Expressive Representations of Missing Values in Relational Database Systems

Paolo Guagliardo | PostgresEDI meet-up | 13 August 2026

NULL values in SQL

Person

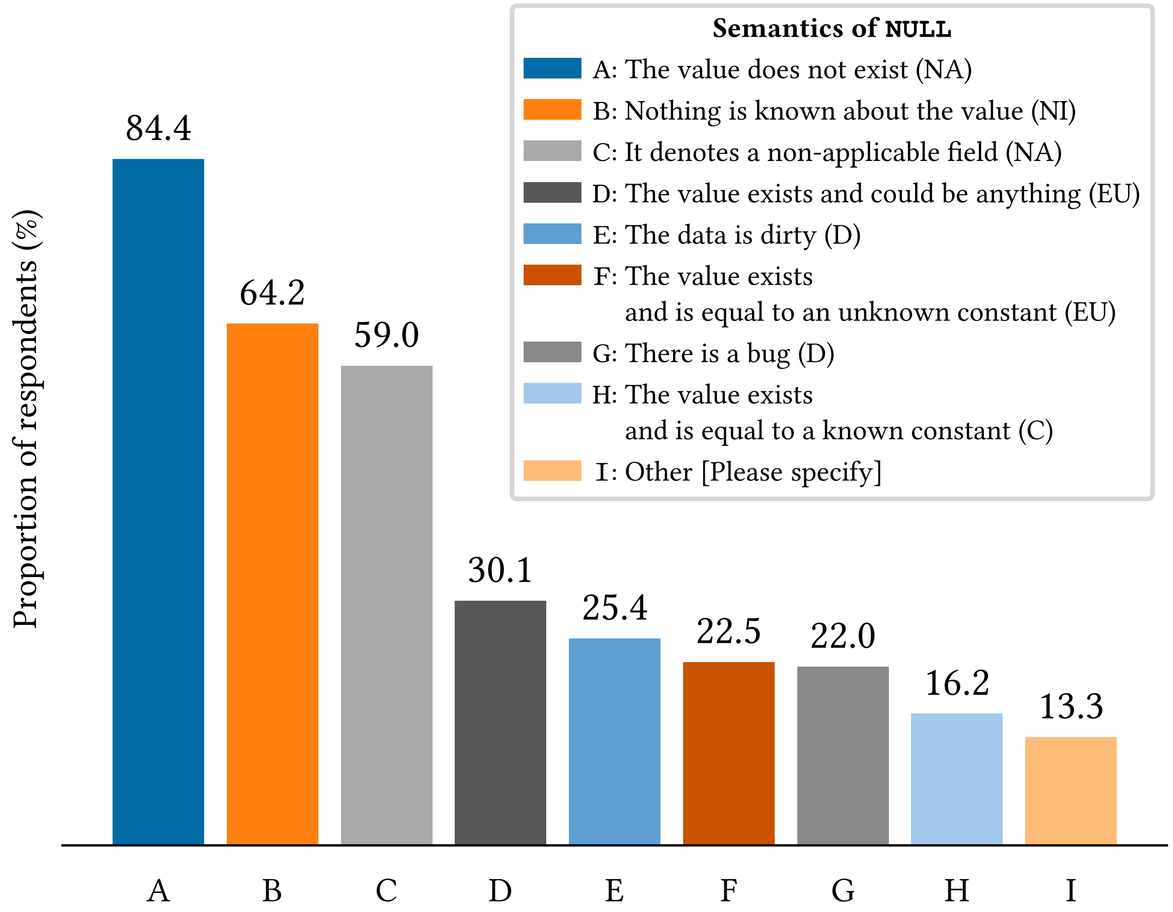
Name	Age
Jane	NULL
John	NULL
Mary	30
Carl	23

```
SELECT Name
FROM   Person
WHERE  Age = Age
```

Name
Mary
Carl

The age of John and Jane is **unknown**

What does NULL mean?



Toussaint, Guagliardo, Libkin, Sequeda: “Troubles with Nulls, Views from the Users”. PVLDB 2022

Existing-unknown or Non-applicable?

Person

ID	Name	Phone
1	Jane	NULL
2	John	NULL

Existing-unknown or Non-applicable?

Person

ID	Name	HasPhone	Phone
1	Jane	Yes	NULL ← Existing-unknown
2	John	No	NULL ← Non-applicable

Disambiguation flag

An excerpt from MusicBrainz

ARTIST

id	*name*	*begin_date*	*ended*	*end_date*
2832217	No Altars	2013-04-01	**TRUE**	**NULL**
2598081	Uppercult	2018-12-27	**TRUE**	2023-04-01
2273959	Maxx	1990-12-19	**FALSE**	**NULL**
2832169	Conviction	2014-11-04	**TRUE**	**NULL**

MusicBrainz

Log in Create account

Search Artist

About us Products Search Documentation English

Search results

Last updated: 2026-05-18 12:30 UTC

Found 1 result for "Maxx && Lambert"

Name	Sort name	Type	Gender	Area	Begin	Begin area	End	End area
Maxx (*Maxime Lambert*, Maxime Lambert)	Maxx	Person	Male	Belgium	1990-12-19	Liège		

Query: Maxx && Lambert

Type: Artist

Results per page: Up to 25

Search method:

- (•) Indexed search
- ( ) Indexed search with advanced query syntax
- ( ) Direct database search

Search

For more information, check the documentation.

