# Non-Standard-Datenbanken und Data Mining

SQL: New Developments

Prof. Dr. Ralf Möller

**Universität zu Lübeck**

**Institut für Informationssysteme**

# Übersicht

- Semistrukturierte Datenbanken (JSON, XML) und Volltextsuche
- Information Retrieval
- Mehrdimensionale Indexstrukturen
- Cluster-Bildung
- Einbettungstechniken
- First-n-, Top-k-, und Skyline-Anfragen
- Probabilistische Datenbanken, Anfragebeantwortung, Top-k-Anfragen und Open-World-Annahme
- Probabilistische Modellierung, Bayes-Netze, Anfragebeantwortungsalgorithmen, Lernverfahren,
- Temporale Datenbanken und das relationale Modell, SQL:2011
- Probabilistische Temporale Datenbanken
- SQL: neue Entwicklungen (z.B. JSON-Strukturen und Arrays), Zeitreihen (z.B. TimeScaleDB)
- Stromdatenbanken, Prinzipien der Fenster-orientierten inkrementellen Verarbeitung
- Approximationstechniken für Stromdatenverarbeitung, Stream-Mining
- Probabilistische raum-zeitliche Datenbanken und Stromdatenverarbeitungsssysteme: Anfragen und Indexstrukturen, Raum-zeitliches Data Mining, Probabilistische Skylines
- Von NoSQL- zu NewSQL-Datenbanken, CAP-Theorem, Blockchain-Datenbanken

# SQL Standard – Brief History

- SO/IEC 9075 Database Language SQL
  - SQL-86 – Transactions, Create, Read, Update, Delete
  - SQL-89 – Referential Integrity
  - SQL-92 – Internationalization, etc.
  - SQL:1999 – User Defined Types
  - SQL:2003 – XML
  - SQL:2008 – Expansions and corrections
  - SQL:2011 – Temporal
  - SQL:2016 – JSON, PTFs (Polymorphic Table Functions), RPR (Row-Pattern Recognition)
  - SQL:2019 – MDA
- 30+ years of support and expansion of the standard

# SQL:2016 New Features

- Support for Java Script Object Notation (JSON)
  - Store, query, and retrieve JSON structures
- Polymorphic table functions
  - Parameters and function return values can be tables whose shape is not known until query time
- Row pattern recognition
  - Regular Expressions across sequences of rows
- Additional functions (e.g., for analytics)
  - Trigonometric and logarithm functions
  - Concatenate strings over groups of rows
- Default values and names for arguments of SQL functions

# Acknowledgements

- Parts of subsequent presentations are taken from

  J. Michels et al., The New and Improved SQL:2016 Standard, SIGMOD Record, Vol. 47, No. 2, pp. 51-60, **2018**

# Support for Java Script Object Notation

| ID | JCOL |
| --- | --- |
| 111 | { "Name" : "John Smith", "address" : { "streetAddress": "21 2nd Street", "city": "New York", "state" : "NY", "postalCode" : 10021 }, "phoneNumber" : [ { "type" : "home", "number" : "212 555-1234" }, { "type" : "fax", "number" : "646 555-4567" } ] } |
| 222 | { "Name" : "Peter Walker", "address" : { "streetAddress": "111 Main Street", "city": "San Jose", "state" : "CA", "postalCode" : 95111 }, "phoneNumber" : [ { "type" : "home", "number" : "408 555-9876" }, { "type" : "office", "number" : "650 555-2468" } ] } |
| 333 | { "Name" : "James Lee" } |

## JSON Type

```
CREATE TABLE T (
  Id INTEGER PRIMARY KEY,
  Jcol CHARACTER VARYING ( 5000 )
           CHECK ( Jcol IS JSON ) )

SELECT * FROM T WHERE Jcol IS JSON
```

## JSON Path Expressions

```
SELECT Id
FROM T
WHERE JSON_EXISTS ( Jcol,
        'strict $.address' )
```

Find 111 and 222 but not 333 (due to strict)

## Extract scalars

```
SELECT Id, JSON_VALUE ( Jcol,
   'lax $.phoneNumber[0].number' )
   AS Firstphone
FROM T
```

Lax is default

Find 111, 222 with respective numbers, and 333 with null (due to lax)

# Support for Java Script Object Notation

| ID | JCOL |
| --- | --- |
| 111 | { "Name" : "John Smith", "address" : { "streetAddress": "21 2nd Street", "city": "New York", "state" : "NY", "postalCode" : 10021 }, "phoneNumber" : [ { "type" : "home", "number" : "212 555-1234" }, { "type" : "fax", "number" : "646 555-4567" } ] } |
| 222 | { "Name" : "Peter Walker", "address" : { "streetAddress": "111 Main Street", "city": "San Jose", "state" : "CA", "postalCode" : 95111 }, "phoneNumber" : [ { "type" : "home", "number" : "408 555-9876" }, { "type" : "office", "number" : "650 555-2468" } ] } |
| 333 | { "Name" : "James Lee" } |

## Filters

```
SELECT Id, JSON_VALUE ( Jcol,
   'lax $.phoneNumber
      ? ( @.type == "fax" ).number' )
    AS Fax
FROM T
```

Find 111 with number, 222 with null, and 333 with null
Return type is string if not specified otherwise

## Extract JSON fragments (objects, arrays, scalars)

```
SELECT Id, JSON_QUERY ( Jcol,
   'lax $.address' ) AS Address
FROM T
```

| ID | ADDRESS |
| --- | --- |
| 111 | { "streetAddress": "21 2nd Street", "city": "New York", "state" : "NY", "postalCode" : 10021 } |
| 222 | { "streetAddress": "111 Main Street", "city": "San Jose", "state" : "CA", "postalCode" : 95111 } |
| 333 | |

# Support for Java Script Object Notation

| ID | JCOL |
|---|---|
| 111 | { "Name" : "John Smith", "address" : { "streetAddress": "21 2nd Street", "city": "New York", "state" : "NY", "postalCode" : 10021 }, "phoneNumber" : [ { "type" : "home", "number" : "212 555-1234" }, { "type" : "fax", "number" : "646 555-4567" } ] } |
| 222 | { "Name" : "Peter Walker", "address" : { "streetAddress": "111 Main Street", "city": "San Jose", "state" : "CA", "postalCode" : 95111 }, "phoneNumber" : [ { "type" : "home", "number" : "408 555-9876" }, { "type" : "office", "number" : "650 555-2468" } ] } |
| 333 | { "Name" : "James Lee" } |

