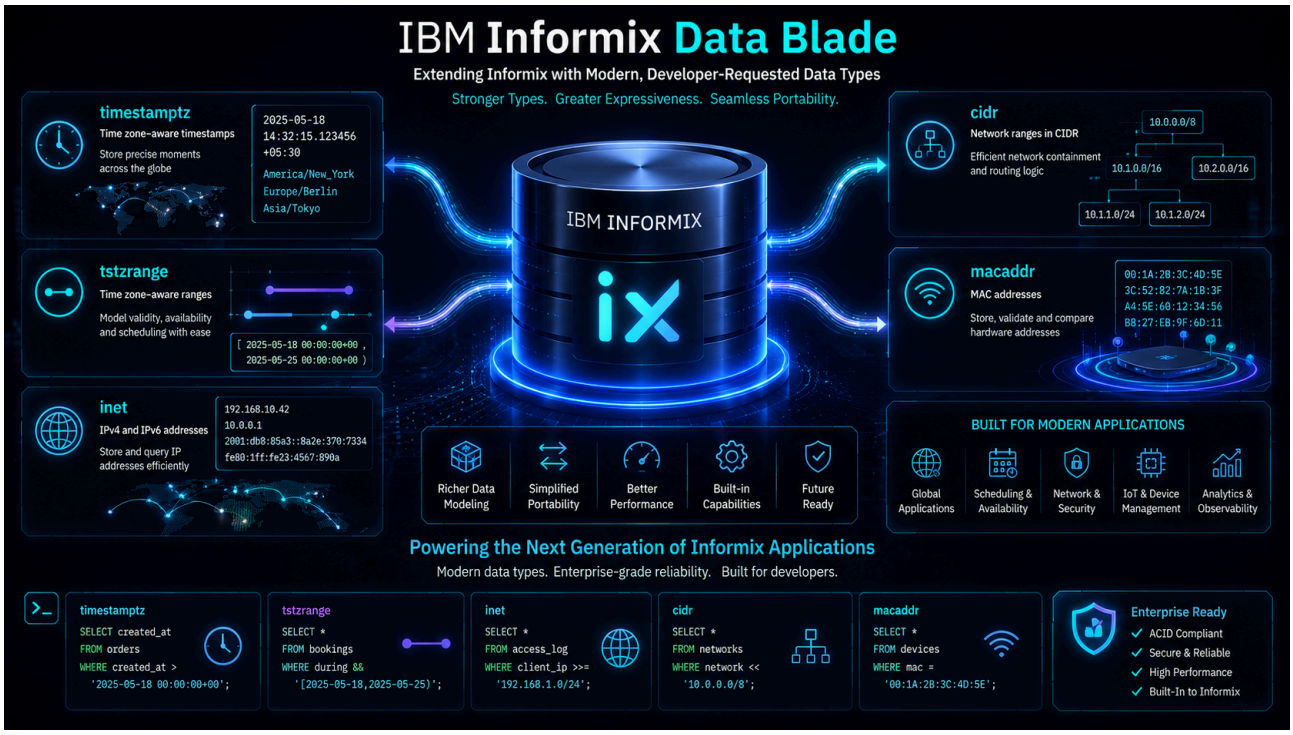
ifxtools

Document Version: 1.0 – 2026-08-13

Informix Extended-Types DataBlade

Airtool

Content



This document describes a family of **PostgreSQL-compatible data types** for IBM Informix: a set of C opaque user-defined types (UDTs) that fill the gaps where Informix has no native equivalent of a type PostgreSQL ships. Ten types are implemented today — the zone-aware instant `timestamptz`; the ranges `tstzrange`, `tstzmultirange`, `daterange`, `int8range` and `numrange`; the identifiers `uuid` and `citext`; and the network types `inet`, `cidr` and `macaddr`. Each carries the same name, the same text representation, the same ordering and the same operations as its PostgreSQL counterpart, so a schema and its queries can move between the two engines unchanged.

The defining property of this blade is not that the types exist, but that they are **proven equivalent**. Every behaviour is exercised by a single test that runs **identically on three engines** — Informix 14, Informix 15 and PostgreSQL — and asserts each against an engine-independent reference (a `java.time` computation or a fixed canonical form). PostgreSQL exercises its native type; Informix exercises the compiled UDT; both must agree.