The artist did not retire so there is no end date

https://musicbrainz.org/search?query=Maxx+%26%26+Lambert&type=artist

MusicBrainz

Log in Create account

Search | Artist

About us Products Search Documentation English

Search results

Last updated: 2026-05-18 12:27 UTC

Found 1 result for "Conviction && Arkansas"

Name	Sort name	Type	Gender	Area	Begin	Begin area	End	End area
Conviction (Hardcore band from Arkansas, US)	Conviction	Group		Bryant	2014-11-04		[unknown]	

Query: Conviction && Arkansas

Type: Artist

Results per page: Up to 25

Search method:
- (•) Indexed search
- ( ) Indexed search with advanced query syntax
- ( ) Direct database search

Search

For more information, check the documentation.

The artist retired but we don't know when

Donate Wiki Forums Chat Bug tracker Blog Mastodon Bluesky Use beta site

Brought to you by MetaBrainz Foundation and our sponsors and supporters.

https://musicbrainz.org/search?query=Conviction+%26%26+Arkansas&type=artist

Getting rid of non-applicable nulls

Person

ID	Name
1	Jane
2	John

PersonWithPhone

ID	Phone
1	`NULL`

PersonWithPhone(ID) `REFERENCES` Person(ID)

Exponential blow-up of the schema, possibly slow query performance

Limitations of SQL nulls as unknown values

Person

Name	Age
Jane	`NULL`
John	`NULL`
Mary	27
Carl	`NULL`

We cannot say that John and Carl have the **same unknown age**

Marked Nulls

Person

Name	Age
Jane	$\perp_1$
John	$\perp_2$
Mary	27
Carl	$\perp_3$

Essentially nulls with identifiers

The MusicBrainz example

with marked nulls

ARTIST

id	*name*	*begin_date*	*ended*	*end_date*
2832217	No Altars	2013-04-01	**TRUE**	**NULL**
2598081	Uppercult	2018-12-27	**TRUE**	2023-04-01
2273959	Maxx	1990-12-19	**FALSE**	**NULL**
2832169	Conviction	2014-11-04	**TRUE**	**NULL**

original

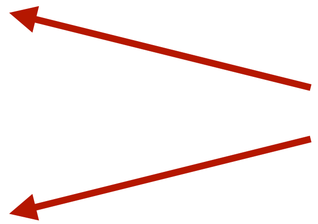
id	*name*	*begin_date*	*end_date*
2832217	No Altars	2013-04-01	$\perp_1$
2598081	Uppercult	2018-12-27	2023-04-01
2273959	Maxx	1990-12-19	**NULL**
2832169	Conviction	2014-11-04	$\perp_2$

with abstract marked nulls

Marked Nulls

Person

Name	Age
Jane	$\perp_1$
John	$\perp_2$
Mary	27
Carl	$\perp_2$



Allow for co-references!