JSON to tables

```
SELECT T.Id, Jt.Name, Jt.Zip
FROM T,
  JSON_TABLE ( T.Jcol, 'lax $'
    COLUMNS (
      Name VARCHAR ( 30 )
           PATH 'lax $.Name'
      Zip  VARCHAR ( 5 ) PATH
           'lax $.address.postalCode'
      )
   ) AS Jt
```

| ID | NAME | ZIP |
|---|---|---|
| 111 | John Smith | 10021 |
| 222 | Peter Walker | 95111 |
| 333 | James Lee | |

Also used for unnesting (deeply) nested JSON structures or arrays

# Support for Java Script Object Notation

| ID | JCOL |
|---|---|
| 111 | { "Name" : "John Smith", "address" : { "streetAddress": "21 2nd Street", "city": "New York", "state" : "NY", "postalCode" : 10021 }, "phoneNumber" : [ { "type" : "home", "number" : "212 555-1234" }, { "type" : "fax", "number" : "646 555-4567" } ] } |
| 222 | { "Name" : "Peter Walker", "address" : { "streetAddress": "111 Main Street", "city": "San Jose", "state" : "CA", "postalCode" : 95111 }, "phoneNumber" : [ { "type" : "home", "number" : "408 555-9876" }, { "type" : "office", "number" : "650 555-2468" } ] } |
| 333 | { "Name" : "James Lee" } |

## JSON to tables

```
SELECT T.Id, Jt.Name, Jt.Type,
       Jt.Number
FROM T,
  JSON_TABLE ( T.Jcol, 'lax $'
    COLUMNS
     ( Name VARCHAR ( 30 )
             PATH 'lax $.Name',
       NESTED PATH
            'lax $.phoneNumber[*]'
       COLUMNS
         ( Type VARCHAR ( 10 )
              PATH 'lax $.type',
           Number VARCHAR ( 12 )
              PATH 'lax $.number' )
     )
  ) AS Jt
```

| ID | NAME | TYPE | NUMBER |
|---|---|---|---|
| 111 | John Smith | home | 212 555-1234 |
| 111 | John Smith | fax | 646 555-4567 |
| 222 | Peter Walker | home | 408 555-9876 |
| 222 | Peter Walker | office | 650 555-2468 |
| 333 | James Lee | | |

# Support for Java Script Object Notation

| ID | JCOL |
|---|---|
| 111 | { "Name" : "John Smith", "address" : { "streetAddress": "21 2nd Street", "city": "New York", "state" : "NY", "postalCode" : 10021 }, "phoneNumber" : [ { "type" : "home", "number" : "212 555-1234" }, { "type" : "fax", "number" : "646 555-4567" } ] } |
| 222 | { "Name" : "Peter Walker", "address" : { "streetAddress": "111 Main Street", "city": "San Jose", "state" : "CA", "postalCode" : 95111 }, "phoneNumber" : [ { "type" : "home", "number" : "408 555-9876" }, { "type" : "office", "number" : "650 555-2468" } ] } |
| 333 | { "Name" : "James Lee" } |

## Structural inspection

```
strict $.* ? ( @.type() == "array"
             && @.size() > 1 )
```

## Tables to JSON objects

```
SELECT JSON_OBJECT
 ( KEY 'department' VALUE D.Name,
   KEY 'employees'
    VALUE JSON_ARRAYAGG
          ( JSON_OBJECT
            ( KEY 'employee'
               VALUE E.Name,
              KEY 'salary'
               VALUE E.Salary )
             ORDER BY E.Salary ASC )
 ) AS Department
FROM Departments D, Employees E
WHERE D.Dept_id = E.Dept_id
GROUP BY D.Name
```

| DEPARTMENT |
|---|
| { "department" : "Sales", "employees" : [ { "employee" : "James", "salary" : 7000 }, { "employee" : "Rachel", "salary" : 9000 }, { "employee" : "Logan", "salary" : 10000 } ] } |
| … |

# Polymorphic Table Functions (PTFs)

- Reading a CSV file returns a table with structure unknown at compile time

```
SELECT *
FROM TABLE
    ( CSVreader (
        File => 'abc.csv',
        Floats => DESCRIPTOR
         ( "principal", "interest" )
        Dates => DESCRIPTOR
         ( "due_date" ) ) ) AS S
```

There must be column names „principal“ and „interest“ as well as a „due_date“ column

```
CREATE FUNCTION CSVreader (
      File VARCHAR(1000),
      Floats DESCRIPTOR DEFAULT NULL,
      Dates DESCRIPTOR DEFAULT NULL )
RETURNS TABLE
NOT DETERMINISTIC CONTAINS SQL
PRIVATE DATA ( FileHandle INTEGER )
DESCRIBE WITH PROCEDURE
         CSVreader_describe
START WITH PROCEDURE
         CSVreader_start
FULFILL WITH PROCEDURE
         CSVreader_fulfill
FINISH WITH PROCEDURE
         CSVreader_finish
```

Different results with same input parameters
Code supplied with SQL stored procedures

DESCRIBE: Determine row type
START: Allocate resources
FULFILL: Read a tuple
FINISH: Deallocate resources

# Polymorphic Table Functions (PTFs)

- User-defined joins

```
CREATE FUNCTION UDJoin
 ( T1 TABLE PASS THROUGH
           WITH SET SEMANTICS
           PRUNE WHEN EMPTY,
   T2 TABLE PASS THROUGH
           WITH SET SEMANTICS
           KEEP WHEN EMPTY
 ) RETURNS ONLY PASS THROUGH
```

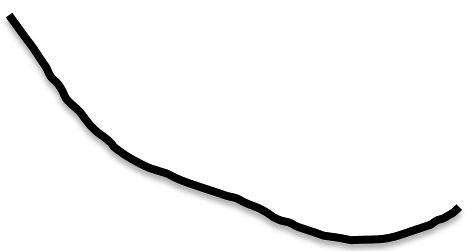
```
SELECT E.*, D.*
FROM TABLE
     ( UDJoin (
           T1 => TABLE (Emp) AS E
                 PARTITION BY Deptno,
           T2 => TABLE (Dept) AS D
                 PARTITION BY Deptno
                 ORDER BY Tstamp ) )
```

- WITH SET SEMANTICS
  all rows of a partition to be processed on the same virtual processor ("partitionable")
- PARTITION BY:
  Table partitioned on list of columns, to be processed on separate virtual processor each

# Row Pattern Recognition

- Search an ordered partition of rows for matches to a regular expression
- RPR can be supported in either the FROM clause or the WINDOW clause

| Symbol | Tradeday | Price |
|---|---|---|
| XYZ | 2009-06-08 | 50 |
| XYZ | 2009-06-09 | 60 |
| XYZ | 2009-06-10 | 49 |
| XYZ | 2009-06-11 | 40 |
| XYZ | 2009-06-12 | 35 |
| XYZ | 2009-06-13 | 45 |
| XYZ | 2009-06-14 | 45 |



