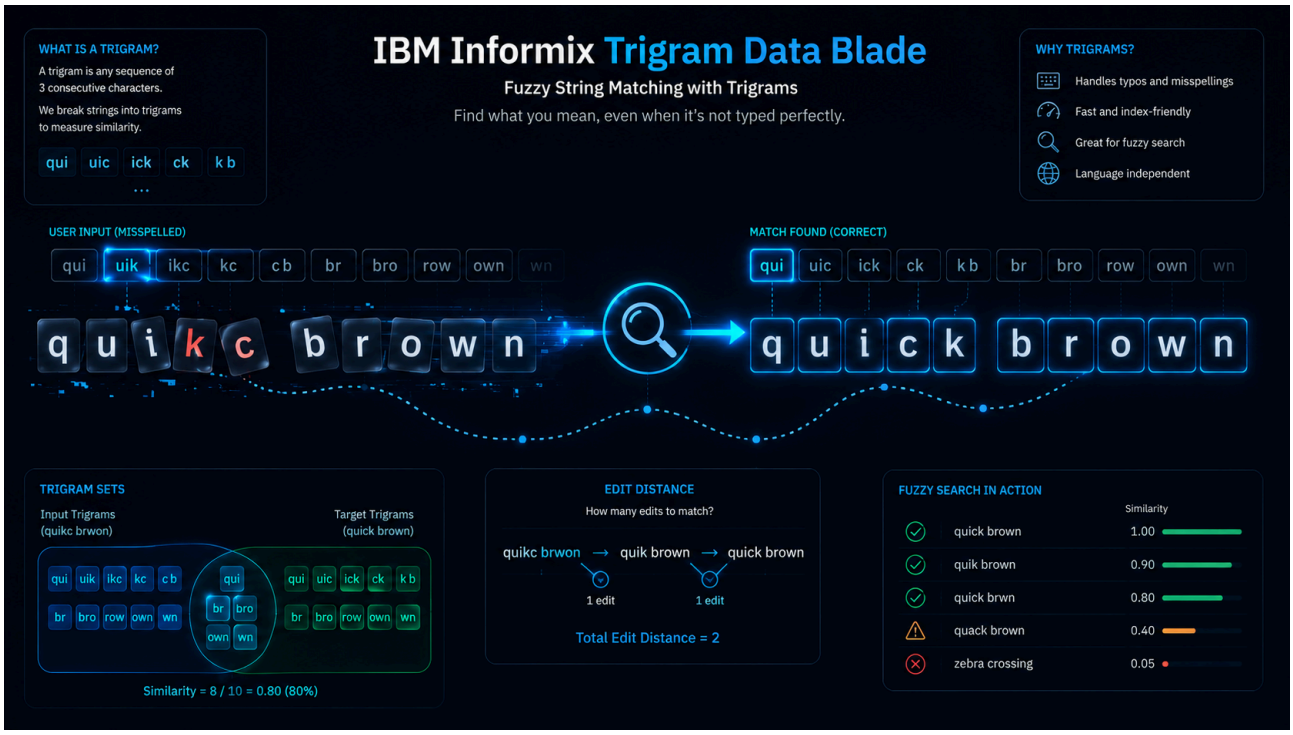
ifxtools

Document Version: 1.0 – 2026-08-13

Informix Trigram — Fuzzy Text Search Data Blade

Airtool

Content



This document describes the complete enterprise implementation of a **fuzzy text-search DataBlade** for IBM Informix — a set of C User-Defined Routines that bring typo-tolerant, accent-insensitive and phonetic string matching directly into the SQL engine, plus a trigram **inverted index** (a Virtual-Index Interface access method) that makes fuzzy search on millions of rows a sub-second candidate lookup instead of a sequential scan.

The design is deliberately **PostgreSQL-compatible**: it mirrors the SQL surface of the `pg_trgm`, `fuzzystrmatch` and `unaccent` extensions (`similarity`, `word_similarity`, `show_trgm`, `levenshtein`, `soundex`, `difference`, ...), so the identical query runs on Informix and PostgreSQL. Accent folding is **built in**: every argument is folded to ASCII before matching, so `'lopes'` finds `'López'` with no extra wrapper.

> **Verified against PostgreSQL as the reference implementation.** On native x86 servers, an exhaustive per-row harness (Informix = PostgreSQL) asserts **122,000 comparisons with 0 mismatches** for `similarity`, `levenshtein`, `soundex` and `difference`; a European-script battery confirms our built-in accent folding equals Postgres `similarity(unaccent(a),unaccent(b))` across 16 name/word pairs with **0 mismatches**. The VTI trigram index (`CREATE INDEX ... USING trgm_am`), measured on Informix 15: at 50,000 rows a sequential scan is **~85 ms/query** and the routed index query **~5 ms** (**~17×**), results identical to the full scan; an earlier SQL-materialized prototype of the same design scaled to 1,000,000 rows (2,508 → 332 ms).

1. Motivation — Fuzzy Text Search in Enterprise Data Platforms

Exact predicates — `=`, `LIKE` — fail the moment real-world data or a user's query carries a typo, a spelling variant, a transposition, an accent, or a phonetic difference. A customer stored as **López** is invisible to a clerk who types **lopes**; **Smith** and **Smyth** never join; **François** and **Francois** are two people. In master-data management, KYC/AML screening, deduplication, product search and call-centre lookup, this gap is not cosmetic — it is missed matches, duplicate records and failed compliance checks.

Modern search workloads have shifted from **exact** matching to **tolerant** matching. Applications are now expected to absorb typographical errors, OCR noise, user misspellings, partial names and foreign spellings without hand-written logic — a search for `Jonhson` should surface **Johnson**, **Johnston** and **Johansson**, ranked by confidence. PostgreSQL's `pg_trgm` extension has become the de-facto reference for this, and it has set developer expectations: engineers now assume they can write `WHERE similarity(name,'jon') > 0.4` or `ORDER BY similarity(name,'jon') DESC` and get an index-accelerated, relevance-ranked answer directly in SQL — no external logic.

Informix already ships **Basic Text Search (BTS)**, a capable full-text engine: Boolean queries, phrase search, stemming and synonym dictionaries. BTS is strong at `traditional information retrieval` over documents — but it is oriented to that task, not to `approximate string matching`. Delivering typo tolerance, similarity scoring or phonetic lookup on top of BTS today means custom preprocessing, multiple indexes, external phonetic algorithms and application-side ranking. The search logic ends up **outside** the database — precisely where it is hardest to secure, tune and keep consistent with the data.

This blade closes that gap **natively**. It brings the full `pg_trgm`-class capability set — trigram similarity and scoring, an index-accelerated similarity operator, Levenshtein / Damerau-Levenshtein and Jaro-Winkler edit distance, Soundex phonetics, ranked fuzzy results and **built-in accent folding** for accented and multilingual data (**José**, **Jose**, **Jóse**, **Josef** all match) — into the Informix SQL engine itself. It does not replace BTS; it **complements** it: BTS keeps serving document retrieval, while this blade adds the approximate-matching layer BTS lacks. Together they cover both exact-document search and typo-tolerant, relevance-ranked fuzzy search.

Performing the match **inside the database** — rather than in a bolt-on search engine such as Elasticsearch or Solr — avoids a second datastore to operate, secure and keep in sync, and keeps fuzzy matching consistent with the transactional source of truth. It also enables **hybrid queries** that combine fuzzy scoring with ordinary SQL filters (tenant, status, date range, permissions) in a single transactional statement. Commercially, the payoff is **competitive parity**: PostgreSQL set the expectation for in-database fuzzy search, and this blade lets Informix meet it for customer-data, product-catalogue, name-screening and multilingual-content applications — while reducing application complexity and eliminating external search infrastructure.

> **The commercial thesis in one line.** Informix BTS answers `“find this document”`; this blade answers `“find the row that was probably meant”` — typo-tolerant, accent-insensitive, phonetically aware and relevance-ranked, in native SQL, with no external search engine to run.

1.1. System architecture

The blade sits entirely inside the Informix server as two shared objects: the scalar functions (`trgm.so`) and the inverted-index access method (`trgm_index.so`). Applications issue ordinary SQL.