Essentially nulls with identifiers

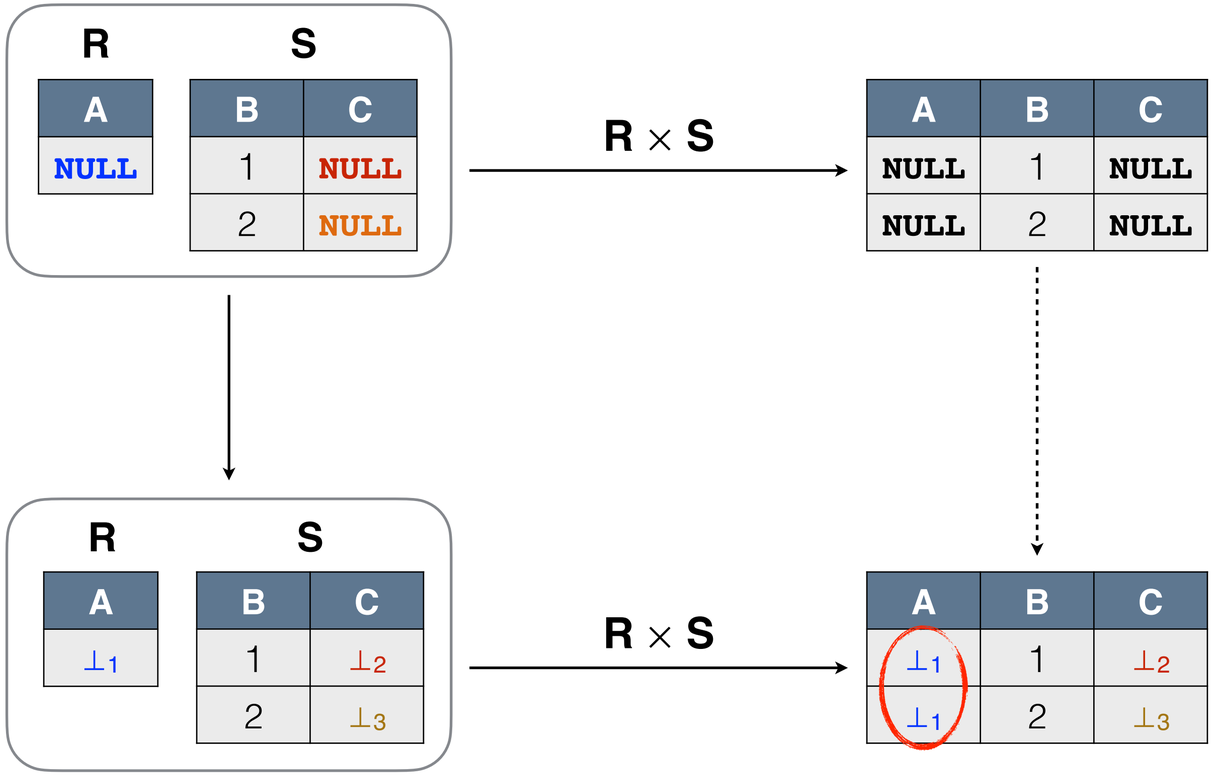
Non-repeating marked nulls

Person

Name	Age
Jane	$\perp_1$
John	$\perp_2$
Mary	27
Carl	$\perp_3$

No two marked nulls have the same identifier

Do SQL nulls model non-repeating marked nulls?



Guagliardo, Libkin: “On the Codd Semantics of SQL Nulls”. Inf. Syst. 86: 46-60 (2019)

SQL nulls are **not fit** for representing unknown values

Implementing marked types in real systems

Fixed-size

Base	Marked
`INT`	`INTT`
`DATE`	`DDATE`

Variable-size

Base	Marked
`TEXT`	`TEXTT`
`NUMERIC`	`NUMERICC`

Desiderata

- `NULL` and marked constants should behave exactly as for base types
- Marked nulls should comply with naive evaluation as follows:
  - two marked nulls are equal iff they have the same id
  - a marked null and a constant are never equal

Semantics

Casting rules

$\mathcal{T}$ = base type

$\widetilde{\mathcal{T}}$ = marked variant of $\mathcal{T}$

arrows denote casts { **single-head** = implicit; **double-head** = explicit

$\boxed{\tau \longrightarrow \widetilde{\tau}}$ lossless cast from base type to corresponding marked type

$\boxed{\widetilde{\tau} \twoheadrightarrow \tau}$ marked nulls are converted to **`NULL`**

$\boxed{\widetilde{\tau_1} \longrightarrow \widetilde{\tau_2}}$ if there is an underlying implicit cast from τ_1 to τ_2

Semantics

Comparisons

$$x = y \mapsto \begin{cases} \texttt{NULL} & \text{if } x \text{ is } \texttt{NULL} \text{ or } y \text{ is } \texttt{NULL}, \\ \texttt{TRUE} & \text{if } x \text{ and } y \text{ are non-}\texttt{NULL} \text{ and equal}, \\ \texttt{FALSE} & \text{otherwise}. \end{cases}$$

Standard SQL behaviour

\+

Naive evaluation

$$x < y \mapsto \begin{cases} \texttt{NULL} & \text{if } x \text{ is } \texttt{NULL} \text{ or } y \text{ is } \texttt{NULL}, \\ \texttt{TRUE} & \text{if } x, y \in \text{Const and } x < y \text{; or} \\ & \quad x = \bot_i \text{ and } y = \bot_j \text{ with } i < j \text{; or} \\ & \quad x \in \text{Const and } y \in \text{Nulls}; \\ \texttt{FALSE} & \text{otherwise}. \end{cases}$$

The reason for this is **indexing**, which requires a total order on non-**`NULL`** values

Semantics

Arithmetic and string operations

$\downarrow$ means converting to base type

$\uparrow$ means converting to marked type

$\oplus$ is an operator

Basic rule

$$x \oplus y = (x\downarrow \oplus y\downarrow)\uparrow$$

Exceptions

$$s \mid\mid \varepsilon = s \mid\mid \varepsilon = s \qquad v \,/\, 1 = v$$

$$v + 0 = 0 + v = v \qquad v \,/\, v = 1$$

$$v * 0 = 0 * v = 0 \qquad v - 0 = v$$

$$v * 1 = 1 * v = v \qquad v - v = 0$$

- These break distributivity but most DBMSs do not optimise arithmetic expressions
- The result may depend on how the query is written (this is not new)

Semantics

Aggregate functions

For a bag S of marked values

$$S{\downarrow} = \{x{\downarrow} \mid x \in S\}$$

MIN / **MAX**	follow the ordering defined by $<$ and $=$
COUNT(S)	counts non-**NULL** values (constants and marked nulls) in S
SUM(S)	same as **SUM**$(S{\downarrow})$
AVG(S)	same as **AVG**$(S{\downarrow})$

Note that **SUM** / **COUNT** is not the same as **AVG** !!!

The ratio $\dfrac{\textbf{COUNT}(S{\downarrow})}{\textbf{COUNT}(S)}$ can be used as a confidence score for **SUM** and **AVG**

What does the SQL Standard offer?