> **Headline result.** A schema written for PostgreSQL — a `timestamptz` event log, a `tstzrange` booking calendar, an `inet` device inventory, a `citext` e-mail column, a `numrange` price band — creates and queries on Informix with the **same DDL and the same canonical text**. The cross-engine suite runs **129 invocations** (each test method × 3 engines) green: Informix 14, Informix 15 and PostgreSQL produce **identical** output, ordering and predicate results for every type.

1. Motivation — The Type-System Gap Between Informix and PostgreSQL

IBM Informix has a rich built-in type system, but a handful of types that application developers reach for on PostgreSQL have no native Informix equivalent. Code ported from PostgreSQL — or a single schema meant to run on both engines — needs those types to behave **the same way** on Informix as on PostgreSQL: same literals, same sort order, same containment and overlap semantics.

- **A zone-aware instant.** Informix has `DATETIME YEAR TO FRACTION(5)`, a `wall-clock` value with no notion of an absolute point in time. PostgreSQL's `timestamptz` stores an instant normalised to UTC. Ported code expects the latter semantics.
- **Range types.** Bookings, validity windows, contract terms, price bands and availability calendars are naturally `tstzrange` / `daterange` / `numrange` / multirange values with overlap and containment operators. Informix has none.
- **Network address types.** `inet` / `cidr` / `macaddr` with subnet-containment underpin infrastructure tooling and security analytics.
- **Identifiers.** `uuid` for distributed keys; `citext` for case-insensitive e-mails, usernames and SKUs.

The Informix DataBlade API lets us add such types **inside the engine** as opaque UDTs: they store, sort, index, compare and cast like first-class types, with the storage and behaviour defined in C. This blade collects them and holds each to one hard standard — it must behave like its PostgreSQL counterpart — enforced by a test that runs on both engines and asserts they agree.

> **Filling gaps, not duplicating.** Types Informix already ships — `MONEY`, `DECIMAL` / `NUMERIC`, native JSON/BSON — are deliberately **not** re-implemented. The blade adds only what is genuinely missing.