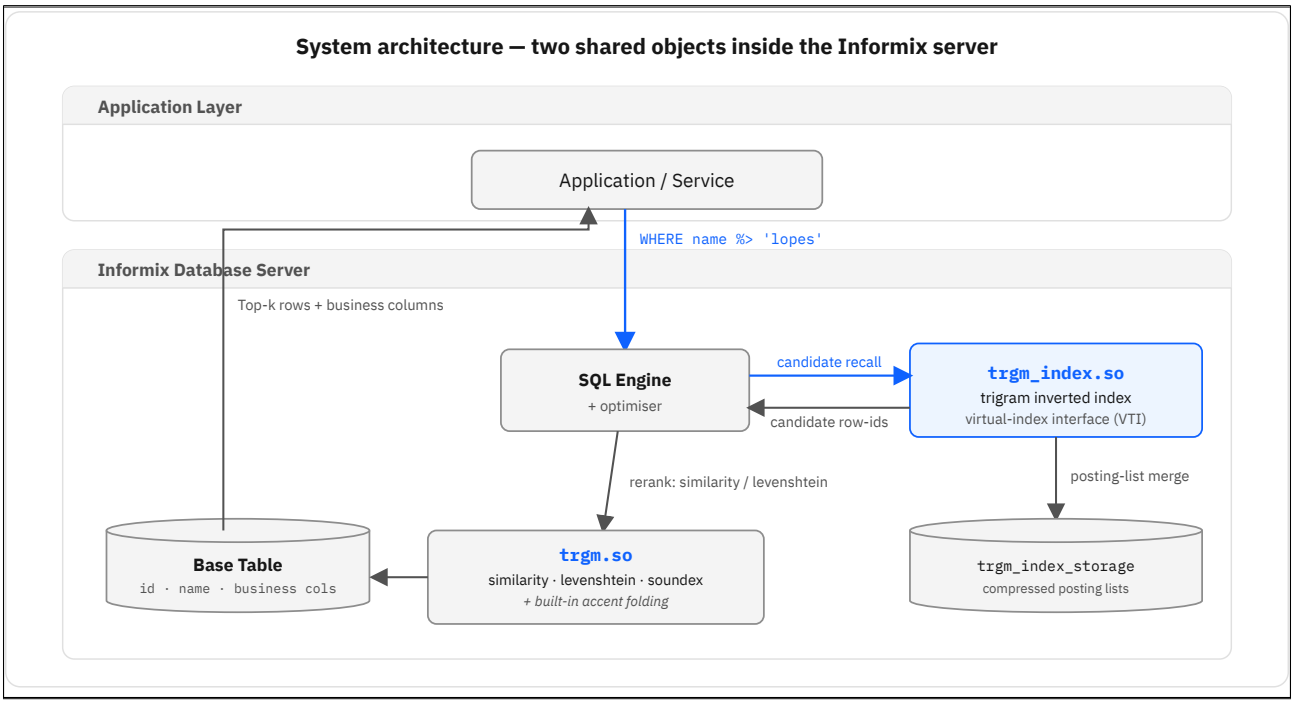


Figure 1 — System architecture — two shared objects inside the Informix server

2. The Matching Algorithms

Fuzzy matching is not one algorithm but a toolbox: different measures capture different kinds of "closeness". This section details each function implemented by the blade, with the theory, an SQL example, and the enterprise use case it fits. Every example runs against the single **worked dataset** defined immediately below, and **every result shown is the actual output captured from the blade running on Informix 15** (and verified equal to PostgreSQL where a reference exists).

Normalization — read this once. Every function **folds accents first** (see `Accent Folding`), so all examples are accent-insensitive (`José # Jose`). The trigram and phonetic functions (`similarity`, `word_similarity`, `show_trgm`, `soundex`, `difference`) **additionally lower-case**, so they are also case-insensitive. The edit-distance functions (`levenshtein`, `damerau_levenshtein`, `jaro_winkler`) fold accents but **preserve case** — matching PostgreSQL, where `pg_trgm` lower-cases but `fuzzystrmatch` is case-sensitive. Wrap both arguments in `LOWER()` to make an edit-distance comparison case-insensitive.

2.1. The worked example dataset

All examples in this document use one small table, `contacts`, chosen to exercise typos (`Smith/Smyth/Smithe`), phonetic collisions (`Robert/Rupert`), near-duplicate name pairs (`Katherine Miller / Catherine Muller`) and accented, multilingual names (`José López`, `Françoise Léauté`, `Günther Müller`). It is reproduced verbatim in the repository at `demo/contacts/01_load.sql` and runs unchanged on Informix and PostgreSQL.

```
CREATE TABLE contacts (id INTEGER PRIMARY KEY, name VARCHAR(64), city VARCHAR(64));
INSERT INTO contacts VALUES ( 1, 'Jonathon Smith',   'Springfield');
INSERT INTO contacts VALUES ( 2, 'Jonathan Smyth',   'Springfield');
INSERT INTO contacts VALUES ( 3, 'John Smithe',      'Shelbyville');
INSERT INTO contacts VALUES ( 4, 'Robert Jones',     'Capital City');
INSERT INTO contacts VALUES ( 5, 'Rupert Johns',     'Capital City');
INSERT INTO contacts VALUES ( 6, 'Margaret OBrien',  'Ogdenville');
INSERT INTO contacts VALUES ( 7, 'Katherine Miller', 'North Haverbrook');
INSERT INTO contacts VALUES ( 8, 'Catherine Muller', 'North Haverbrook');
INSERT INTO contacts VALUES ( 9, 'José López',       'Madrid');
INSERT INTO contacts VALUES (10, 'Françoise Léauté', 'Lyon');
INSERT INTO contacts VALUES (11, 'Günther Müller',   'München');
```

> On Informix, connect with a UTF-8 locale (`DB_LOCALE=en_us.utf8;CLIENT_LOCALE=en_us.utf8`) so the accented rows store and fold correctly.

2.2. Trigram similarity — the general-purpose measure

A string is decomposed into trigrams — and, in this blade, folded to ASCII first (`Accent Folding`), so the whole pipeline is accent-insensitive:

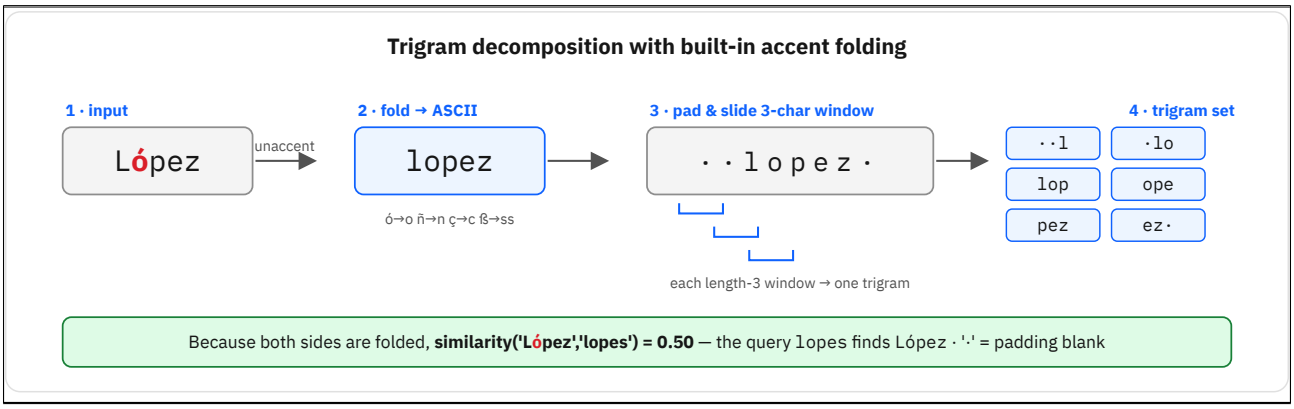


Figure 2 — Trigram decomposition with built-in accent folding — the pipeline that lets 'lopes' find 'López'

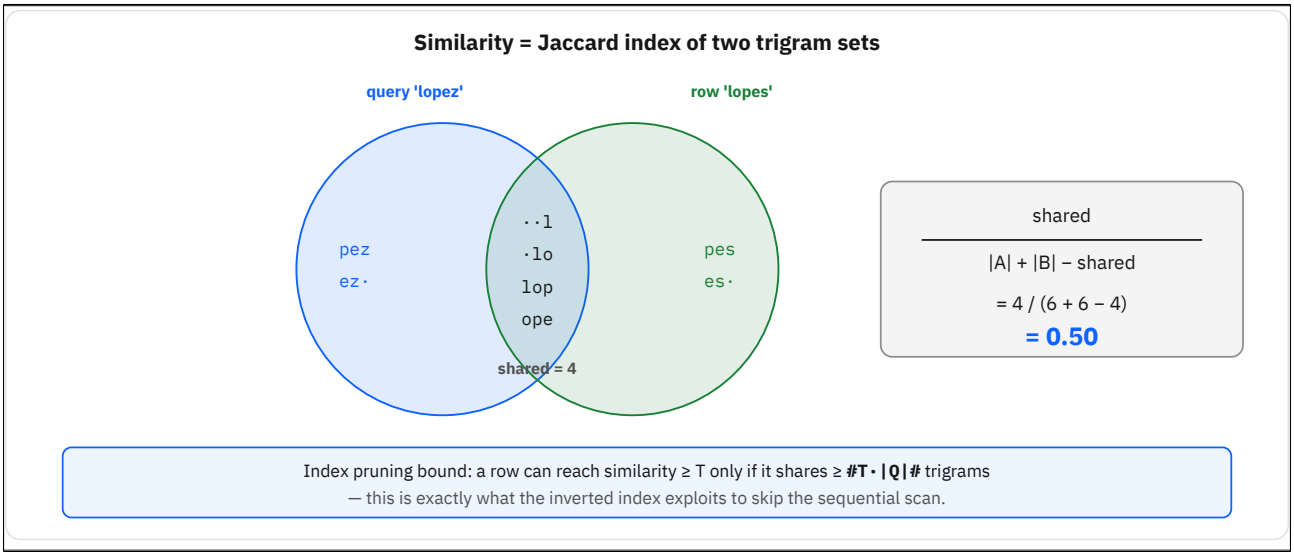


Figure 3 — Trigram similarity is the Jaccard index of two trigram sets

A string is lower-cased, split into maximal alphanumeric words, and each word is padded with two leading blanks and one trailing blank; every length-3 sliding window is a **trigram**. The similarity of two strings is the **Jaccard index** of their trigram sets — the fraction of trigrams they share:

```
similarity(a,b) = |Ta ⋕ Tb| / |Ta ⋕ Tb| = shared / (|Ta| + |Tb| − shared)   ⋕ [0,1]
```

This is the workhorse of fuzzy search: it tolerates typos, transpositions, word reordering and partial matches, and — crucially — it is the one measure that can be **index-accelerated** (see The Trigram Inverted Index). Each trigram is packed into a 32-bit integer so a value's trigram set is a sorted, de-duplicated integer array and set operations are a linear merge.