SQL allows for the creation of **ROW** types

User-defined composite types that are rows (tuples) of values

Can be forced to satisfy specific constraints

Flag-based encoding

Constant

flag is **`TRUE`**, value is non-**`NULL`**

Null ID

flag is **`FALSE`**, value is non-**`NULL`**

We need to rule out the case when both flag and value are **`NULL`**

(**`NULL`**, **`NULL`**) **`IS NULL`** evaluates to **`TRUE`** but it's typed differently from **`NULL`**

ID-based encoding

Constant

const is non-**NULL**, nullid is **NULL**

Null ID

const is **NULL**, nullid is non-**NULL**

```
CREATE TYPE τ′ AS (const τ, nullid INT);

CREATE DOMAIN τ̃ AS τ′ CHECK (
  VALUE IS DISTINCT FROM (NULL::τ,NULL::INT)
  AND ((VALUE).nullid IS NULL OR
       ((VALUE).nullid > 0 AND (VALUE).const IS NULL)
));
```

The MusicBrainz example again

but this time with SQL-based marked types

id	*name*	*begin_date*	*end_date*
2832217	No Altars	2013-04-01	$\perp_1$
2598081	Uppercult	2018-12-27	2023-04-01
2273959	Maxx	1990-12-19	**NULL**
2832169	Conviction	2014-11-04	$\perp_2$

with abstract marked nulls

```
     id      |       name       |  begin_date    |   end_date
-------------+------------------+----------------+---------------
 (2832217,)  |  ("No Altars",)  |  (2013-04-01,) |  (,1)
 (2598081,)  |  (Uppercult,)    |  (2018-12-27,) |  (2023-04-01,)
 (2273959,)  |  (Maxx,)         |  (1990-12-19,) |
 (2832169,)  |  (Conviction,)   |  (2014-11-04,) |  (,2)
```

https://git.ecdf.ed.ac.uk/pguaglia/markedsql

What works with Standard SQL features

Casts

- implemented via standard **`CREATE CAST`** command

Comparisons | string operations | arithmetic

- implemented as functions via standard **`CREATE FUNCTION`**
- SQL does not support user-defined operators
- we must write `eq(x,y)` rather than `x = y` ⟶ **Queries must be rewritten!**

Aggregates

- **`COUNT`** works out of the box
- **`MIN`** / **`MAX`** work consistently with < and > but only by chance!

Other limitations of Standard SQL

`AVG` / **`SUM`** should be explicitly defined on marked types

- SQL does not support user-defined aggregations
- workaround: **`AVG`**(**`CAST`** (*expr* **`AS`** *base_type*)) ⟶ **Queries must be rewritten!**

`MIN` / **`MAX`** work only because of how mark types are defined in PostgreSQL

- they break if we swap the **const** and **nullid** fields in our composite type
- they return the *record* pseudo-type (not usable in functions / casts)

  ⟹ type mismatches in comparisons with **`MIN`** / **`MAX`** subqueries

```
··· WHERE x = ( SELECT MIN(y) FROM ··· )
```

in which *x* and *y* are of the same marked type

Should we do away with portability?

Or sacrifice important functionality to maximise portability?

DBMS	UDTs	Row types	Low-level UDTs	User-defined operations	User-defined aggregations	Advanced features
PostgreSQL	✓	✓	✓ C	✓	✓	✓
Oracle Database	✓	✓	✓ C/C++/Java	✗	✗	✗
MS SQL Server	✓	✗	✓ IL (.NET)	✓	✗	✗
IBM DB2	✓	✗	✗	✓	✗	✗
MariaDB	✗ †	✗ †	✗	✗	✗	✗
MySQL	✗	✗	✗	✗	✗	✗
SQLite	✗	✗	✗	✗	✗	✗

No widespread support even for essential features (UDTs and row types)

Non-standard SQL-based implementation

in PostgreSQL

https://git.ecdf.ed.ac.uk/pguaglia/markedsql

Custom operators and aggregate functions

- solve the problems with **`AVG`** / **`SUM`** / **`MIN`** / **`MAX`**

Interfacing marked types with indexes and access methods

- enables hash and sort-merge joins, parallel execution, partial aggregation

Non-standard SQL-based implementation

in PostgreSQL

https://git.ecdf.ed.ac.uk/pguaglia/markedsql

Limitations

Explicit casts are needed to avoid type mismatch errors in some cases

- x = `'abc'` fails when x is of type **`TEXTT`**

  $\Longrightarrow$ we must write x = `'abc'::TEXT`

- **`CASE WHEN`** $\cdots$ **`THEN`** x **`ELSE`** `0` **`END`** fails when x is of type **`NUMERICC`** or **`INTT`**

  $\Longrightarrow$ we must explicitly cast the scalar 0 to x's type

- Set operations fail for the same reason: **`SELECT`** `1::INTT` **`UNION SELECT`** `0`

→ **query rewriting is unavoidable**

SQL-based marked types

Estimating storage requirements

Worst-case scenario

- All base types converted to corresponding marked types
- No marked nulls
- Same information content

Each marked value requires a 24-byte header (including a 1B null bitmap)

Thanks to TOAST (*The Oversized-Attribute Storage Technique*)