1.1. System architecture

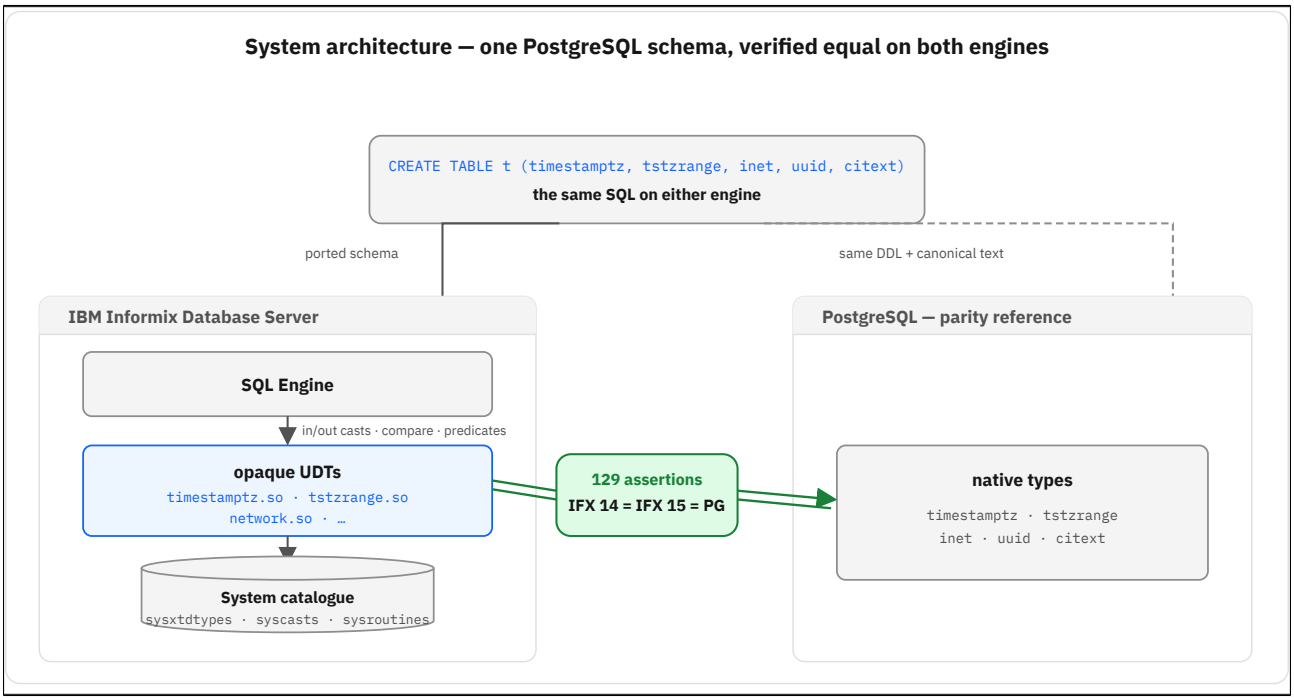


Figure 1 — System architecture — one PostgreSQL schema, verified equal on both engines

2. The Opaque-Type Model — How a PostgreSQL Type Becomes an Informix UDT

Each type is an **opaque UDT**: a fixed- or variable-length byte blob whose meaning lives entirely in its C support functions. A type is usable from SQL once it defines the mandatory support functions and binds the comparison operators.

2.1. Support functions and casts

Table 1 — The support-function contract every type implements

Function	Role	Registered as
`<type>_in(lvarchar)`	Parse the text literal into the internal form.	`CREATE IMPLICIT CAST (lvarchar AS <type>)` — so a quoted literal works directly.
`<type>_out(<type>)`	Render the internal form as canonical text.	`CREATE EXPLICIT CAST (<type> AS lvarchar)` — used by `value::lvarchar`.
`<type>_send / _recv`	Binary client↔server transfer (endian-canonical).	Casts to/from the `sendrecv` type.
`compare(<type>,<type>)`	Total order (−1/0/+1).	The B-tree strategy function — enables `ORDER BY`, `GROUP BY`, indexing.
`equal, notequal, lessthan, lessthanorequal, greaterthan, greaterthanorequal`	Relational tests.	Bound by **name** to the SQL operators `= <> < <= > >=`.

2.2. The operator constraint — why predicates are functions

Informix extensibility predates PostgreSQL's modern extension APIs. There is **no** `CREATE OPERATOR` for a UDT — only a fixed set of named functions (`compare`, `equal`, `lessthan`, …) bind to the built-in operators. PostgreSQL's symbolic range/network operators therefore have no Informix equivalent and are exposed as **named functions**:

Table 2 — PostgreSQL operator → Informix function

PostgreSQL	Informix (this blade)	Meaning
`a && b`	`range_overlaps(a,b)` / `mr_overlaps_range` / `daterange_overlaps` / `int8range_overlaps` / `numrange_overlaps`	Ranges overlap.
`a @> b`	`range_contains(a,b)` (and `_contains_point` / `_contains_date` / `_contains_elem` for an element)	Range contains range / element.
`a -|- b`	`range_adjacent(a,b)` (per range type)	Ranges abut without overlap.
`ip << net`	`network_sub(ip,net)` (and `_subeq` / `_sup` / `_supeq`)	Subnet containment.

The relational operators (`= < <= > >= <>`) **do** bind, so comparisons, sorting and B-tree indexing read identically on both engines.

2.3. Fixed vs variable length

Most types are fixed-length (declared with an `internallength` in bytes). `tstzmultirange` and `citext` are **variable-length** (`INTERNALLENGTH = VARIABLE`): their value is an `mi_lvarchar` whose data is read with `mi_get_vardata` / `mi_get_varlen` and built with `mi_new_var` — the same mechanism the Informix `vector` type uses for variable-dimension embeddings.