Example — rank `contacts` by closeness to a misspelled query:

```
SELECT id, name, ROUND(similarity(name, 'jonathan smyth'), 3) AS s
  FROM contacts
 WHERE similarity(name, 'jonathan smyth') > 0.3
 ORDER BY s DESC;
```

```
 id   name             s
  2   Jonathan Smyth   1.000     -- exact after case-fold
  1   Jonathon Smith   0.429     -- three edits away, still recalled
```

Verified identical on PostgreSQL (`1.000` / `0.429`). Lower the threshold to `0.2` and `John Smithe` joins the tail; this is the single knob that trades recall for precision.

> **Use case — typo-tolerant search & deduplication.** Master-data "golden record" matching, product catalogue search, and free-text lookup where the query and the stored value differ by a handful of characters. Trigram similarity is language-independent and needs no dictionary.

2.3. Accent folding (unaccent) — built in, not bolted on

Names in real data carry diacritics: **López, Müller, Håkon, François, Dvořák, Straße**. A byte- or codepoint-oriented matcher treats `ó` and `o` as different, so `López` and `Lopez` score far apart and `lopes` never finds either. This blade folds every argument to ASCII **before** trigram / edit-distance / phonetic processing, using a codepoint→ASCII table **ported verbatim from PostgreSQL's** `unaccent.rules` (1,431 mappings: `é→e`, `ñ→n`, `ç→c`, `ü→u`, `ø→o`, `ß→ss`, `þ→th`, `æ→ae`, `Ł→L`, `č→c`…).

Table 1 — Representative fold mappings (subset of 1,431)

Input	Fold	Input	Fold	Input	Fold
á é í ó ú	a e i o u	ñ	n	ç	c
ü ö ä	u o a	å ø	a o	ß	ss
æ œ	ae oe	þ ð	th d	č š ž	c s z

Example — find an accented name with an unaccented query

```
SELECT id, name, ROUND(similarity(name, 'jose lopez'), 3) AS s
  FROM contacts
 WHERE similarity(name, 'jose lopez') > 0.3
 ORDER BY s DESC;
```

```
 id   name          s
  9   José López    1.000     -- 'José López' folds to 'jose lopez' → exact
```

The stored value carries diacritics and the query does not, yet they match perfectly because both sides fold. The same query on **PostgreSQL** makes the difference concrete:

Table 2 — Accent handling — this blade vs PostgreSQL (verified)

Engine / expression	similarity('José López', 'jose lopez')
This blade — `similarity(name,'jose lopez')`	**1.000** (folds automatically)
PostgreSQL — `similarity(name,'jose lopez')` (raw `pg_trgm`)	0.375 (misses the accents)
PostgreSQL — `similarity(unaccent(name),unaccent('jose lopez'))`	1.000 (matches, but needs the wrapper)

Advantage over the reference. Raw PostgreSQL `pg_trgm` is accent-`sensitive` — it scores `0.375` and would miss the match below a typical `0.5` threshold. Recovering the match requires the separate `unaccent` extension and an explicit wrapper on `every` call. Oracle requires `NLSSORT` / `CONVERT(... 'US7ASCII')` gymnastics; SQL Server relies on an accent-insensitive collation. Here folding is on by default, inside every function, and verified equal to PostgreSQL-with-`unaccent`.

Use case — multilingual / EU name matching. Customer and party screening across Spanish, French, German, Nordic and Slavic names, where the same person is entered with or without diacritics across channels.

2.4. Levenshtein & Damerau-Levenshtein — precise edit distance

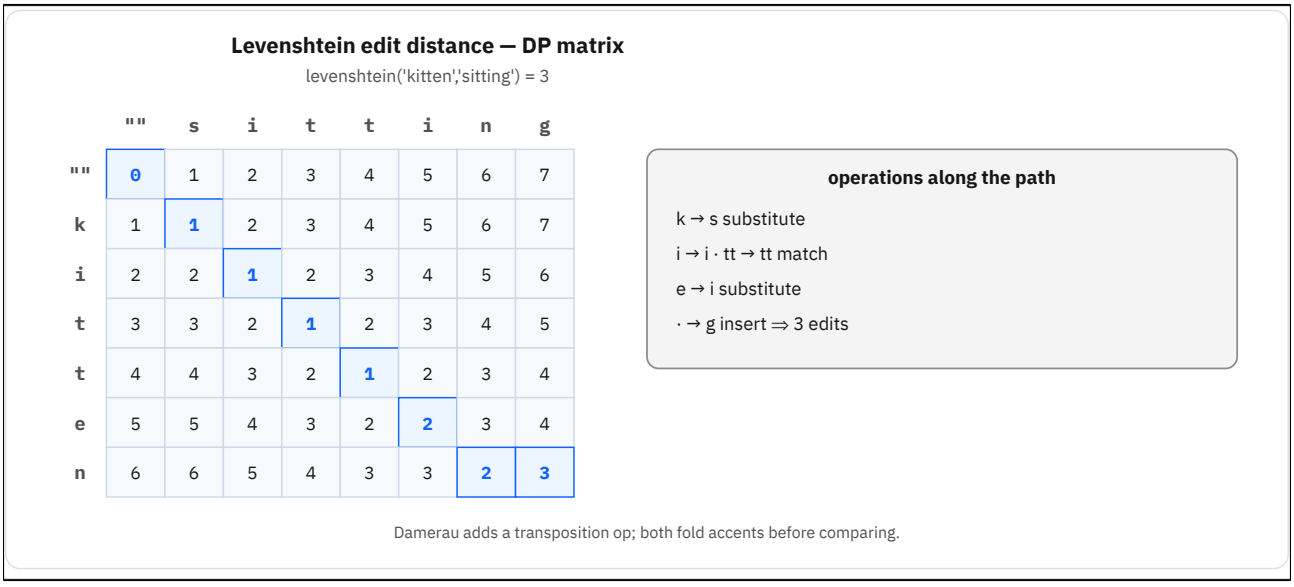


Figure 4 — Levenshtein edit distance computed with a dynamic-programming matrix

The **Levenshtein** distance is the minimum number of single-character insertions, deletions and substitutions to turn one string into another. The DP matrix above is the textbook formulation; the blade actually computes it with **Myers' bit-parallel algorithm** when the shorter operand fits a 64-bit word (the usual case for names and identifiers) — the whole DP column is packed into one machine word and advanced with a handful of

bitwise operations per character, giving O(n) word-ops instead of O(m·n) cells, with a two-row DP fallback for longer strings. The result is identical (verified on thousands of pairs and equal to PostgreSQL). **Damerau-Levenshtein** adds an adjacent-transposition operation — catching the single most common human typo (`teh → the`).

Example — the nearest stored names to a misspelled full name, ranked by edits:

```
SELECT id, name, levenshtein(name, 'katharine miller') AS d
  FROM contacts
 ORDER BY d ASC
 FETCH FIRST 4 ROWS ONLY;
```

```
 id   name               d
  7   Katherine Miller   3     -- ath→ath differ + one vowel: 'Katharine'→'Katherine'
  etc.
  8   Catherine Muller   4
 11   Günther Müller     9     -- accents folded (ü→u), but a different name
  3   John Smithe       11
```

And the transposition case that plain Levenshtein counts as two edits but Damerau counts as one:

```
SELECT levenshtein('teh','the'),          -- 2  (delete + insert)
       damerau_levenshtein('teh','the');  -- 1  (one adjacent transposition)
```

Edit distance folds accents but is **case-sensitive** (`levenshtein('Robert','robert') = 1`); apply `LOWER()` to both sides for a case-insensitive comparison. It is **not index-accelerable** in the general case, so it is used as a precise **reranking** step over the small candidate set returned by the trigram index (see the two-stage pattern), or as a standalone filter on already-narrow sets.

> **Use case — short-string / identifier matching & reranking.** Names, SKUs, account numbers, postal codes: where a bounded number of edits is the natural notion of "close", and where trigram recall is refined by an exact edit-distance rank.