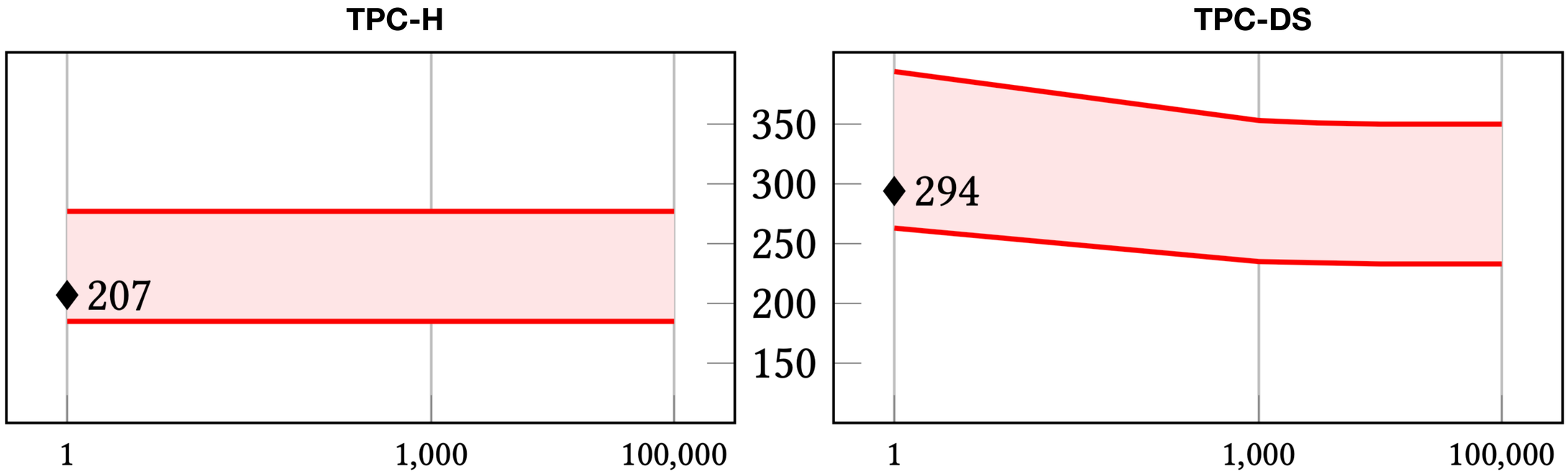
- the header is reduced by 3 bytes for data up to 105 bytes
- up to 5 bytes of padding may also be ammortized in such cases

Overhead per table is between $16m \cdot n$ and $24m \cdot n$ bytes

with m = # of rows and n = # of columns

Heap size overhead for SQL-based marked types

on TPC instances at different scale factors



The overhead here is measured in %

The red-shaded part is the estimated overhead

The diamond is the effective measured overhead

Measured overhead in size of indexes

on TPC instances with scale factor 1

TPC-H

218%

TPC-DS

226%

Total (measured) overhead in database size

on TPC instances with scale factor 1

TPC-H

209%

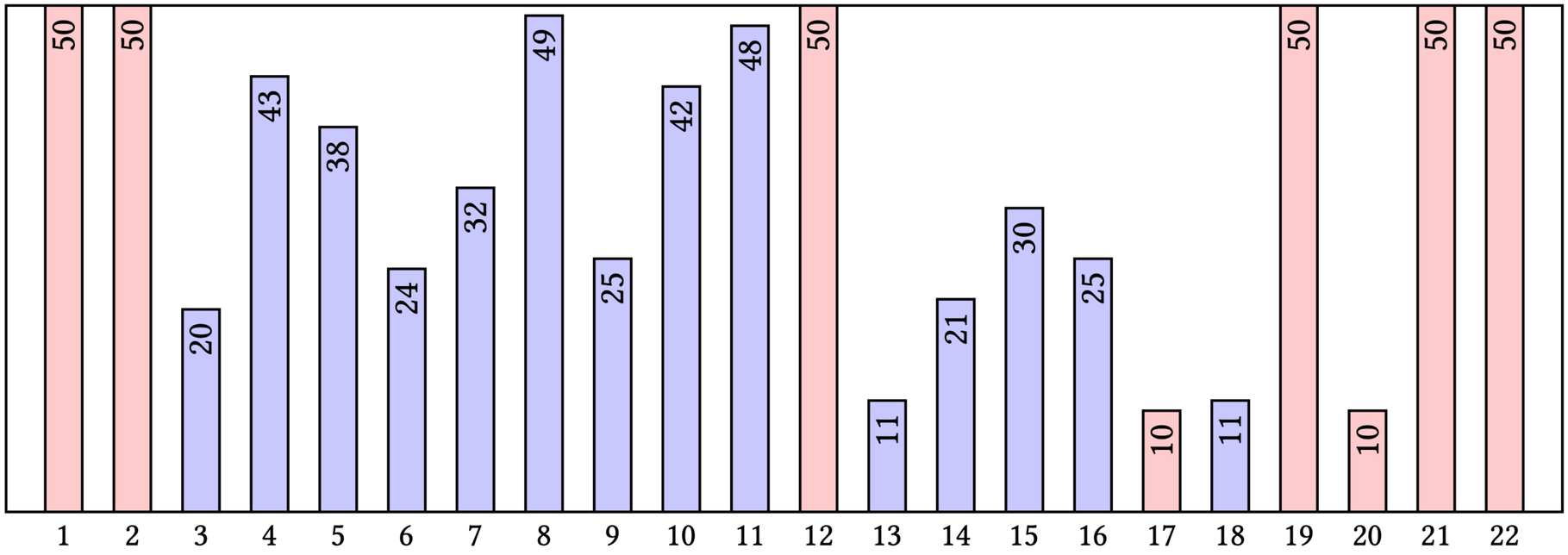
TPC-DS

277%

The size of the database more than triples!