Table 3 — Internal representations

Type	Internal form	Length
`timestamptz`	int64 microseconds since 2000-01-01 UTC (PostgreSQL's integer-datetime encoding)	8 bytes
`tstzrange` / `daterange` / `int8range`	two int64 bounds (µs / day-number / integer) + flag byte	24 bytes
`numrange`	two IEEE-double bounds + flags	24 bytes
`tstzmultirange`	array of 24-byte range elements	variable
`uuid`	16 raw bytes	16 bytes
`citext`	original text bytes (case folded only for comparison)	variable
`inet` / `cidr`	family (4/6) + netmask bits + 16-byte address	18 bytes
`macaddr`	6 address bytes	6 bytes

3. The Type Catalogue

3.1. Temporal — timestamptz

A PostgreSQL-faithful `timestamp with time zone` : an **absolute instant** stored as int64 microseconds since 2000-01-01 00:00:00 UTC — byte-for-byte PostgreSQL's integer-datetime encoding. The zone normalises input and renders output; it is never stored. `±infinity` are the int64 extremes.

Table 4 — timestamptz functions

Function	Returns	PostgreSQL equivalent
`timestamptz_in / _out`	cast in/out	type I/O (implicit lvarchar→timestamptz)
`timestamptz_now()`	`timestamptz`	`clock_timestamp()`
`tstz_epoch(t)`	`FLOAT`	`extract(epoch from t)`
`tstz_at_utc(t)`	`LVARCHAR`	render in UTC
`tstz_at_zone(t, mins)`	`LVARCHAR`	render at a fixed numeric offset

3.2. Ranges — tstzrange, tstzmultirange, daterange, int8range, numrange

Range types model an interval between two bounds, each inclusive `[ ]` or exclusive `( )` , either possibly infinite, plus the distinguished `empty` . **Discrete** ranges (`daterange` , `int8range`) canonicalise to half-open `[lower,upper)` as PostgreSQL does; **continuous** ranges (`tstzrange` , `numrange`) keep their bounds as given. A **multirange** is an ordered set of non-overlapping, non-adjacent ranges.

Table 5 — Range function families (per type: <t>_overlaps / _contains / _adjacent / _lower / _upper / _isempty)

Type	Over	Kind	Element-containment fn
`tstzrange`	`timestamptz`	continuous	`range_contains_point(r, ts)`
`tstzmultirange`	set of `tstzrange`	variable length	`mr_contains_point(mr, ts)`
`daterange`	`DATE`	discrete	`daterange_contains_date(r, 'YYYY-MM-DD')`
`int8range`	`BIGINT`	discrete	`int8range_contains_elem(r, 'n')`
`numrange`	numeric	continuous	`numrange_contains_elem(r, 'v')`

`tstzrange` adds the constructors `tstzrange(lo, hi [, bounds])` and the bound-introspection functions `range_lower_inc` / `range_upper_inc` ; `tstzmultirange` adds `mr_overlaps` (multirange vs multirange), `mr_contains_range` , `mr_num_ranges` and the `tstzmultirange(range)` constructor.

3.3. Identifiers — uuid, citext

Table 6 — uuid and citext

Type	Behaviour	Notable function
`uuid`	16 bytes; liberal input (hyphenated / unhyphenated / braced, any case); canonical	`gen_random_uuid()` — version-4 random UUID (PostgreSQL's built-in name)

Type	Behaviour	Notable function
	lower-case hyphenated output; byte-wise ordering.	
citext	Case-insensitive text: stores the original bytes (display preserves case), folds case only for comparison/ordering/lookup.	the relational operators are case-insensitive (ASCII fold)

3.4. Network — inet, cidr, macaddr

IPv4 and IPv6 throughout (parsing/formatting via libc inet_pton / inet_ntop). inet is a host with optional netmask; cidr is a network with host bits zeroed; macaddr is a 6-byte MAC. An implicit cidr → inet cast lets either type flow into the network functions.

Table 7 — Network functions

Function	Returns	PostgreSQL equivalent
network_sub / _subeq / _sup / _supeq(a,b)	BOOLEAN	<< <<= >> >>=
inet_masklen(ip)	INTEGER	masklen(ip)
inet_family(ip)	INTEGER (4/6)	family(ip)
inet_host(ip)	LVARCHAR	host(ip)
inet_network(ip)	inet	network(ip)

4. Examples

Worked, self-contained cases. Every result below is reproduced by the project's automated cross-engine test suite on live Informix 14, Informix 15 and PostgreSQL containers — the numbers are real, not illustrative.