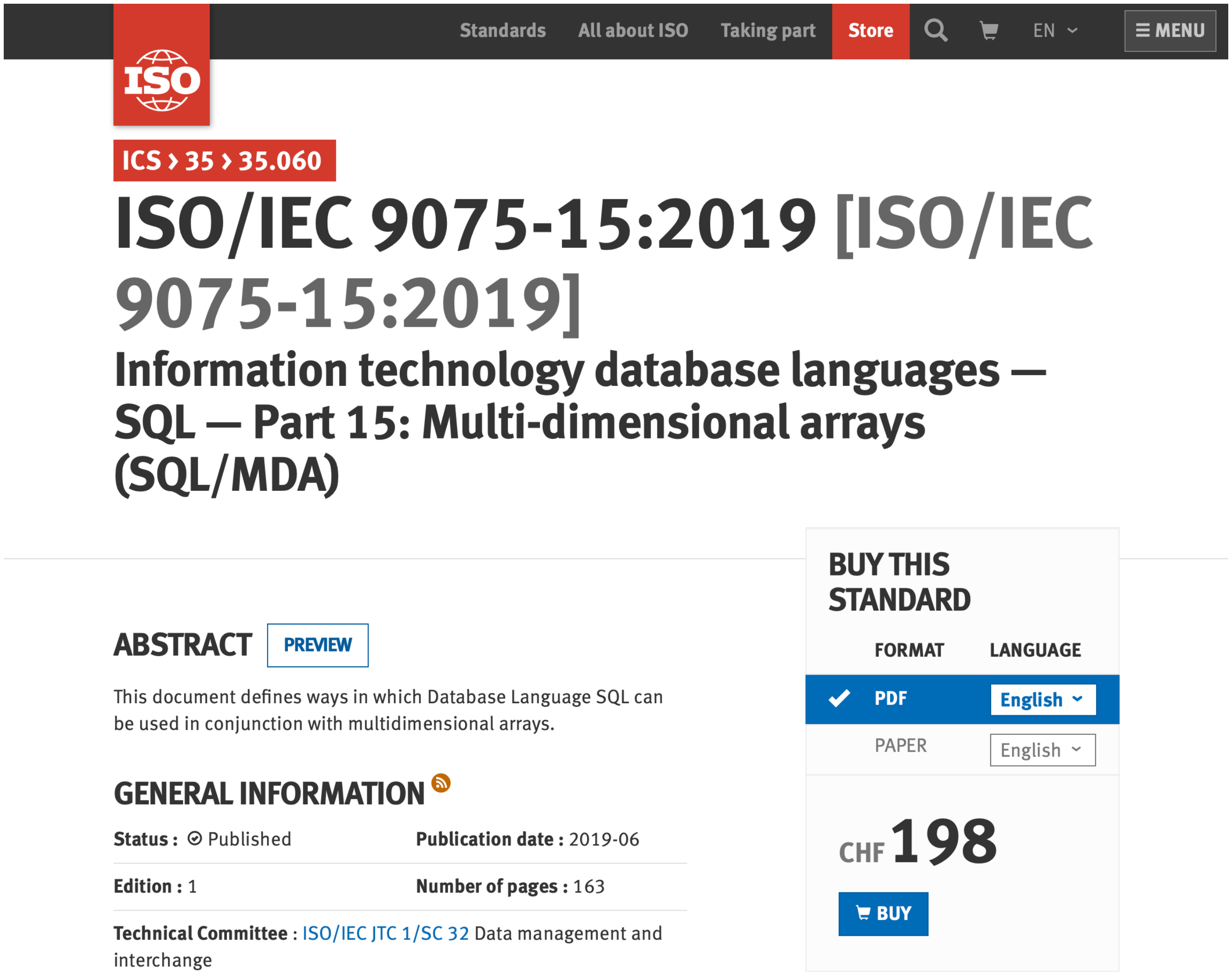
| Symbol | Matchno | Startp | Bottomp | Endp | Avgp |
|---|---|---|---|---|---|
| XYZ | 1 | 60 | 35 | 45 | 45.8 |

# What's Next?

- SQL:2019: Multi Dimensional Arrays – SQL/MDA
  - Applications in natural sciences research (heat maps), geo applications (e.g., remote sensing image processing)
  - 1D-Arrays already in SQL:1999 (very limited ways to query and update)
  - Multidimensional arrays discussed for more than 20 years
  - Prominent example array databases:
    - Rasdaman (Peter Baumann)
    - SciDB (Michael Stonebraker)
- SQL:2020+: Property Graphs
  - Also discussed for about 20 years
    - Neo4j graph database (Cypher)
    - RDF (SPARQL)
- In the works: Streaming SQL
- Probabilistic modeling in SQL:2030?