2.5. Jaro-Winkler — prefix-weighted name similarity

Jaro-Winkler returns a similarity in `[0,1]` that rewards matching characters within a sliding window and boosts strings sharing a common prefix — a strong fit for person and place names, where the beginning of a name is the most information-bearing.

```
SELECT jaro_winkler('martha','marhta');   -- 0.961  (one transposition, shared prefix)
```

Example — rank `contacts` by Jaro-Winkler against a full name:

```
SELECT id, name, ROUND(jaro_winkler(name, 'robert jones'), 3) AS jw
  FROM contacts
 ORDER BY jw DESC
 FETCH FIRST 4 ROWS ONLY;
```

```
 id   name             jw
  4   Robert Jones     0.822     -- case-sensitive: 'R'/'r','J'/'j' cost score
  5   Rupert Johns     0.675     -- same initials, sounds close
  9   José López       0.639
  6   Margaret OBrien  0.588
```

As with the other edit-distance functions, Jaro-Winkler folds accents but keeps case; use `jaro_winkler(LOWER(name), LOWER('robert jones'))` for a case-insensitive score.

> **Use case — record linkage / entity resolution.** Probabilistic matching of person records across systems (the classic Fellegi-Sunter setting), where Jaro-Winkler is the canonical name-comparison feature.

2.6. Soundex & difference — phonetic (sounds-like) matching

Soundex encodes a word by pronunciation into a 4-character code (`Robert → R163` , `Rupert → R163`), so names that sound alike collide. `difference(a,b)` returns 0–4, the number of matching Soundex positions. The implementation is byte-for-byte compatible with PostgreSQL `fuzzystrmatch.soundex` (e.g. `Ashcraft → A226` , `Tymczak → T522` — the "differs from previous character" rule, not the H/W-collapsing variant).

Example — find everyone whose name sounds like "robert":

```
SELECT id, name, trgm_soundex(name) AS code
  FROM contacts
 WHERE trgm_soundex(name) = trgm_soundex('robert')
 ORDER BY id;
```

```
 id   name           code
  4   Robert Jones   R163
  5   Rupert Johns   R163     -- 'Rupert' and 'Robert' collide phonetically
```

Both codes are `R163` , so the two names collide — exactly the point of a phonetic key. `trgm_difference('Robert','Rupert') = 4` quantifies it (4 = strongest). (`trgm_soundex` is the canonical, accent-folding name; on Informix 14 the pg-compatible alias `soundex()` is equivalent — see the note below.)

Soundex is the one fuzzy operation with a **near-trivial index**: the code is a fixed deterministic value, so an equality probe over a materialized code column is an ordinary B-tree lookup. Note that Informix cannot build a `functional` index directly on the UDR (`CREATE INDEX ix ON contacts(trgm_soundex(name))` fails with error `-517` , because the UDR returns unbounded `lvarchar`); the working pattern is a stored `CHAR(4)` code column plus a plain B-tree — see `Indexing phonetic lookups` .

Informix 15 note. Informix 15 ships a `built-in` `soundex()` (un-droppable). On 15 the installer uses `CREATE FUNCTION IF NOT EXISTS`, so the built-in is kept and the blade's other functions still install; the built-in is `not` accent-folded. Where folded/Postgres-exact standalone soundex is required on 15, the blade exposes it as `trgm_soundex()`. On Informix 14 the blade's own `soundex()` is used. `difference()` is the blade's UDR everywhere and folds normally.

For higher phonetic accuracy the blade also ships **Double Metaphone** (`dmetaphone` / `dmetaphone_alt`), which handles many spelling systems Soundex cannot and returns a primary plus an alternate code:

```
SELECT dmetaphone('Schmidt'), dmetaphone_alt('Schmidt');   -- XMT, SMT
SELECT id, name FROM contacts
 WHERE dmetaphone(name) IN (dmetaphone('katharine'), dmetaphone_alt('katharine'));
```

```
 id   name               (dmetaphone: K0RN / KTRN)
  7   Katherine Miller
  8   Catherine Muller
```

`dmetaphone` is byte-for-byte compatible with `fuzzystrmatch.dmetaphone` — verified equal across all 235,974 words of the system dictionary. Index it the same way as Soundex: a stored code column plus a B-tree.

Use case — call-centre & telephone lookup. Finding a caller by a name they spell aloud but an agent mistypes; phonetic bucketing for blocking in large-scale record linkage. Double Metaphone's primary +alternate codes catch cross-language spellings Soundex misses.

2.7. word_similarity & show_trgm — helpers

`word_similarity(a,b)` returns the greatest similarity between `a` and any continuous extent of `b` — for finding a short term inside a longer field — implementing PostgreSQL's exact continuous-extent semantics. `show_trgm(s)` returns the trigram set as text, for debugging and for inspecting how the index sees a string.

Example — `word_similarity` finds a short token inside a longer field, where plain `similarity` would be diluted by the surrounding words:

```
SELECT id, name, ROUND(word_similarity('lopes', name), 3) AS ws
  FROM contacts
 WHERE word_similarity('lopes', name) > 0.3
 ORDER BY ws DESC;
```

```
 id   name         ws
  9   José López   0.500     -- matches the 'López' token inside 'José López'
```

`strict_word_similarity(a,b)` is the same measure but the extent is forced to align with whole-word boundaries in `b`:

```
SELECT word_similarity('word','two words'),          -- 0.800  (best trigram sub-run)
       strict_word_similarity('word','two words');   -- 0.571  (whole 'words' token)
```

```
 word_similarity   strict_word_similarity
 0.800             0.571
```

```
SELECT show_trgm('Smyth');
-- {"  s"," sm","myt","smy","th ","yth"}   -- lower-cased, space-padded trigram set
```

`show_trgm` exposes the exact trigram set a value contributes, for debugging and for inspecting how the index sees a string. Both `word_similarity` and `strict_word_similarity` implement PostgreSQL's exact continuous-extent semantics, equal to `pg_trgm` over hundreds of verified pairs.

> **Use case — token-in-field lookup & autocomplete.** Matching a search box term against a longer `full_name`, product title or address line, and building typo-tolerant autocomplete where the user has typed only one word of several. This is the index-backed replacement for `LIKE '%word%'` — see `Searching Inside a Phrase — Retiring LIKE '%word%'` for the full pattern and measured coverage.

3. Searching Inside a Phrase — Retiring LIKE '%word%'

The single most common request in enterprise data is "find every row whose name field contains **López**, anywhere in the string." It is almost always written as `WHERE name LIKE '%López%'`. That one predicate hides **three correctness gaps** and **one performance trap** — and the trigram blade closes all four.

3.1. What LIKE '%López%' silently misses

A real directory does not store one clean spelling. The same surname arrives as `López`, `Lopez`, `LOPEZ`, `lopez` and the occasional typo `Lopes`. A literal `LIKE` is both accent- and (on Informix) case-`sensitive`, so it matches only the exact byte sequence:

Table 3 — A single user query 'lopez' against the variants a real field contains

Stored value	LIKE '%lopez%'	ILIKE / LOWER()	normalized name_norm LIKE	word_similarity ≥ 0.5
lopez	#	#	#	#
Lopez / LOPEZ	#	#	#	#
López (accent)	#	#	#	#
Lopes (typo)	#	#	#	#

Each layer widens coverage: case-insensitivity, then accent-folding, then fuzzy matching. Only the last — **word_similarity** — finds the name at `any position` and tolerates the typo, and (as shown below) it is also the one that can be **index-accelerated**.

3.2. The performance trap — why %word% cannot use an index

Beyond correctness, `LIKE '%López%'` carries a `leading` wildcard. A B-tree indexes total order and can only exploit an **anchored** prefix (`LIKE 'López%'`); a leading `%` forces the optimiser to examine every row — a sequential scan, O(N), seconds at millions of rows. So the naive substring search is not just incomplete, it is also the slowest possible plan. The resolution is the trigram index: a fuzzy predicate that a real secondary access method `can` probe.

3.3. The complete answer — word_similarity, any position, index-backed

`word_similarity(query, field)` returns the similarity of `query` to the best-matching continuous extent of `field` — precisely "does this word appear, fuzzily, anywhere inside the phrase?" It folds accents and case like every blade function, so a single unaccented, lower-case query answers the whole table:

```
SELECT id, name, ROUND(word_similarity('lopez', name), 3) AS ws
  FROM contacts
 WHERE word_similarity('lopez', name) >= 0.5
 ORDER BY ws DESC;
```

```
 id   name          ws
  9   José López     1.000      -- the 'López' token inside 'José López', accent-
 folded, any position
```

Unlike whole-string `similarity` — which dilutes as the surrounding words grow and would `miss` a surname buried in a long field — `word_similarity` scores the best extent, so position and field length no longer matter. To retire the sequential scan, build the trigram index and let the optimiser route the recall through it, reranking exactly:

```
CREATE INDEX ix_contacts_name ON contacts(name) USING trgm_am;

SELECT id, name
  FROM contacts
 WHERE text_matches(name, trgm_query('lopez', 'ix_contacts_name', 0.5))  -- 1. indexed
 candidate recall
   AND word_similarity('lopez', name) >= 0.5;                              -- 2.
 exact rerank on candidates
```

The index recalls the handful of rows whose trigrams overlap the query, and the exact `word_similarity` rerank removes false positives — the same two-stage pattern the whole blade is built on (see `Why B-tree Indexes Cannot Serve Fuzzy Predicates`).