4.1. timestamptz — an instant, normalised to UTC

```
CREATE TABLE events (id INTEGER, occurred_at timestamptz);
INSERT INTO events VALUES (1, '2024-07-01 08:00:00+05:30');   -- India offset
INSERT INTO events VALUES (2, '2024-03-10 06:30:00-08');      -- US Pacific offset
INSERT INTO events VALUES (3, '2024-01-15T12:00:00Z');        -- ISO 8601 / Zulu

SELECT id, occurred_at::lvarchar AS utc FROM events ORDER BY occurred_at;
```

```
 id | utc
----+---------------------------
  3 | 2024-01-15 12:00:00+00
  2 | 2024-03-10 14:30:00+00      -- 06:30 −08:00  →  14:30 UTC
  1 | 2024-07-01 02:30:00+00      -- 08:00 +05:30  →  02:30 UTC

Stored as the same instant regardless of input zone; sorted by that instant.
```

`tstz_epoch()` matches PostgreSQL's `extract(epoch …)` — the type's internal zero is the PostgreSQL epoch itself:

```
SELECT FIRST 1 tstz_epoch(CAST('2000-01-01 00:00:00+00' AS timestamptz)) FROM
 systables;
```

```
 946684800.0      -- seconds from 1970-01-01 to the 2000-01-01 internal epoch
```

4.2. tstzrange — a booking calendar (overlap)

```
CREATE TABLE reservations (room_id INTEGER, booked tstzrange);
INSERT INTO reservations VALUES (1, '[2026-06-01 10:00,2026-06-01 12:00)');
INSERT INTO reservations VALUES (2, '[2026-06-01 12:00,2026-06-01 14:00)');
INSERT INTO reservations VALUES (3, '[2026-06-01 13:00,2026-06-01 15:00)');

-- PostgreSQL:  WHERE booked && '[2026-06-01 11:00,2026-06-01 13:00)'
SELECT room_id FROM reservations
 WHERE range_overlaps(booked, CAST('[2026-06-01 11:00,2026-06-01 13:00)' AS tstzrange))
 ORDER BY room_id;
```

```
┌─────────┐
│ room_id │
├─────────┤
│       1 │   -- 10:00–12:00 overlaps 11:00–13:00
│       2 │   -- 12:00–14:00 overlaps 11:00–13:00
└─────────┘
 Room 3 (13:00–15:00) does not: the probe's upper bound 13:00 is exclusive.
```

4.3. tstzmultirange — an availability calendar (normalisation)

Input is normalised exactly as PostgreSQL does: empty ranges dropped, the rest sorted and overlapping/adjacent ranges merged.

```
SELECT          CAST('{[2026-01-01,2026-01-10),[2026-01-05,2026-01-20)}'          AS
 tstzmultirange)::lvarchar
  FROM systables WHERE tabid = 1;
```

```
 {["2026-01-01 00:00:00+00","2026-01-20 00:00:00+00")}

 The two overlapping ranges merge into one canonical range.
```

4.4. daterange — a contract term (discrete canonicalisation)

```
CREATE TABLE contracts (id INTEGER, term daterange);
INSERT INTO contracts VALUES (1, '[2026-01-01,2026-06-30]');   -- inclusive end

SELECT term::lvarchar FROM contracts;
```

```
 [2026-01-01,2026-07-01)

 The inclusive end date 2026-06-30 is canonicalised to the exclusive next day,
 matching PostgreSQL's half-open daterange.
```

4.5. int8range / numrange — quantity tiers and price bands

```
-- discrete integer band: an inclusive [1,100] becomes half-open [1,101)
SELECT FIRST 1 CAST('[1,100]' AS int8range)::lvarchar FROM systables;
-- continuous price band: bounds kept as given
SELECT FIRST 1 CAST('[0,9.99]' AS numrange)::lvarchar FROM systables;
-- is 9.99 inside the budget band [0,10)?
SELECT  FIRST  1  numrange_contains_elem(CAST('[0,10)'  AS  numrange),  '9.99')  FROM
 systables;
```

```
 [1,101)
 [0,9.99]
 t
```