Runtime overhead for SQL-based marked types

on TPC-H instances with scale factor 1



The overhead here is measured as the ratio $\frac{\text{runtime on marked instances}}{\text{runtime on base instances}}$

Red bars indicate timeout

SQL-based marked types

Summary

- Huge storage overhead
- Very poor query performance
- Not exactly “user-friendly”
- Portability is not guaranteed

Standard SQL features are **not fit** for implementing marked types

An efficient implementation

in PostgreSQL

https://git.ecdf.ed.ac.uk/pguaglia/ids4nulls

id	*name*	*begin_date*	*end_date*
2832217	No Altars	2013-04-01	$\perp_1$
2598081	Uppercult	2018-12-27	2023-04-01
2273959	Maxx	1990-12-19	**NULL**
2832169	Conviction	2014-11-04	$\perp_2$

```
    id    |     name     | begin_date |  end_date
----------+--------------+------------+------------
 2832217  | No Altars    | 2013-04-01 | NULL:1
 2598081  | Uppercult    | 2018-12-27 | 2023-04-01
 2273959  | Maxx         | 1990-12-19 |
 2832169  | Conviction   | 2014-11-04 | NULL:2
```

Fixed-size types

`INTT` (marked variant of a 4-byte **`INT`**)

We use the highest order bit as a flag

- 🙂 same size (4 bytes) ⟹ no storage overhead
- 🙂 fits in a datum ⟹ can be passed by value (fast)
- 🙁 half the range ⟹ casts from **`INT`** may overflow

The binary representations of non-negative integer constants coincide

On a big-endian machine:

`01111111 11111111 11111111 11111111`	is -1 as an **`INTT`**
`11111111 11111111 11111111 11111111`	is -1 as an **`INT`**
`11111111 11111111 11111111 11111111`	is an **`INTT`** marked null with id 2147483647

Fixed-size types

`DDATE` (marked variant of a **`DATE`**)

As for **`INTT`** we use the highest order bit as a flag

- 🙂 same size (4 bytes) ⟹ no storage overhead
- 🙂 fits in a datum ⟹ can be passed by value (fast)
- 🙂 full range of dates fits in 31 bits

Variable-size types

`TEXTT` (marked variant of **`TEXT`**)

Pure *varlena* type = header + raw data (flexible member array)

We append one byte to the raw data as the flag ⟹ 😕 1-byte overhead

- If the highest-order bit of the flag is set
  ⟹ raw data is a 31-bit null ID stored in reverse-byte order
- Otherwise, the raw data (except the last byte) is a character string

Variable-size types

`TEXTT` (marked variant of **`TEXT`**)

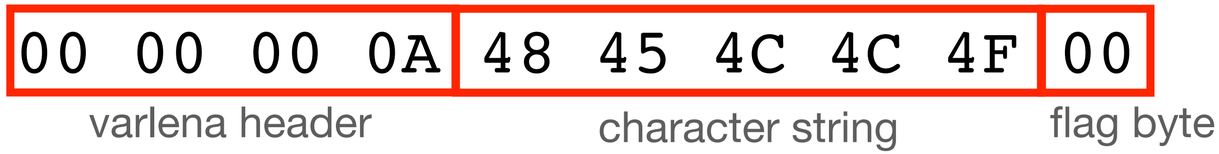
Marked null with ID 287346378

raw data in reverse order `91 20 8E CA`

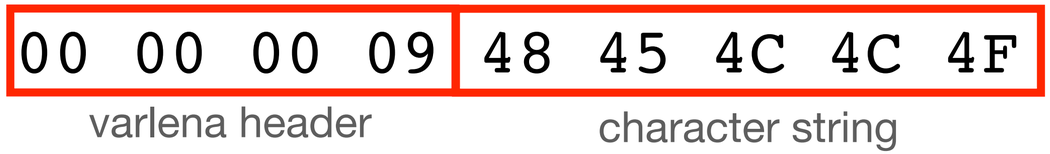
clear bit 32 to get null ID `11 20 8E CA` = 287346378

ASCII string “HELLO”

`TEXTT`



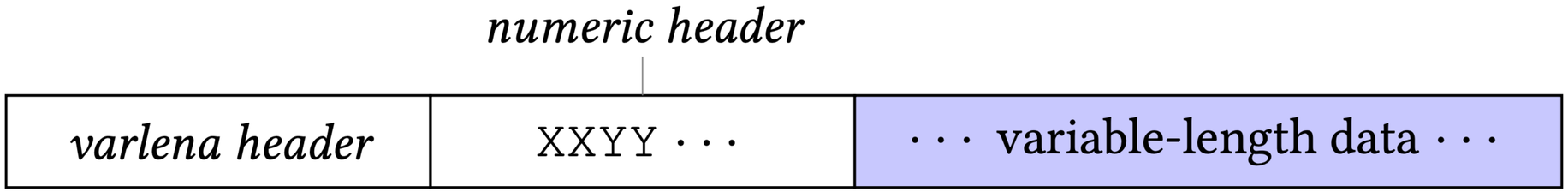
`TEXT`



Variable-size types

`NUMERICC` (marked variant of **`NUMERIC`**)