It takes about 20-30 years from non-standard features to standard elements

# SQL:2019 Part 15 ISO Standard

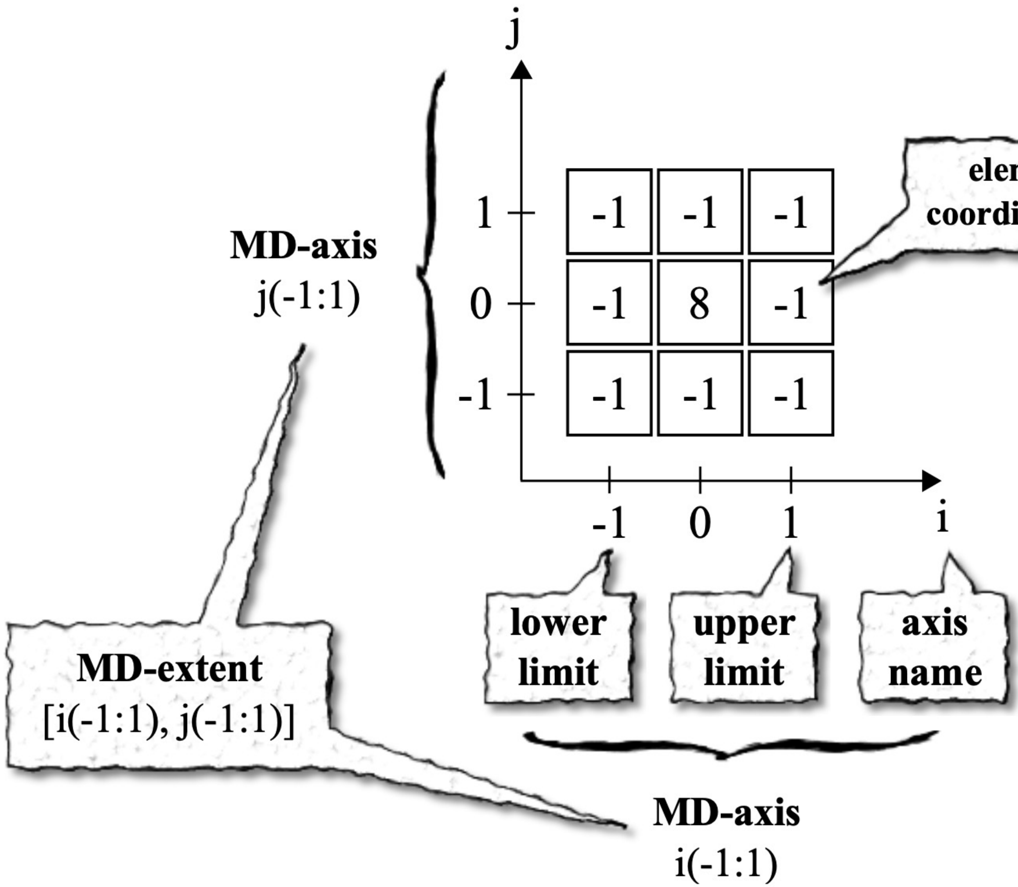


# Acknowledgements

- Parts of subsequent presentations are taken from

Dimitar Mišev, Peter Baumann, SQL Support for Multidimensional Arrays
Jacobs Universität, Technical Report No. 34, July **2017**

# Array Data Model



# MD-arrays Constructed by Element Enumeration

| **Example** | **SQL fragment** |
|---|---|
| 1-D MD-array of 10 floating-point elements at coordinates ranging from [10] to [19]. The element at coordinate [10] is $-0.5$, at [11] is $-1.5$, and so on. | `MDARRAY [temp(10:19)] [-0.5, -1.5, -0.34, 0.1, 1.12, 0.34, 1.5, 0.2, 1.15, 0.033]` |
| 2-D 3x3 convolution kernel, as shown on Figure 4. The element at coordinate $[0,0]$ is 8, which is the 5th element in the `<md-array element list>`, while the elements at all other coordinates are $-1$. | `MDARRAY [i(-1:1), j(-1:1)] [-1, -1, -1, -1, 8, -1, -1, -1, -1]` |
| 3-D 2x2x2 MD-array of 8 SMALLINT elements, such that the element with value 1 is at coordinate $[0,1,2]$, 2 is at coordinate $[0,1,3]$, 3 at $[0,2,2]$, 4 at $[0,2,3]$, 5 at $[1,1,2]$, and so on. | `MDARRAY [x(0:1), y(1:2), z(2:3)] [1, 2, 3, 4, 5, 6, 7, 8]` |

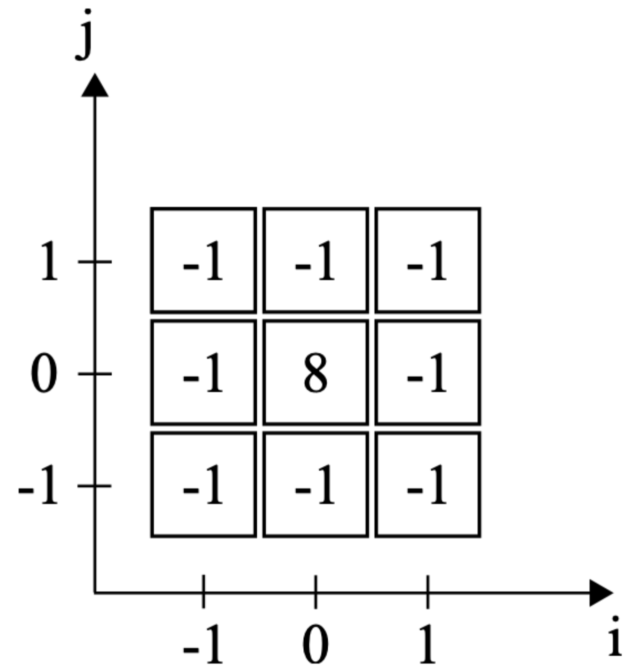
# MD-arrays Created with Constructor by Iteration

| Example | SQL fragment |
|---|---|
| 2-D constant MD-array such that the value of each element is 0 (zero). | `MDARRAY [x(0:9), y(0:9)]`<br>`ELEMENTS 0` |
| 1-D “gradient” MD-array of 10 elements, in which the value of each element is equal to its coordinate. | `MDARRAY [x(0:9)]`<br>`ELEMENTS x` |
| 2-D “gradient” MD-array of 100 elements, in which the value of each element is equal to the sum of its x and y coordinates. | `MDARRAY [x(0:9), y(0:9)]`<br>`ELEMENTS x + y` |
| 2-D MD-array, which is derived from an existing MD-array A with MD-extent [x(0:9),y(0:9)], so that the value of each element in the newly created MD-array is the square of the corresponding element in A. Note that MD-array element referencing is used in this example, | `MDARRAY MDEXTENT(A)`<br>`ELEMENTS POWER(A[x, y], 2)` |

# From Tables to Arrays

| j | i | element |
|---|---|---|
| -1 | -1 | -1 |
| -1 | 0 | -1 |
| -1 | 1 | -1 |
| 0 | -1 | -1 |
| 0 | 0 | 8 |
| 0 | 1 | -1 |
| 1 | -1 | -1 |
| 1 | 0 | -1 |
| 1 | 1 | -1 |