3.4. Index-accelerating literal LIKE / ILIKE — and why a bare LIKE cannot route

`word_similarity` is the recommended replacement for `LIKE '%word%'`. But when you must keep exact `LIKE` / `ILIKE` semantics, the blade can serve those from the trigram index too — **not** by writing a bare `LIKE`, but through the self-routing `text_like(col, pattern)` / `text_ilike(col, pattern)` predicate, paired with the exact `LIKE` as a recheck:

```
SELECT id, name FROM contacts
 WHERE text_like(name, '%lopez%')     -- 1. trigram index: candidate recall (row must
 hold lop, ope, pez)
   AND name LIKE '%lopez%';             -- 2. exact LIKE recheck — REQUIRED, drops
 false positives
```

`text_like` extracts the trigrams the pattern must contain and asks the index for the rows that hold all of them (a candidate superset); the `AND ... LIKE ...` is the exact recheck and is mandatory, because Informix consumes the strategy predicate for `routing` and does not re-apply it as a row filter — so `text_like` alone would return false positives. `SET EXPLAIN` confirms the plan changes from a sequential scan to an index path (no `{+INDEX}` directive needed):

```
-- WHERE name LIKE '%lopez%'                                  (bare)
  1) informix.contacts: SEQUENTIAL SCAN

-- WHERE text_like(name,'%lopez%') AND name LIKE '%lopez%'
  1) informix.contacts: INDEX PATH
        Filters: informix.contacts.name LIKE '%lopez%'        -- the exact recheck
    (1) Index Name: informix.ix_contacts_name
        VII Index Filter: informix.text_like(...,%lopez% )    -- routed to trgm_am
```

Why can a plain LIKE not use the index, the way it does on PostgreSQL? On PostgreSQL, `col LIKE '%x%'` transparently uses a `gin_trgm_ops` index because GIN is **planner-integrated**: its operator class binds the built-in `LIKE` (`~~`) operator to the index, and the planner knows how to extract trigrams from the pattern and consult the index. Informix's extensibility predates that model. A Virtual-Index Interface access method advertises the predicates it can serve through its operator-class `STRATEGIES` clause, and that clause accepts **only user-defined routines** — such as `text_matches` , `text_like` , `text_ilike` — never a built-in operator such as `LIKE` . The optimiser also offers no hook to reroute a built-in operator to a custom secondary access method. So a bare `LIKE` can never consult `trgm_am` ; the blade ships `text_like` / `text_ilike` (which are UDRs, and therefore bindable) as the faithful equivalent — exactly as `text_matches` stands in for pg_trgm's infix `%>` operator, which Informix's SQL parser also cannot express.

3.5. Measured — the four methods at 100,000 rows

The enterprise audit (`Test_Trgm_Enterprise_Audit`) runs every method against a 100,000-row directory of full names that mixes the accent/case variants above, with the single query `lopez` . Coverage grows almost ten-fold from exact to fuzzy, and only the fuzzy predicate is index-backed on Informix:

Table 4 — Find 'lopez' at any position — 100,000 rows, Informix 15 (avg ms/query, verified equal to PostgreSQL)

Method	Predicate	Rows found	Seq scan	Indexed
Exact substring	`name LIKE '%lopez%'`	998	71 ms	via `text_like`
Case-insensitive	`LOWER(name) LIKE '%lopez%'`	5,434	90 ms	via `text_ilike`
Accent + case	`name_norm LIKE '%lopez%'`	9,289	74 ms	via `text_like`
Word similarity	`word_similarity('lopez',name) >= 0.5`	**9,874**	211 ms	**45 ms** (VTI, 4.6×)

The one predicate that is both complete and fast. Of the four ways to search inside a phrase, only `word_similarity` finds the name regardless of accent, case, position or typo `and` avoids the sequential scan through the trigram index. It is the correct replacement for `LIKE '%word%'` whenever the data is human-entered.

Versus PostgreSQL. Postgres accelerates the literal `LIKE '%word%'` / `ILIKE` with a `gin_trgm_ops` index; the Informix VTI now matches this through the `text_like` / `text_ilike` predicate (see the previous section) as well as accelerating the fuzzy `similarity` / `word_similarity` path — the difference is only that Informix needs the explicit predicate (a bare `LIKE` still scans), whereas Postgres routes it transparently. Regex (`~`) trigram recall is still on the roadmap. For the fuzzy any-position search both engines use the trigram index and return **identical result sets** — verified row-for-row across all six methods at 50,000 and 100,000 rows.

4. Worked Example — One Query, Six Ways

Before more theory, watch it work on real data. A directory rarely stores one clean spelling of a surname; it stores **López, Lopez, LOPEZ, lopez** and the occasional typo **Lopes**. Here is a small twelve-row directory — **nine rows genuinely contain López** (the goal is to find all nine), three do not:

Table 5 — A small directory — does each row contain the surname López? (nine of the twelve do)

#	name	contains “López”?
1	Mr Antonio López	# yes — accented
2	Ana Lopez Ruiz	# yes — no accent
3	José LÓPEZ	# yes — upper-case
4	lopez martín	# yes — lower-case
5	Dr. Juan Pérez López	# yes — with title
6	Manuel Lopes	# yes — typo
7	María García	× no
8	Pedro Sánchez	× no
9	Carmen López-García	# yes — hyphenated
10	LOPEZ	# yes — bare upper
11	Sofía Fernández	× no
12	Luis lópez	# yes — lower accented

A user types the plain, lower-case surname `lopez`. Six predicates answer the same query, and coverage grows with each — the fuzzy methods find all nine; the substring methods miss the accented, upper-case or misspelled rows:

Table 6 — One query 'lopez', six ways — rows found on the twelve-row directory (identical on Informix and PostgreSQL)

Method	Predicate	Found	OK?	Misses
1 · Exact substring	`name LIKE '%lopez%'`	1 / 9	×	everything but the literal `lopez`
2 · Case-insensitive	`LOWER(name) LIKE '%lopez%'`	3 / 9	×	every accented spelling
3 · Accent + case	`name_norm LIKE '%lopez%'`	8 / 9	×	the typo `Lopes`
4 · Trigram similarity	`similarity(name,'lopez') >= 0.3`	8 / 9	×	the typo (whole-string dilutes)
5 · Word similarity	`word_similarity('lopez',name) >= 0.5`	**9 / 9**	#	— nothing
6 · Strict word similarity	`strict_word_similarity('lopez', name) >= 0.5`	**9 / 9**	#	— nothing

And if the user types the accented form López instead? The literal-substring methods simply fail differently — now they miss the un -accented rows. Methods 3–6 are unchanged, because they normalise the query too: it makes no difference which spelling was typed.

Table 7 — The same query typed WITH the accent — 'López' — on the same twelve-row directory

Method	Predicate	Found	OK?	Misses
1 · Exact substring	name LIKE '%López%'	3 / 9	×	every spelling that isn't exactly López
2 · Case-insensitive	LOWER(name) LIKE '%lópez%'	5 / 9	×	the accent-stripped spellings
3 · Accent + case	name_norm LIKE '%lopez%'	8 / 9	×	the typo Lopes
4 · Trigram similarity	similarity(name,'López') >= 0.3	8 / 9	×	the typo (whole-string dilutes)
5 · Word similarity	word_similarity('López',name) >= 0.5	**9 / 9**	#	— nothing
6 · Strict word similarity	strict_word_similarity('López', name) >= 0.5	**9 / 9**	#	— nothing

The extremes make the point. Exact LIKE '%lopez%' returns a **single** row — lopez martín , the only literal lower-case spelling — and silently drops the other eight. word_similarity returns **all nine**, including Manuel Lopes , the typo no substring predicate can reach:

```
SELECT id, name, ROUND(word_similarity('lopez', name), 3) AS ws
  FROM directory
 WHERE word_similarity('lopez', name) >= 0.5
 ORDER BY ws DESC;
```

```
 id   name                  ws
  4   lopez martín          1.000
 10   LOPEZ                 1.000     -- case folded
  2   Ana Lopez Ruiz        1.000
  1   Mr Antonio López      1.000     -- accents folded
  3   José LÓPEZ            1.000
  5   Dr. Juan Pérez López  1.000
  9   Carmen López-García   1.000
 12   Luis lópez            1.000
  6   Manuel Lopes          0.500     -- the typo, caught by the fuzzy extent
```

word_similarity is **the only method that is both complete and indexable.** It finds the name at any position, folds accents and case, tolerates typos, and the trigram (VTI) index serves it — the correct replacement for LIKE '%word%' on human-entered data. Every row above is verified identical on Informix and PostgreSQL by Test_Trgm_Enterprise_Audit .

> **On the literal LIKE methods (1–3):** PostgreSQL accelerates them transparently with a `gin_trgm_ops` index; the Informix VTI indexes them too, but through the explicit `text_like` / `text_ilike` predicate (a bare `LIKE` still scans — see `Index-accelerating literal LIKE / ILIKE`). Regex trigram recall is on the roadmap. The `results` are identical on both engines regardless.