varlena type with an additional numeric header

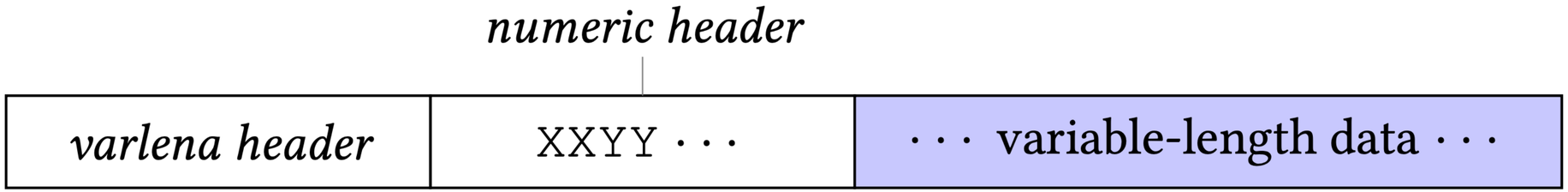


`XX`
`00` means positive number
`01` means negative number
`10` short numeric
`11` special value

`YY` (when `XX` = `11`)
`00` Not-a-Number (NaN)
`01` negative infinity
`11` positive infinity
`10` **not used** ⟹ we hijack it for the flag!

Variable-size types

`NUMERICC` (marked variant of **`NUMERIC`**)



- If the 4 highest-order bits of the numeric header are 1110
  ⟹ the flexible member array is a null ID (4-byte unsigned integer)
- Otherwise
  ⟹ we have a standard numeric constant (including NaN and infinity)

🙂 no overhead

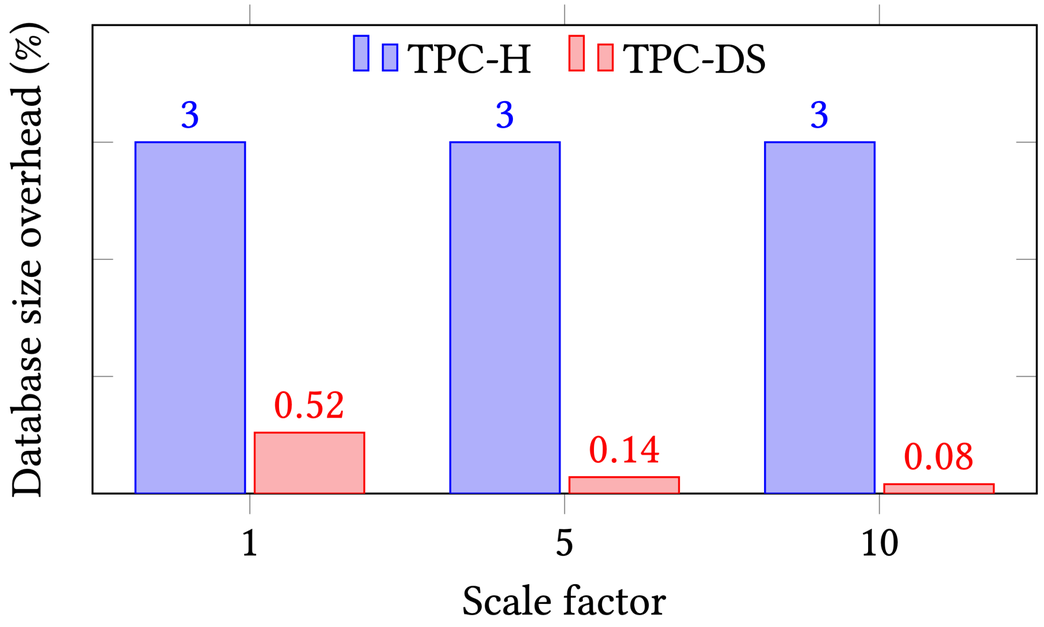
🙂 piggy-back on **`NUMERIC`** for constants

Storage requirements

Theoretical analysis

Proposition *In the worst case, the overhead with respect to table size introduced by the marked types of our ids4nulls extension never exceeds 100%, and it converges to 50% as the number n of rows increases. Moreover, it is zero when $0 \leq n \leq t$ and $1.5t < n \leq 2t$, where t is the number of marked records that fit on a page.*