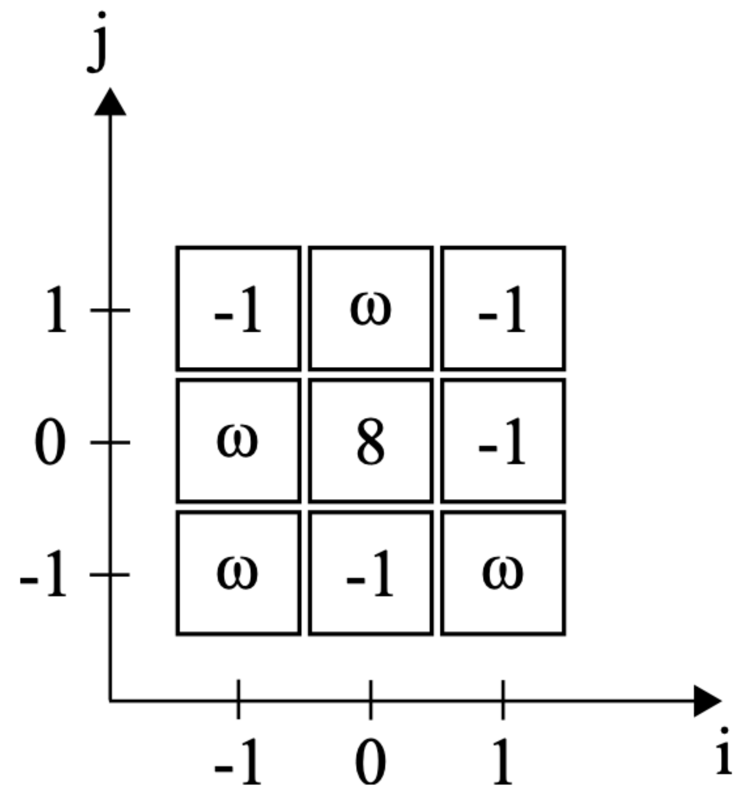
$\longrightarrow$



# From Tables to Arrays with Nulls ($\omega$)

| j | i | element |
|---|---|---|
| -1 | 1 | -1 |
| 0 | -1 | -1 |
| 0 | 0 | 8 |
| 1 | 0 | -1 |
| 1 | 1 | -1 |

$\longrightarrow$



# Array Updates

There are three general patterns that can be observed when updating a target MD-array $T$ with a source value $S$:

— $S$ and $T$ are MD-arrays of the same MD-dimension;

— $S$ is an MD-array of MD-dimension that is less than the MD-dimension of $T$;

— $S$ is of a compatible type to the element type of $T$, rather than an MD-array.

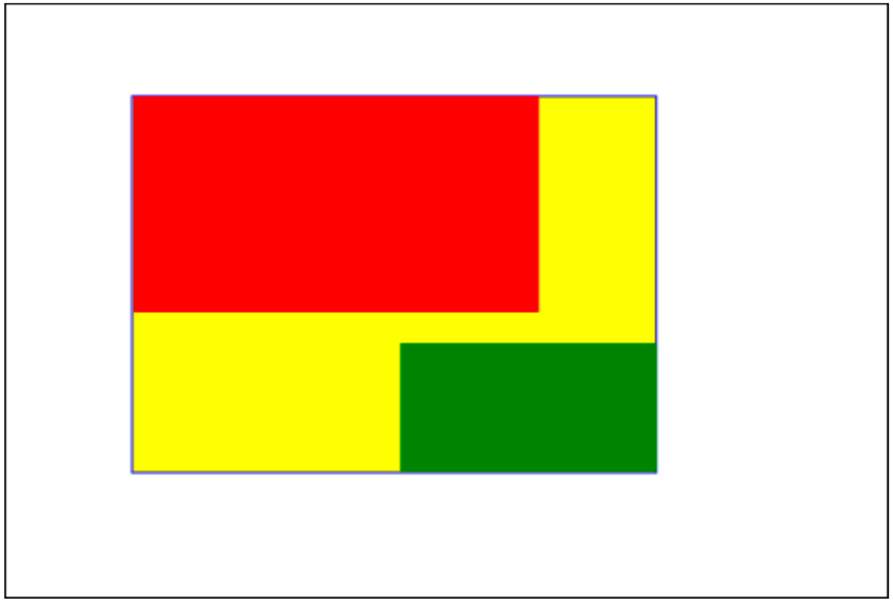
```
TABLE Temp(
  T REAL MDARRAY[ t(1:12), x(1:1000), y(1:1000) ]
)
```

S

```
UPDATE Temp SET T = MDARRAY[t(1:1), x(1:1), y(1:3)] [0.0, 1.0, 2.0]
```

```
UPDATE Temp SET T[t(1:1), x(1:1), y(1:3)] =
  MDARRAY[t(1:1), x(1:1), y(1:3)] [0.0, 1.0, 2.0]
```

# Array Updates as an Extension



- Red rectangle: MD-extent of *T* ,
- White rectangle with black border: maximum MD-extent
- Green rectangle: MD-extent of *S*
- Result MD-array of update: rectangle formed of the red, yellow and green parts; elements in the yellow subset are set to null.

# From Arrays to Tables

```
SELECT T.* FROM UNNEST(MDARRAY[x(1:2), y(1:2)] [1, 2, 5, 6])
                AS T(x, y, value)
```

| x | y | value |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 2 | 2 |
| 2 | 1 | 5 |
| 2 | 2 | 6 |

| ord | x | y | value |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 2 | 1 | 2 | 2 |
| 3 | 2 | 1 | 5 |
| 4 | 2 | 2 | 6 |

```
SELECT T.* FROM UNNEST(MDARRAY[x(1:2), y(1:2)] [1, 2, 5, 6])
                WITH ORDINALITY AS T(ord, x, y, value)
```

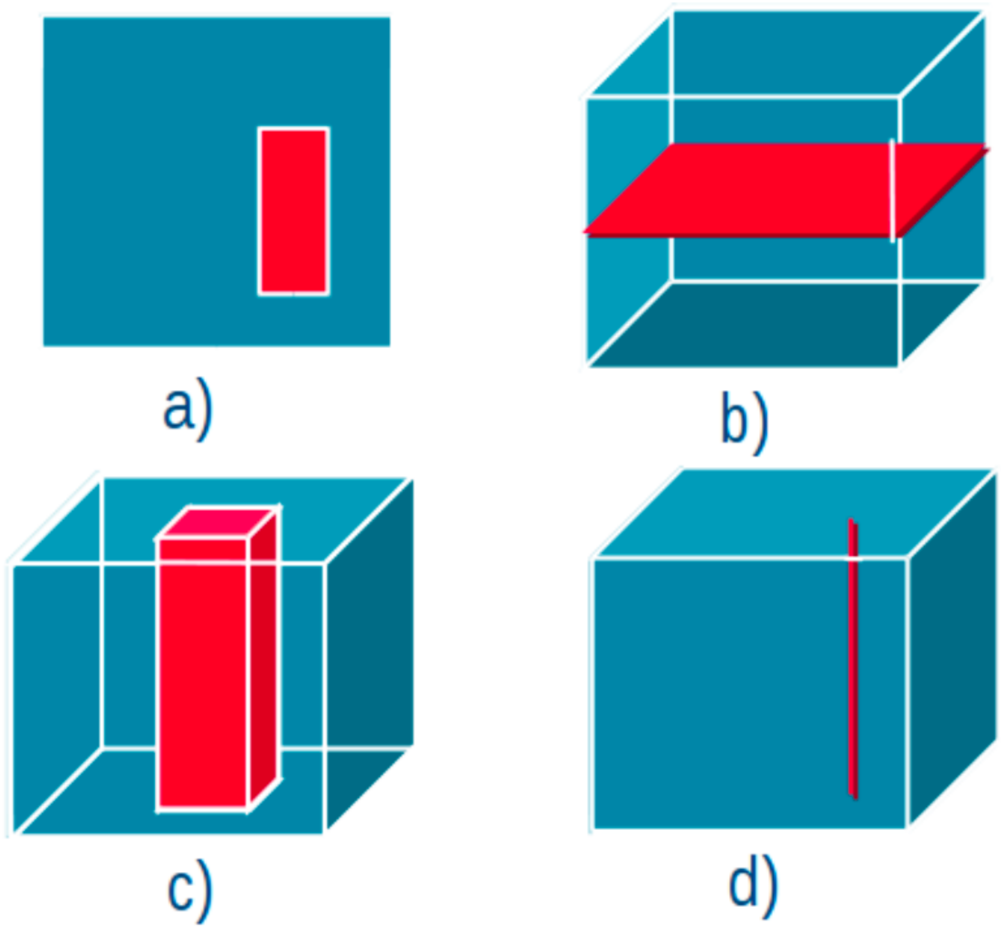
# Examples

```
CREATE TABLE kernels (
  id INT PRIMARY KEY,
  name CHARACTER VARYING(50),
  kernel SMALLINT MDARRAY [i(-100:100), j(-100:100)],
  filter SMALLINT MDARRAY [i(-100:100), j(-100:100)] )

INSERT INTO kernels VALUES
  (1, ‘Edge detection’,
   MDARRAY [i(-1:1), j(-1:1)] [-1, -1, -1,
                               -1,  8, -1,
                               -1, -1, -1],
   MDARRAY [i(-2:2), j(-2:2)] [2,  4,  5,  4, 2,
                               4,  9, 12,  9, 4,
                               5, 12, 15, 12, 5,
                               4,  9, 12,  9, 4,
                               2,  4,  5,  4, 2])
```