4.6. uuid — distributed keys

```
-- liberal input, canonical lower-case output; two spellings are equal
SELECT  FIRST  1  CAST('550E8400-E29B-41D4-A716-446655440000'  AS  uuid)::lvarchar  FROM
 systables;
SELECT FIRST 1 1 FROM systables
  WHERE  CAST('550E8400E29B41D4A716446655440000'  AS  uuid)  =  CAST('550e8400-e29b-41d4-
a716-446655440000' AS uuid);
SELECT FIRST 1 gen_random_uuid()::lvarchar FROM systables;
```

```
 550e8400-e29b-41d4-a716-446655440000
 1                                    -- equal regardless of case / hyphenation
 b3f1c2a8-4d6e-4f2a-9c71-2e5a8d0f1b34       -- a fresh version-4 UUID
```

4.7. citext — case-insensitive identifiers

```
CREATE TABLE accounts (id INTEGER, email citext);
INSERT INTO accounts VALUES (1, 'John@Example.com');

SELECT id, email::lvarchar FROM accounts WHERE email = 'john@example.com';
```

```
┌────┬──────────────────┐
│ id │ email            │
├────┼──────────────────┤
│  1 │ John@Example.com │  -- matched case-insensitively, displayed in original case
└────┴──────────────────┘
```

4.8. inet — subnet membership

```
CREATE TABLE devices (id INTEGER, ip inet);
INSERT INTO devices VALUES (1, '192.168.1.10');
INSERT INTO devices VALUES (2, '192.168.2.5');
INSERT INTO devices VALUES (3, '2001:db8::1');

-- PostgreSQL:  WHERE ip << '192.168.1.0/24'
SELECT id, ip::lvarchar FROM devices
 WHERE network_sub(ip, CAST('192.168.1.0/24' AS cidr)) ORDER BY id;
```

```
 id  ip
  1  192.168.1.10/32    -- inside 192.168.1.0/24
```

Device 2 is in a different /24; device 3 is IPv6. The ::lvarchar cast shows the mask like PostgreSQL's inet::text (/32 for a full host).

5. Cross-Engine Equivalence — The Verification Methodology

Parity is the product, so it is **tested, not asserted**. Each type has one JUnit class whose `@TestTemplate` methods are run **once per engine** — Informix 15, Informix 14 and PostgreSQL — by a multi-engine extension. A single test body executes on all three: on Informix it exercises the compiled UDT and its functions; on PostgreSQL it exercises the native type and operators. Small dialect helpers hide the difference.

```
// PostgreSQL:  booked && '[…)'::tstzrange       Informix:  range_overlaps(booked,
 CAST('[…)' AS tstzrange))
String overlaps(String a, String b) {
    return isPostgres() ? a + " && " + b
                        : "range_overlaps(" + a + ", " + b + ")";
}
```

Crucially, the expected value is an **engine-independent reference** — a hardcoded canonical form or a `java.time` computation, not "whatever PostgreSQL returned". A green run therefore means all three engines independently agree with the reference. Type names and table DDL are identical across engines; only the operator/function spelling and the text-cast keyword (`::text` vs `::lvarchar`) differ.

> **129 invocations green.** Across the ten types, every cross-engine assertion passes on Informix 14, Informix 15 and PostgreSQL — canonical text, ordering, overlap/containment, subnet membership, epoch extraction, UUID generation and case-insensitive matching all identical. Two invocations are skipped where there is no PostgreSQL counterpart (the numeric-offset `AT TIME ZONE` form; `mr_num_ranges`).

The hardest pure logic — the proleptic-Gregorian calendar, range bound comparison, discrete canonicalisation, IPv6 and MAC parsing — is additionally unit-tested **off the server** in plain C, before the slow compile-on-server cycle, so logic bugs are caught in milliseconds.

6. Known Differences from PostgreSQL

Deliberate, documented deltas within the current scope. The tests pin PostgreSQL to `TimeZone='UTC'` and use values inside each delta so the comparison is on equal footing.