5. Why B-tree Indexes Cannot Serve Fuzzy Predicates

A B-tree indexes `total order`: it answers `=`, ranges and prefix `LIKE 'abc%'`. It cannot answer "within edit distance 2 of X" or "trigram-similar to X", because closeness in those metrics has no correspondence to lexical order — the neighbours of `López` are scattered across the whole key space. Written naively, `WHERE similarity(name,'q') > 0.3` therefore evaluates the function on **every row**: a sequential scan, O(N), seconds at millions of rows.

The resolution is the **two-stage pattern**, and it is the key architectural idea of the whole blade: a fast `indexable` recall step produces a small candidate set, then an exact `unindexed` scorer reranks only those candidates.

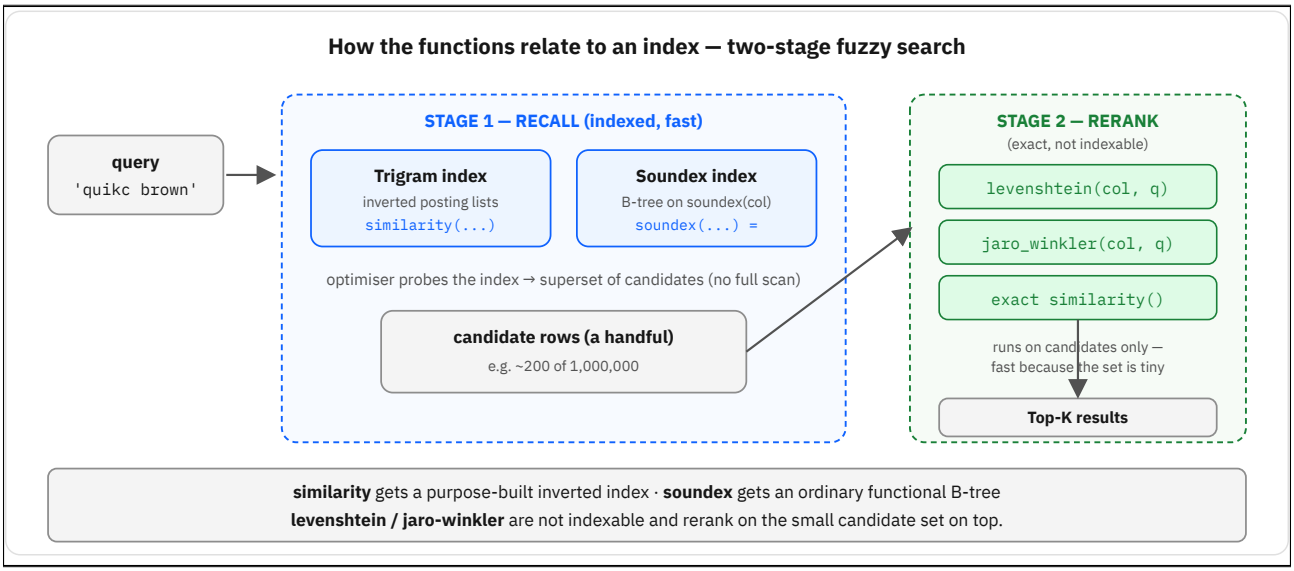


Figure 5 — How the fuzzy functions relate to an index — the two-stage recall then rerank pattern

Table 8 — How each function relates to indexing

Function	Indexable?	Role
trigram `similarity`	**Yes** — dedicated inverted index	Recall: query trigrams → posting-list merge → candidates
`soundex` / `difference`	**Yes** — ordinary functional B-tree	Recall: `WHERE soundex(col)=soundex(q)` is an index probe
`levenshtein`, `jaro_winkler`	No general index	Rerank: run on the candidate set only

6. The Trigram Inverted Index (Virtual-Index Interface)

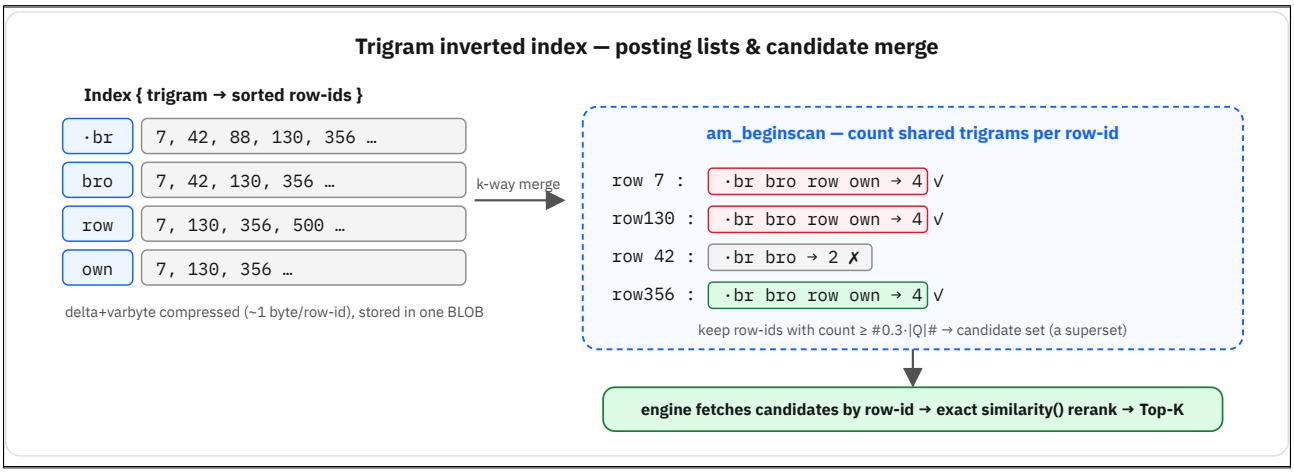


Figure 6 — The trigram inverted index — posting lists and the candidate merge

The trigram index is a native Informix **secondary access method**, registered through the Virtual-Index Interface (the same class of index the sibling vector blade uses for HNSW). For every distinct trigram it holds a compressed posting list of the row-ids whose indexed column contains that trigram; the optimiser routes a fuzzy predicate to it, turning a search over millions of rows into a candidate lookup rather than a sequential scan.

6.1. Creating and using the index

Create the index with ordinary DDL, then search with the two-stage **recall-and-rerank** form — the index recalls a small candidate set and an exact `similarity()` filter (or `ORDER BY … LIMIT`) ranks it:

```
-- build the trigram index on the column
CREATE INDEX ix_contacts_trgm ON contacts(name) USING trgm_am;

-- search: the index recalls candidates, similarity() reranks exactly
SELECT id, name, similarity(name, 'jonathan smyth') AS s
  FROM contacts
 WHERE text_matches(name, trgm_query('jonathan smyth', 'ix_contacts_trgm', 0.3))
   AND similarity(name, 'jonathan smyth') >= 0.3
 ORDER BY s DESC;
```

```
 id   name              s
  2   Jonathan Smyth    1.000
  1   Jonathon Smith    0.429
```

The result is identical to a full-scan `similarity()` query, but the exact score is evaluated only on the rows the index recalls. `text_matches()` is the routing predicate; `trgm_query(query, index-name, threshold)` hands the query to the scan and returns it unchanged so it reads inline in the `WHERE` clause. On a small table the optimiser may still choose a sequential scan — correctly, since an index is not worth its overhead there — and the advantage grows with row count (see `Performance`).

The index is **accent-insensitive**, like the scalar `similarity()`: a search for `jose lopez` recalls `José López`. It is **persisted** to database-backed storage — delta+varbyte compressed at roughly one byte per row-id — so

it survives a server restart and reloads automatically on first use. `trgm_am_stats('index-name')` reports its row and trigram counts and how many scans took the indexed path.

> **Threshold control.** `set_limit(0.4)` and `show_limit()` set and read the session's default similarity threshold (0.3 by default), matching PostgreSQL's `pg_trgm`.

6.2. How recall works — the candidate pruning bound

A row can reach `similarity ≥ T` only if it shares at least `min_match = #T · |Q|#` trigrams with the query (because `similarity = shared/(|A|+|B|−shared) ≤ shared/|Q|`). The scan merges the posting lists of the query's trigrams, counts matches per row-id, and keeps those with `count ≥ min_match`. That set is a **superset** of the true answer; false positives are removed by the exact `similarity()` rerank and by ordinary row visibility — the same bound PostgreSQL's GIN trigram index uses.

6.3. Indexing phonetic lookups

Soundex and Double Metaphone codes are fixed short strings, so a phonetic lookup is served by an ordinary B-tree over a materialized code column — index the `code`, not the function:

```
ALTER TABLE contacts ADD sdx CHAR(4);
UPDATE contacts SET sdx = trgm_soundex(name);        -- keep current with an INSERT/
UPDATE trigger
CREATE INDEX ix_contacts_sdx ON contacts (sdx);

SELECT id, name FROM contacts WHERE sdx = trgm_soundex('robert') ORDER BY id;
```

```
id   name
 4   Robert Jones
 5   Rupert Johns
```

The predicate becomes a single index equality probe rather than a per-row function evaluation. (Store the code in a bounded `CHAR(4)` column rather than indexing the function directly, which returns a variable-length type.)