# Referencing Elements

| **Example** | **Result** |
|---|---|
| `kernel[0, 0]`<br>`kernel[i(0), j(0)]`<br>`kernel[j(0), i(0)]` | 8 |
| `kernel[50, 0]` | null value |
| `kernel[-1, 1000]`<br>`kernel[x(0), y(0)]`<br>`kernel[i(0), 0]` | error |

# Projection on Arrays („Subsetting“)

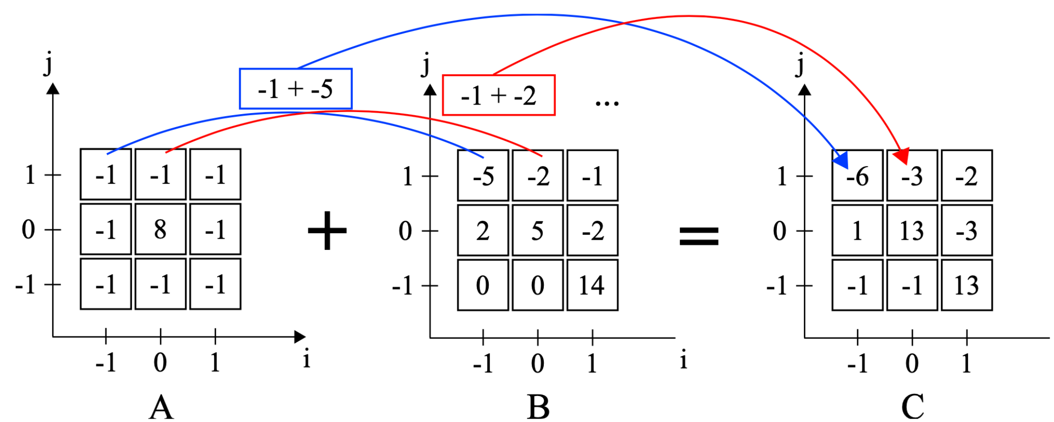


# Projection

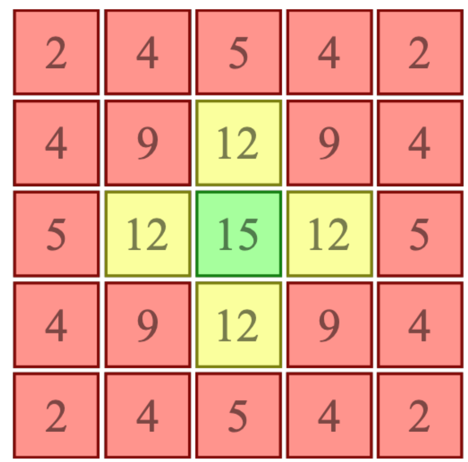
| **Example** | **Result** |
| --- | --- |
| `kernel[0:1, 0:1]`<br>`kernel[i(0:1), j(0:1)]`<br>`kernel[j(0:1), i(0:1)]` | `MDARRAY [i(0:1), j(0:1)] [8, -1, -1, -1]` |
| `kernel[0, 0:1]`<br>`kernel[0, 0:*]`<br>`kernel[i(0), j(0:1)]`<br>`kernel[i(0), j(0:*)]`<br>`kernel[j(0:1), i(0)]` | `MDARRAY [j(0:1)] [8, -1]` |
| `kernel[0:0, 0:1]`<br>`kernel[0:0, 0:*]`<br>`kernel[i(0:0), j(0:1)]`<br>`kernel[i(0:0), j(0:*)]`<br>`kernel[j(0:1), i(0:0)]` | `MDARRAY [i(0:0), j(0:1)] [8, -1]` |
| `kernel[0, -1:1]`<br>`kernel[0, *:*]`<br>`kernel[i(0)]`<br>`kernel[i(0), j(*:*)]` | `MDARRAY [j(-1:1)] [-1, 8, -1]` |

# Further Array Operations

- Reshaping
- Shifting
- Axis renaming
- Scaling
- Concatenation
- Operations
  - Function application
- Cast operations



# Array Construction

| Example | SQL fragment | Result |
|---|---|---|
| Replace negative elements with a 0 (zero), and positive with 1 (one). | `CASE`<br>`WHEN kernel <= 0`<br>`THEN 0 ELSE 1`<br>`END` | `MDARRAY [i(-1:1), j(-1:1)] [`<br>`0, 0, 0,`<br>`0, 1, 0,`<br>`0, 0, 0]` |
| Colorize an MD-array with "traffic-light" RGB color scheme (elements smaller than 10 "colored" as red, between 10 and 13 as yellow, and greater than 12 as red). | `CASE`<br>`WHEN filter < 10`<br>`THEN (255,0,0)`<br>`WHEN filter < 13`<br>`THEN (255,255,0)`<br>`ELSE (0,255,0)`<br>`END` |  |