Table 8 — Intentional deltas

Area	PostgreSQL	This blade
Zone-less `timestamptz` input	applies the session `TimeZone` GUC	interpreted as UTC (a UDR has no per-session GUC); deterministic
Named IANA zones / DST	full tz database	fixed numeric offsets only; named-zone support is a planned phase
Symbolic operators (`&&` , `@>` , `-|-` , `<<`)	operators	named functions (Informix has no `CREATE OPERATOR` for UDTs)
`AT TIME ZONE` output	returns a zone-less timestamp	`tstz_at_zone` includes the offset suffix (`…+05:30`)
`citext` folding	full Unicode by database locale	ASCII (A–Z); non-ASCII bytes compare verbatim
`numrange` bounds	arbitrary-precision `numeric`	IEEE double (~15–16 sig. digits); trailing-zero scale not preserved (`10.00` → `10`)

7. Architecture Comparison

Table 9 — Three ways to get PostgreSQL-style types on Informix

Dimension	This blade (opaque UDT)	Emulate with base types + CHECK	Application-side
Same DDL as PostgreSQL	**Yes** — same type name	No — column is VARCHAR/DECIMAL + constraints	No — type lives in code
Canonical text parity	**Yes** — verified 3-way	Manual, per query	Manual
Ordering / indexing	Native (`compare` + B-tree)	Lexical, often wrong (e.g. inet, uuid case)	Not in the database
Overlap / containment / subnet	In-engine functions	Hand-written SQL, error-prone	In application
Where logic lives	In the database, next to the data	Split between schema and queries	Outside the database
Best for	Schema/query portability and correctness	One-off columns with simple rules	Logic already centralised in one app

8. Limitations

8.1. No operator symbols

Range and network predicates are functions, not operators (`range_overlaps(a,b)` rather than `a && b`). Equality and ordering operators do bind, so `=`, `<`, `ORDER BY` and B-tree indexing work normally.

8.2. Precision and locale boundaries

`numrange` uses IEEE-double bounds (a decimal-backed, scale-preserving variant is a tracked future option), and `citext` folds ASCII case. Both are exact for the business data they target (price/quantity bands; e-mail/SKU identifiers); see the deltas table.

8.3. Time-zone scope

`timestamptz` handles fixed numeric offsets, which is exact and dependency-free. Named IANA zones with daylight-saving transitions require linking a tz database and are a deliberate future phase.

9. Compilation and Registration

Each type is C compiled to a shared object (`<type>.so`) and registered as a UDT plus its support functions and casts. In this project the compile happens **on the Informix server**: the install tooling (`InformixBladeModule`) streams the `.c` source over JDBC, writes it via `LOTOFILE` , and invokes gcc through an SPL `SYSTEM` call, then runs the registration SQL. The server therefore needs a C compiler on its `PATH` . The steps below are the manual equivalent.

9.1. Compiling the shared object

A standard DataBlade C-UDR compile: a position-independent shared object with the Informix public headers on the include path. Shared range logic lives in headers (`tstz_common.h` , `tstzrange_common.h`) that the compiler inlines into the streamed source.

```
gcc -O3 -fPIC -shared timestamptz.c  -I$INFORMIXDIR/incl/public -lm -o timestamptz.so
gcc -O3 -fPIC -shared tstzrange.c    -I$INFORMIXDIR/incl/public -lm -o tstzrange.so
gcc -O3 -fPIC -shared network.c      -I$INFORMIXDIR/incl/public      -o network.so
gcc -O3 -fPIC -shared uuid.c         -I$INFORMIXDIR/incl/public      -o uuid.so
```

9.2. Registering a type

The opaque type is created first, then its support functions bind to the `.so` entry points, then the casts. The path placeholder is the directory of the compiled `.so` (see each `<type>_install.sql`).

```
CREATE OPAQUE TYPE timestamptz (internallength = 8, alignment = 8);

CREATE FUNCTION timestamptz_in(lvarchar) RETURNING timestamptz
    EXTERNAL NAME '<sopath>/timestamptz.so(timestamptz_in)' LANGUAGE C;
CREATE FUNCTION timestamptz_out(timestamptz) RETURNING lvarchar
    EXTERNAL NAME '<sopath>/timestamptz.so(timestamptz_out)' LANGUAGE C;
CREATE IMPLICIT CAST (lvarchar AS timestamptz WITH timestamptz_in);
CREATE EXPLICIT CAST (timestamptz AS lvarchar WITH timestamptz_out);

CREATE FUNCTION compare(timestamptz, timestamptz) RETURNING INTEGER
    EXTERNAL NAME '<sopath>/timestamptz.so(compare)' LANGUAGE C NOT VARIANT;
-- equal / notequal / lessthan / lessthanorequal / greaterthan / greaterthanorequal …

-- A variable-length type only differs in the type declaration:
CREATE OPAQUE TYPE tstzmultirange (INTERNALLENGTH = VARIABLE, alignment = 8, MAXLEN
 = 32000);
```