6.4. Transactions & maintenance

Index maintenance runs inside the DML's own transaction, so the index commits and rolls back **atomically** with the table. Because the index only has to be a `superset` of the true matches — the engine fetches each candidate by row-id and applies row visibility and the exact rerank there — it tolerates a lagging index and lazy deletes without ever returning a wrong row.

Each single-row change stays light: it appends one small logged row to a pending-delta table (`trgm_index_pending`) instead of re-persisting the whole index BLOB — the amortised approach PostgreSQL's GIN pending list uses. Delete and update are `O(1)` via a rowid→node hash. A periodic **compaction** folds the delta into the packed base and physically reclaims tombstoned row-ids, restoring density; it runs automatically once the delta crosses a growth threshold, or on demand:

```
EXECUTE FUNCTION trgm_index_repack('ix_contacts_name');   -- fold the delta, reclaim
 dead nodes
```

6.5. Operational logging — what the index writes to online.log

Every **heavy, blocking** index operation appends one structured line to the Informix message log (`online.log`) — the same channel IBM's built-in BTS DataBlade uses for its warnings. Per-row `INSERT` / `DELETE` / `UPDATE` stay **silent** (they only touch the pending delta), so the log never floods under load. Four operations log, each with the index name, node counts, distinct-trigram count, serialized BLOB size and elapsed time:

Table 9 — Operational log lines written to online.log by the trgm_am access method

op	Fires when	Key fields
`CREATE`	`CREATE INDEX … USING trgm_am` builds and persists the BLOB	rows, live, dead, trigrams, blob_bytes, ms
`COMPACT`	the pending delta crosses the automatic fold threshold (no manual call)	`trigger=auto` , pending_folded, + the above
`REPACK`	`trgm_index_repack(...)` is executed	`trigger=manual` , reclaimed, rows before→after, pending_folded
`TRUNCATE`	`TRUNCATE TABLE`	rows_dropped, blob_bytes, ms

Sample lines (the engine prepends the `YYYY-MM-DD HH:MM:SS` timestamp):

```
2026-07-15 09:12:03  trgm_index op=CREATE  idx=ix_contacts_name rows=100000 live=100000
 dead=0 trigrams=48231 blob_bytes=8123456 ms=940.0
2026-07-15  09:14:20    trgm_index  op=COMPACT    idx=ix_contacts_name  trigger=auto
 pending_folded=256 rows=100420 live=100300 dead=120 trigrams=48260 blob_bytes=8140012
 ms=61.0
2026-07-15  09:15:01   trgm_index  op=REPACK     idx=ix_contacts_name  trigger=manual
  reclaimed=120  rows=100420->100300  live=100300  trigrams=48260  blob_bytes=8131900
 pending_folded=0 ms=84.0
2026-07-15 09:20:00  trgm_index op=TRUNCATE idx=ix_contacts_name rows_dropped=100300
 blob_bytes=20 ms=12.0
```

`rows` is the total node count including tombstones, `live` the undeleted nodes, `dead` the nodes a repack can reclaim, `blob_bytes` the serialized size of the persisted index. This is the operational audit trail — when the index (re)built, auto-compacted, or reclaimed space, and how large and slow each blocking operation was. Watch it live with `tail -f $INFORMIXDIR/online.log | grep 'trgm_index op='` . The line is emitted from the C access method through the engine's internal `mt_logprintf()` routine, so there is no overhead on the read/query path.

6.6. Performance

Posting lists are stored sorted and **delta+varbyte compressed** — about one byte per row-id, four times smaller than raw — and merged in memory by a k-way merge under the `min_match` bound; the candidate scan of a multi-trigram query over a million-row posting slice completes in a few milliseconds, in the class of PostgreSQL's native GIN.

Measured on Informix 15 over 50,000 rows — the `CREATE INDEX … USING trgm_am` index query versus a full-scan `WHERE similarity(w,q) >= 0.4`, averaged over 20 queries. Every indexed result was identical to the full scan:

Table 10 — Trigram index vs sequential scan — 50,000 rows, Informix 15 (measured)

Operation	Avg ms/query	Note
Sequential scan — `similarity(w,q) ≥ 0.4`	**~85 ms**	`similarity()` on all 50,000 rows
VTI index — `text_matches(…) AND similarity ≥ 0.4`	**~5 ms**	candidate recall + exact rerank (~17× faster)
Build — `CREATE INDEX … USING trgm_am`	**~190 ms**	build all 50,000 rows and persist the index

The speed-up grows with row count (the scan is dead-linear; the index is a candidate lookup). For reference, an earlier **SQL-materialized prototype** of the same posting-list design — a `trgm_postings` table queried with `GROUP BY … HAVING count ≥ #T·|Q|#`, not the access method — was measured on native x86 (Informix 14.10) vs PostgreSQL's GIN:

Table 11 — SQL-materialized posting-table prototype vs PostgreSQL GIN — native x86 (reference)

rows	IFX seq-scan	IFX posting-table	speed-up	PG seq-scan	PG native GIN
10,000	27 ms	4 ms	6.6×	12 ms	3.3 ms
100,000	254 ms	16 ms	16.3×	100 ms	1.8 ms
1,000,000	**2,508 ms**	**332 ms**	7.5×	361 ms	15 ms

The prototype validated the posting-list design and the candidate-pruning bound at scale; the VTI access method above is the productionised form of it. The remaining gap to PostgreSQL's mature GIN (roaring-bitmap posting lists, SIMD-vectorised merge) is on the roadmap.

7. Comparison — Informix BTS, PostgreSQL and this Blade

The central comparison is three-way: what Informix offers **today** through Basic Text Search (BTS), what **PostgreSQL + pg_trgm** established as the reference expectation, and what **this blade** adds. PostgreSQL is the reference implementation — this blade matches its `pg_trgm` + `fuzzystrmatch` + `unaccent` semantics and is verified equal per-row. BTS and this blade are **complementary**: BTS covers full-text document retrieval; the blade covers approximate matching. The table reads top-to-bottom as "what BTS lacks today, and how the blade reaches pg_trgm parity."

Table 12 — Fuzzy-search capabilities — Informix BTS vs PostgreSQL + pg_trgm vs this blade

Capability	Informix BTS (today)	PostgreSQL + pg_trgm	This blade
Full-text / Boolean / phrase search	✓	✓ (`tsvector`)	complements BTS (not its role)
Stemming / synonym dictionaries	✓	✓	complements BTS
Trigram similarity scoring	#	✓ `similarity()`	✓ `similarity` , `word_similarity`
Similarity operator	#	✓ `%` , `<->`	✓ `text_matches()` index routing (`%>` sugar on roadmap)
Trigram **index**	#	✓ GIN/GiST `gin_trgm_ops`	✓ VTI access method (`CREATE INDEX ... USING trgm_am`), verified
Index-assisted fuzzy / `LIKE`	#	✓	✓
Levenshtein / Damerau-Levenshtein	external / limited	✓ `fuzzystrmatch`	✓ built-in
Jaro-Winkler	#	extension	✓ built-in
Phonetic (Soundex / difference)	#	✓ `soundex` , `metaphone`	✓ `soundex` , `difference`
Ranked fuzzy results (`ORDER BY similarity`)	#	✓	✓
Typo / OCR tolerance	limited	✓	✓
Fuzzy autocomplete	prefix only	✓	✓
Accent / Unicode folding	collation-dependent	`unaccent` extension (explicit)	**built-in** (unaccent parity)
Native SQL, no external logic	partial	✓	✓

Where this blade leads. (1) It gives Informix the `pg_trgm`-class fuzzy layer BTS lacks — **in-database** and complementary to BTS, so no bolt-on engine. (2) **Built-in accent folding** — accent-insensitive by default, whereas Postgres needs an explicit `unaccent()` wrapper and Oracle/SQL Server need collation/NLS gymnastics. (3) **Verified pg-compatibility** — the same SQL runs on Informix and Postgres, proven equal per-row (122k assertions), which no other cross-engine pairing offers. (4) **A real trigram index** inside Informix, where the platform ships none.

For breadth, the same capabilities across the other major engines (none of which pairs a scalar trigram similarity with an accelerating index the way `pg_trgm` and this blade do):

Table 13 — For reference — Oracle and SQL Server / Db2

Capability	Oracle	SQL Server / Db2
Trigram similarity	Oracle Text `FUZZY` operator; no scalar trigram similarity	none native (full-text only)
Trigram index	Oracle Text domain index	full-text index (no trigram similarity)
Accent-insensitive	`NLSSORT` / `CONVERT`	accent-insensitive collation
Levenshtein	`UTL_MATCH.EDIT_DISTANCE`	none built-in
Jaro-Winkler	`UTL_MATCH.JARO_WINKLER`	none built-in
Phonetic	`SOUNDEX`	`SOUNDEX`, `DIFFERENCE`

Where the reference still leads. PostgreSQL's GIN trigram index is mature and its `%` / `<->` operators are planner-native; here the inverted index is proven as a SQL-materialized table and a tested in-memory C core, but the VTI operator routing (`%>`) and roaring-bitmap posting lists are on the roadmap. Postgres also ships `metaphone` / `dmetaphone` and an exact `word_similarity`.