Storage overhead on TPC instances



Individual table statistics for TPC-H

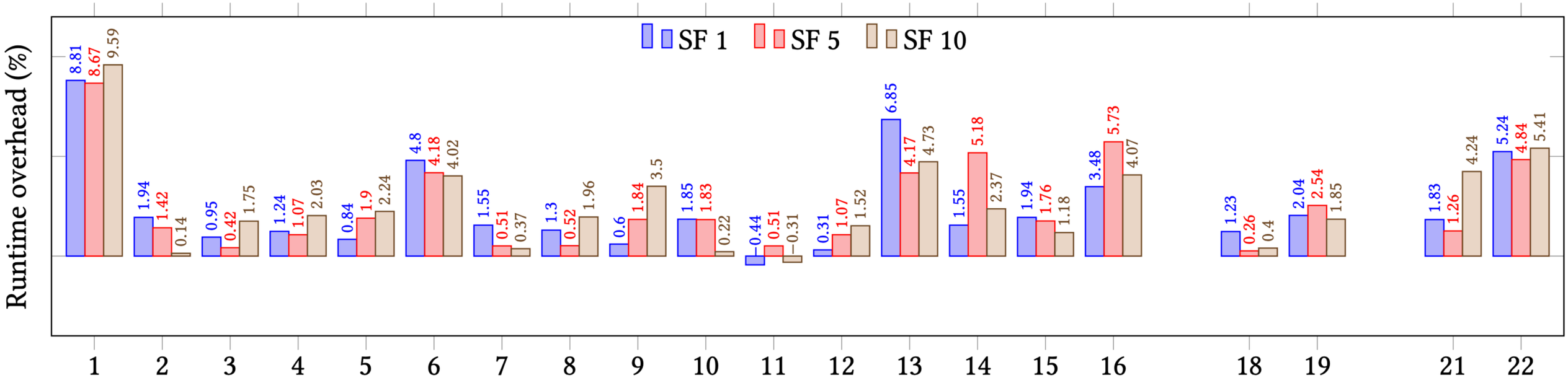
table	db %	cols	TEXT	overhead (%)		
				heap	indexes	total
lineitem	66.2	16	5	4.4	0.0	3.8
orders	16.9	9	4	1.9	0.0	1.6
partsupp	11.8	5	1	0.6	0.0	0.5
part	2.5	9	6	4.0	0.0	3.5
customer	2.4	8	5	2.7	0.0	2.4
supplier	0.1	7	4	2.4	0.0	2.1
nation	< 0.1	4	2	0.0	0.0	0.0
region	< 0.1	3	2	0.0	0.0	0.0

Storage overhead on the IMDB dataset

table	db %	cols	TEXT	overhead (%)		
				heap	indexes	total
cast_info	37.58	7	1	0.66	0	0.33
movie_info	24.11	5	2	2.38	0	1.65
person_info	8.31	5	2	0.75	0	0.6
name	8.27	9	7	5.42	0	4.5
char_name	5.42	7	5	4.21	0	3.41
title	5.35	12	5	1.83	0	1.46
movie_keyword	3.64	3	0	0	0	0
movie_companies	2.8	5	1	0.54	0	0.28
aka_name	1.7	8	6	4.47	0	3.19
movie_info_idx	1.18	5	2	6.54	0	3.31
aka_title	0.92	12	5	2.89	0	2.16
company_name	0.45	7	5	4.67	0	3.84
keyword	0.14	3	2	3.56	0	2.57

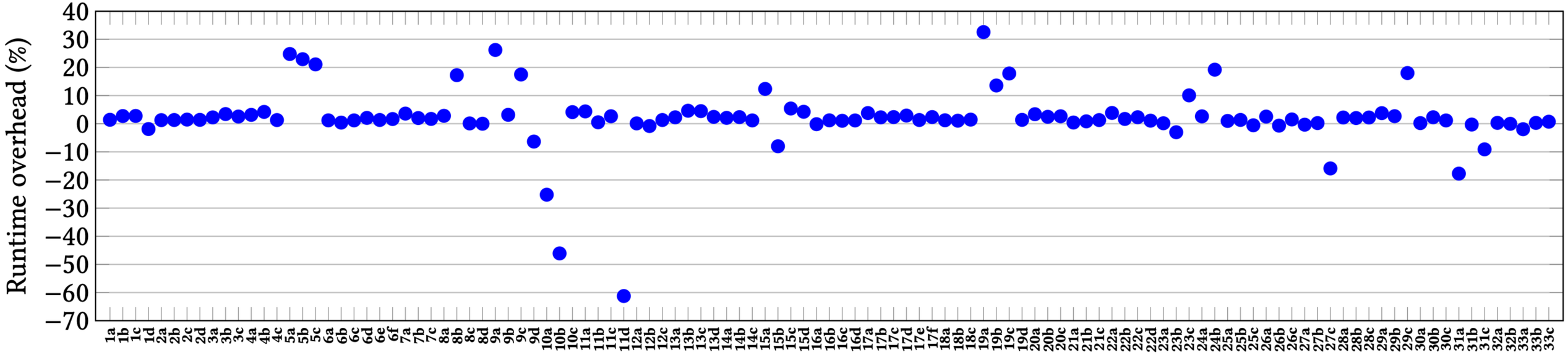
The overall overhead in database size is 1.2%

Runtime overhead on TPC-H benchmark



Query plans are the same as or very similar to those generated on base instances

Runtime overhead on IMDB/JOB



Future work

- **Reminder to myself**: finish experiments!
- Case study: computation of certain answers
- Usability study with real database users
- Specialised statistics and index structures for marked values
- Handle constraints (starting with Primary Keys and Foreign Keys)

About Me

Dr Paolo Guagliardo

paolo.guagliardo@ed.ac.uk

- Reader in Databases
  School of Informatics, University of Edinburgh
- Member of BSI IST/40
- UK expert on ISO JTC1/SC32/WG3

Teaching

- undergraduate course on databases

Research

- incomplete information
- semantics of query languages

Consulting

- RelationalAI Inc.