9.3. Deployment prerequisites

An ordinary logged database. Even though the types store no smart blobs, the server-side compile streams the C source through a **CLOB temp table**, so the default `sbspace` must exist (the install otherwise fails with `"Invalid default sbspace name"`).

```
-- 1. C compiler on the Informix server (e.g. microdnf install -y gcc).
-- 2. Default sbspace present (onstat -d shows an 'S' space) — the compile uses a
 CLOB temp table.
-- 3. Create the logged database; the .c sources compile on the server and the types
 register.
CREATE DATABASE typesdb WITH LOG;
```

10. Glossary — Type and Range Terms

Terms specific to the PostgreSQL type model and to range/network types, for readers fluent in Informix. (Informix/DataBlade terminology is assumed known.)

Table 10 — Types glossary

Term	Definition
Opaque type (UDT)	A user-defined type whose internal byte representation is opaque to the engine; behaviour is supplied by C support functions.
Support function	One of the C routines a UDT must provide — input, output, send, receive, compare — bound by name so SQL can use the type.
Implicit / explicit cast	An implicit cast is applied automatically (so a quoted literal becomes the type); an explicit cast must be written (`value::lvarchar`).
compare strategy function	The `compare(t,t)→integer` that gives a total order, enabling `ORDER BY` , `GROUP BY` and B-tree indexing.
Variable-length opaque type	A UDT declared `INTERNALLENGTH = VARIABLE` ; its value is a varying buffer accessed with `mi_get_vardata` / `mi_get_varlen` .
timestamptz	An absolute instant stored in UTC (here int64 µs since 2000-01-01 UTC); the input zone normalises, output renders, the zone is not stored.
PostgreSQL epoch	2000-01-01 00:00:00 UTC — the zero point of the integer-datetime encoding both engines use here.
Range type	An interval between a lower and upper bound, each inclusive/exclusive, possibly infinite, plus the empty range.
Discrete vs continuous range	Discrete ranges (date, integer) canonicalise to half-open `[lo,upper)` ; continuous ranges (timestamptz, numeric) keep their bounds as given.
Canonicalisation	Normalising a range to a standard form — e.g. an inclusive integer/date upper bound becomes the exclusive next value.
Half-open interval	`[lo,hi)` — includes the lower bound, excludes the upper; the canonical discrete-range form.
Multirange	An ordered set of non-overlapping, non-adjacent, non-empty ranges; input is sorted and merged.
Overlap (`&&`)	Two ranges share at least one point.
Contains (`@>`)	A range includes another range, or a single element/ point.
Adjacent (`-|-`)	Two ranges abut exactly — one ends where the other begins — without overlapping or leaving a gap.

Term	Definition
Empty range	A range containing no points (e.g. `[5,5)`); sorts before every non-empty range.
inet	A host IP address (IPv4 or IPv6) with an optional netmask length.
cidr	A network specification — an address whose host bits are zero — written `network/prefix`.
Netmask length / prefix	The number of leading bits that identify the network (e.g. `/24`).
Subnet containment (`<<`)	An address or network falls within another network's address range.
macaddr	A 6-byte hardware (MAC) address, rendered in lower-case colon form.
uuid	A 128-bit universally unique identifier; this blade emits version-4 (random) UUIDs.
citext	Case-insensitive text — compared and ordered case-folded, displayed in the original case.
Cross-engine equivalence	The property — verified by running each test on Informix 14, Informix 15 and PostgreSQL — that the Informix UDT behaves identically to PostgreSQL's native type.

11. 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.