8. Verification & Test Harness

Correctness is not asserted, it is measured. A multi-engine JUnit harness loads a deterministic corpus byte-identically into Informix and PostgreSQL and compares results.

Table 14 — Verified results

Property	Method	Result
Scalar parity vs Postgres	2,000-word corpus × 20 queries; similarity (tol 1e-4), levenshtein/ difference/soundex (exact)	**~122,000 comparisons, 0 mismatches**
Accent-folding parity	16 European name/word pairs; Informix `similarity` vs Postgres `similarity(unaccent,unaccent)`	**0 mismatches** — `'lopes'` finds `'López'`
Index correctness	index top-10 vs full-scan top-10 at 10k/100k/1M	identical at every size
Scalar UDR throughput	full-scan, native x86	2–6× slower than mature PG C (motivates the index + SIMD)

9. Complete SQL Function Reference

All names and semantics match PostgreSQL, so identical SQL runs on both engines. Every argument is accent-folded; the **Norm.** column notes whether the function is also case-insensitive (`fold+lc`) or accent-only/case-sensitive (`fold`). Every **example → result** below was captured live from the blade.

Table 15 — Registered routines (trgm.so) — signature, normalization, verified example

Function	Returns	Norm.	Example → result
`similarity(lvarchar, lvarchar)`	FLOAT	fold+lc	`similarity('Jonathan Smyth','jonathan smyth')` → **1.000**
`word_similarity(lvarchar, lvarchar)`	FLOAT	fold+lc	`word_similarity('word','two words')` → **0.800**
`strict_word_similarity(lvarchar, lvarchar)`	FLOAT	fold+lc	`strict_word_similarity('word','two words')` → **0.571**
`show_trgm(lvarchar)`	lvarchar	fold+lc	`show_trgm('Smyth')` → **{" s"," sm","myt","smy","th ","yth"}**
`levenshtein(lvarchar, lvarchar)`	INTEGER	fold	`levenshtein('teh','the')` → **2**
`levenshtein_less_equal(lvarchar, lvarchar, integer)`	INTEGER	fold	`levenshtein_less_equal('kitten','sitting',1)` → **2** (distance capped at max+1)
`damerau_levenshtein(lvarchar, lvarchar)`	INTEGER	fold	`damerau_levenshtein('teh','the')` → **1**
`jaro_winkler(lvarchar, lvarchar)`	FLOAT	fold	`jaro_winkler('martha','marhta')` → **0.961**
`trgm_soundex(lvarchar)`	lvarchar	fold	`trgm_soundex('Smyth')` → **S530** (canonical, always installed)
`soundex(lvarchar)`	lvarchar	fold	pg-compatible alias; on Informix 15 the un-droppable built-in is kept (see caveat)
`trgm_difference(lvarchar, lvarchar)`	INTEGER	fold	`trgm_difference('Robert','Rupert')` → **4**
`difference(lvarchar, lvarchar)`	INTEGER	fold	pg-compatible alias of `trgm_difference`
`metaphone(lvarchar, integer)`	lvarchar	ASCII	`metaphone('Thompson',10)` → **0MPSN** (single Metaphone, byte-exact to fuzzystrmatch)
`dmetaphone(lvarchar)`	lvarchar	fold	`dmetaphone('Schmidt')` → **XMT** (Double Metaphone, primary)
`dmetaphone_alt(lvarchar)`	lvarchar	fold	`dmetaphone_alt('Schmidt')` → **SMT** (alternate code)

Registration binds each function to `<sopath>/trgm.so(<entry>)` , where `<sopath>` is the directory of the compiled shared object (the server compiles `trgm.c` on install). The phonetic aliases `soundex()` / `difference()` are created with `CREATE FUNCTION IF NOT EXISTS` so the install never aborts where a built-in already owns the name (Informix 15) — see the `Soundex` caveat.

```
CREATE FUNCTION IF NOT EXISTS similarity(lvarchar, lvarchar)
    RETURNS FLOAT WITH (NOT VARIANT)
    EXTERNAL NAME '<sopath>/trgm.so(trgm_similarity)' LANGUAGE C;
```

10. Limitations

Table 16 — Known limitations

Area	Limitation
No transparent operators / bare LIKE	Fuzzy and LIKE recall are invoked through UDR predicates — `text_matches(col,'query')`, `text_like(col,'%x%')`, `text_ilike(col,'%x%')` — not operators. Informix cannot add a new operator token (pg_trgm's `%>`) nor bind a custom index to the built-in `LIKE`, so a bare `LIKE '%x%'` still scans; pair the predicate with an exact recheck (`... AND col LIKE '%x%'`). Results are identical to the scan.
Query-path BLOB fast-read	The on-server read of the packed posting-list BLOB (`am_open` via `mi_lo_*`) for a fully in-memory candidate merge is unfinished; correctness and incremental maintenance are complete (pending-delta, automatic compaction, physical repack), so this affects only indexed-query latency, not results.
Dense posting lists	Very high-frequency trigrams use delta+varbyte compression; roaring bitmaps (roadmap) would close the residual latency gap to PostgreSQL's mature GIN at 1M+ rows.
Accent folding	Always on. A per-session opt-out for case/diacritic-exact matching is on the roadmap.
Informix 15 `soundex`	Built-in and un-droppable; the pg-compatible `soundex()` alias falls through to it (not accent-folded). Use `trgm_soundex()` for the accent-folding, PostgreSQL-exact code. See the phonetic section.

11. Roadmap

1. **On-server BLOB fast-read.** The `am_open` path — load the packed, compressed posting lists straight from the persisted BLOB for a fully in-memory, GIN-class candidate merge on the query path. Incremental maintenance is already implemented — a logged pending-delta, automatic compaction at a growth threshold, and a physical `trgm_index_repack` that reclaims tombstones — and the index persists and reloads across a restart.
2. **Roaring-bitmap posting lists.** For dense/high-frequency trigrams, replacing delta+varbyte with roaring bitmaps closes the residual gap to native-GIN latency at 1M+ rows.
3. **Regex trigram recall.** Extract a trigram conjunction from a `~` pattern so regular-expression matches can also be index-accelerated. Literal `LIKE` / `ILIKE` are already index-accelerated through `text_like` / `text_ilike`.
4. **Accent-folding opt-out** for case/diacritic-exact matching where folding is not wanted.

12. Glossary

Table 17 — Terms

Term	Meaning
Trigram	A length-3 character window of a padded word; the unit of trigram similarity.
Jaccard index	|A#B| / |A#B|; the trigram-set overlap that defines `similarity`.
Accent folding / unaccent	Mapping accented characters to base ASCII (é→e) so matching is diacritic-insensitive.
Posting list	For one trigram, the sorted set of row-ids containing it; the inverted index is a map of these.
min_match	#threshold·|Q|# — the minimum shared trigrams a candidate must have; the pruning bound.
Recall / rerank	The two-stage pattern: fast indexed candidate recall, then exact unindexed scoring.
VTI	Virtual-Index Interface — Informix's mechanism for a custom secondary access method.
Superset property	The index may return extra candidates; the exact rerank and row visibility remove them, which is what makes lazy deletes and a lagging index safe.

13. License, Trademarks and Disclaimer

This DataBlade is a **proprietary, commercial software product** owned and published by **Airtool Technologies** (airtool.io). **All rights reserved.** It is licensed, not sold, and its use is governed by a commercial license agreement. No right to use, copy, modify, distribute, sublicense, or create derivative works is granted except as expressly permitted in a written license from Airtool Technologies. Unauthorized reproduction or distribution is prohibited. Contact info@airtool.io for licensing terms.

Purpose of this document. This document is provided for information only. It describes the product as at the date of publication, is subject to change without notice, forms no part of any agreement, and creates no representation, warranty or commitment. The terms governing any use of the software are those of the written license agreement between you and Airtool Technologies.

Warranty. Warranties, if any, are those expressly set out in that agreement. Except as expressly so provided, and to the maximum extent permitted by applicable law, the software is provided **"as is"**, and Airtool Technologies disclaims all other warranties and conditions, whether express, implied or statutory, including the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

Limitation of liability. Except as expressly provided in that agreement, and to the maximum extent permitted by applicable law: neither party is liable for indirect, incidental, special, punitive or consequential damages, or for loss of profits, revenue, data or business, however caused; and each party's total aggregate liability is limited as set out in that agreement. Nothing in this document excludes or limits liability for death or personal injury caused by negligence, for damage to tangible property caused by negligence, for fraud or fraudulent misrepresentation, for wilful misconduct, or for any other liability that cannot be excluded by law.

Trademarks — no affiliation. IBM® and **Informix®** are trademarks or registered trademarks of International Business Machines Corporation; Informix is currently developed and distributed by HCL under license. **PostgreSQL** is a trademark of the PostgreSQL Community Association. This project is **independent** and is **not** affiliated with, sponsored by, or endorsed by IBM, HCL or the PostgreSQL project. Product names are used solely for identification and interoperability (nominative fair use); no third-party logos are distributed.

Commercial support. Licensing, production support (with SLAs), custom development and consulting are available from **Airtool Technologies** — airtool.io · info@airtool.io. Support entitlements are defined by your commercial license or support agreement. Bundled third-party components retain their own licenses.