# Joins on Arrays

- MDJOIN performs a join on two or more MD-arrays of equal MD-extents based on their coordinates
- Let A be defined as MDARRAY [x(0:2)] [1, 2, 3] and B as MDARRAY [x(0:2)] [4.1, 6.12, -0.2]

| Example | Result type | Result value |
|---|---|---|
| `MDJOIN(A, B, A)` | `ROW(FIELD1 SMALLINT, FIELD2 FLOAT, FIELD3 SMALLINT) MDARRAY [x(0:2)]` | `MDARRAY [x(0:2)] [ ROW(1, 4.1, 1), ROW(2, 6.12, 2), ROW(3, -0.2, 3) ]` |
| `MDJOIN(A AS red, B AS green, A AS blue)` | `ROW(red SMALLINT, green FLOAT, blue SMALLINT) MDARRAY [x(0:2)]` | `MDARRAY [x(0:2)] [ ROW(1, 4.1, 1), ROW(2, 6.12, 2), ROW(3, -0.2, 3) ]` |

# Decoding/Encoding

- Arrays from/to JSON structures
- Arrays from/to TIFF files
- …

| **Example** | **SQL fragment** |
|---|---|
| 1-D “gradient” JSON array of 6 elements, in which the value of each element is equal to its coordinate. | `MDDECODE('{ "data": [1, 2, 3, 4, 5, 6] }', 'application/json' RETURNING INT MDARRAY [x(1:6)])` |
| 2-D MD-array from a 3x3 convolution kernel array encoded as JSON (cf. Figure 4). | `MDDECODE('{ "data": [[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]] }', 'application/json' RETURNING INT MDARRAY [i(-1:1), j(-1:1)])` |
| 3-D MD-array from a 1x3x2 array encoded as JSON. | `MDDECODE('{ "data": [[[1, 2], [3, 4], [5, 6]]] }', 'application/json' RETURNING INT MDARRAY [t(0:0), x(0:2), y(0:1)])` |

# Scalability for Array Query Answering

- Scalability by specific partitioning techniques to distribute array data over multiple computing nodes
- Transactions across machines automatically managed for updates
- Exploit GPUs

# Applications

- Image processing (e.g., photogrammetry)
- (Static) time series (e.g., histograms)

# Summary SQL:2016 and Beyond

- Data structures in SQL
  - (Multi-)Sets of records (tables of tuples)
  - Trees of nested semi-structured objects (XML and JSON)
  - Multidimensional arrays (unnested)
  - Graphs
- Each structure with respective data definition and query language elements

# Dynamic Time Series Data

- Occur in many application scenarios (e.g., sensor networks)
  - Could be seen implemented as an extensible 1D-array
  - Could be implemented in a table [Timestamp, sensor, measurements…]
- Two challenges for data insertion and query answering
  - Scaling up: Swapping from disk is expensive
  - Scaling out: Transactions across machines expensive
- Specific features for time series management required to ensure scalability

# Acknowledgements

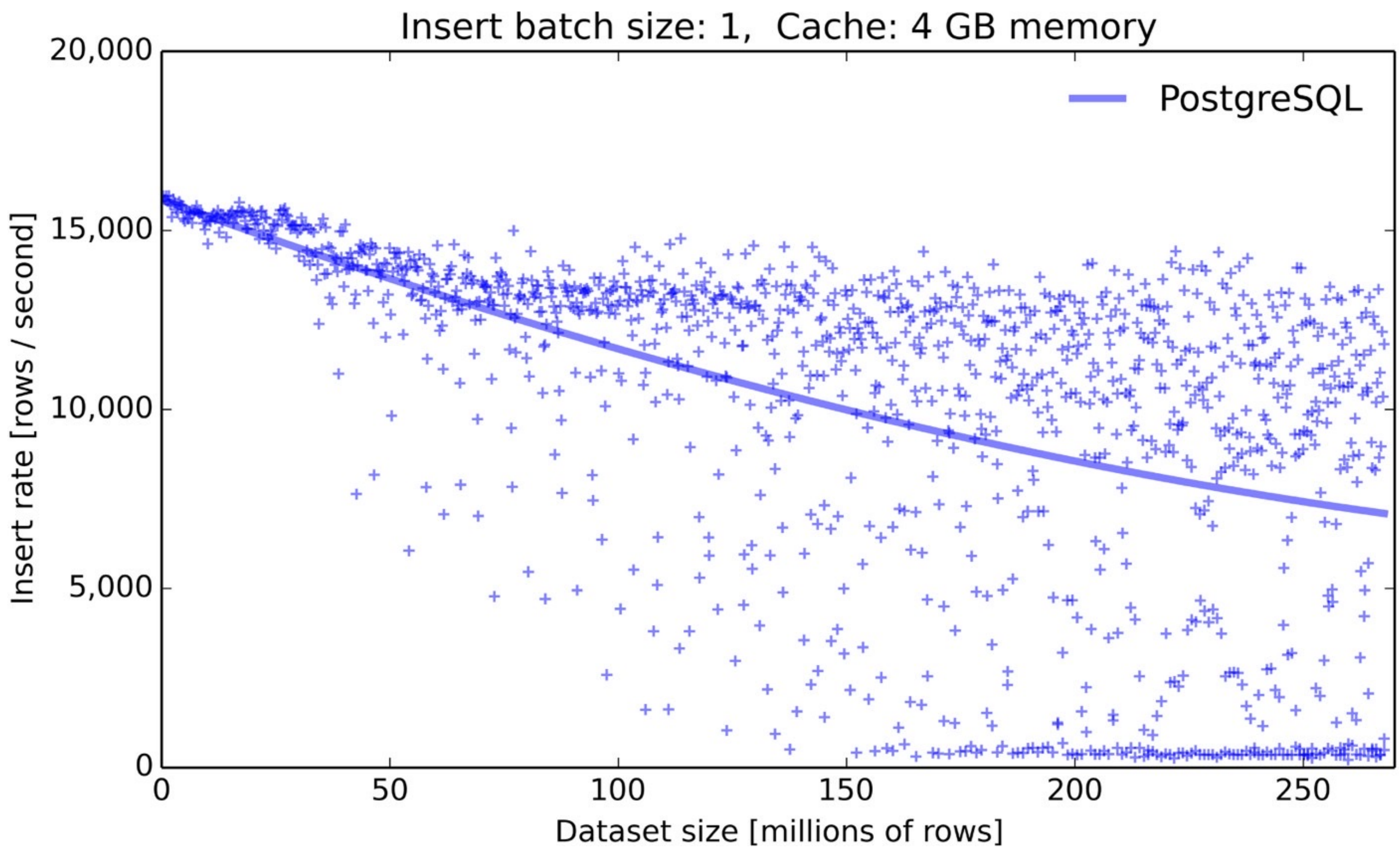
The following slides show diagrams of a presentation provided by timescale.com:

Building a scalable time-series database using Postgres

by Mike Freedman

https://github.com/timescale/timescaledb

# Inserting Rows into a Postgres Database



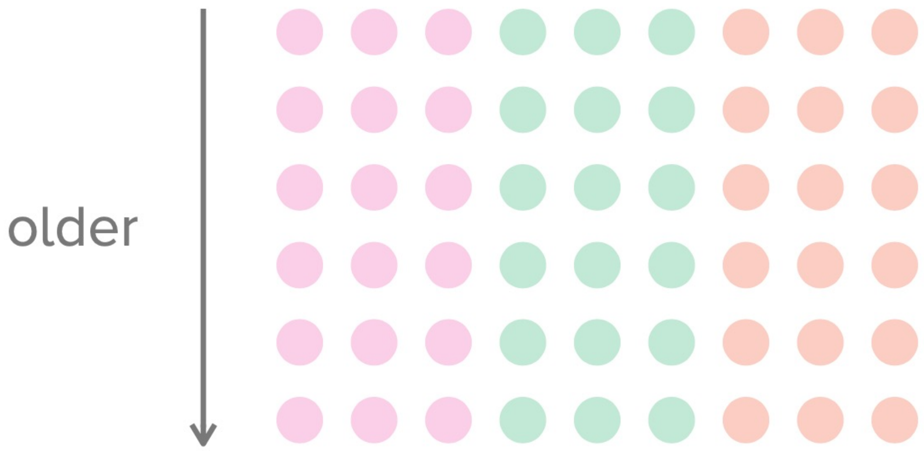

Postgres 9.6.2 on Azure standard DS4 v2 (8 cores), SSD (premium LRS storage)
Each row has 12 columns (1 timestamp, indexed, 1 host ID, 10 metrics)

LRS = Locally redundant storage

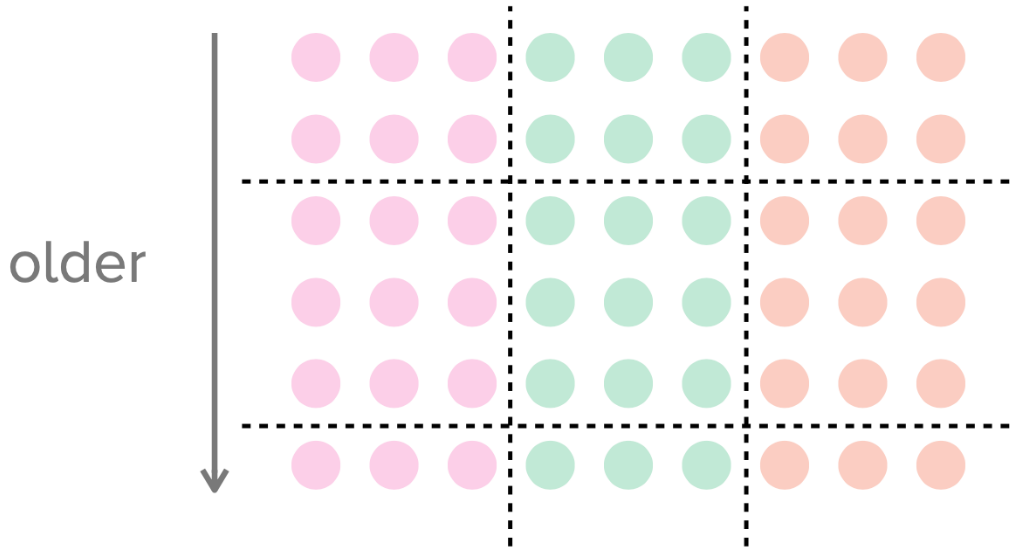
# Challenges to scalability

- As tables grow larger
  - Data and indexes no longer fit in memory
  - Reads/writes to random locations in B-tree
  - Separate B-tree for each secondary index
- I/O amplification makes it worse
  - Reads/writes at full-page granularity (8KB), not individual series elements
  - It doesn’t help to shrink DB page:
    - HDD still seeks
    - SSD has min Flash page size

# Application Scenario



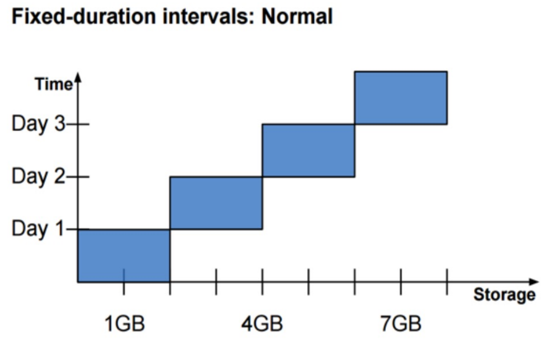
# Adaptive time/space partitioning

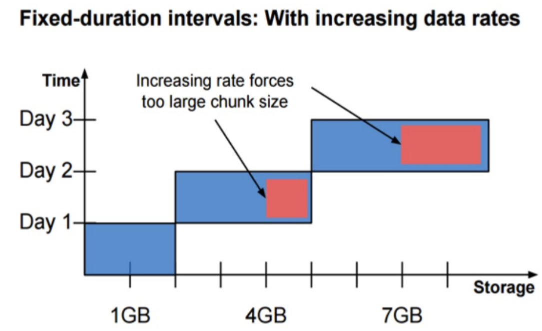


# How EXACTLY do we partition by time?

### Static, fixed duration?

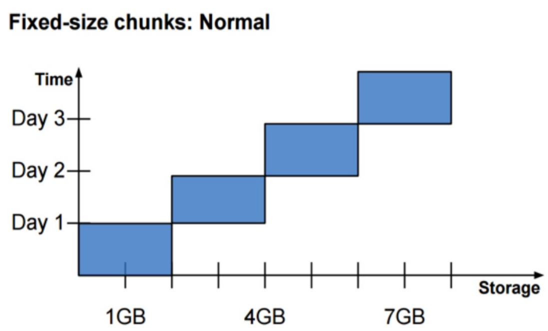
- Insufficient: Data volumes can change

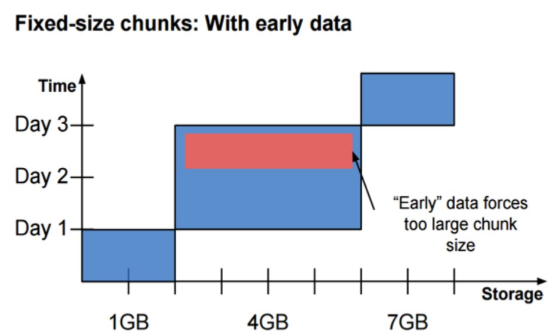


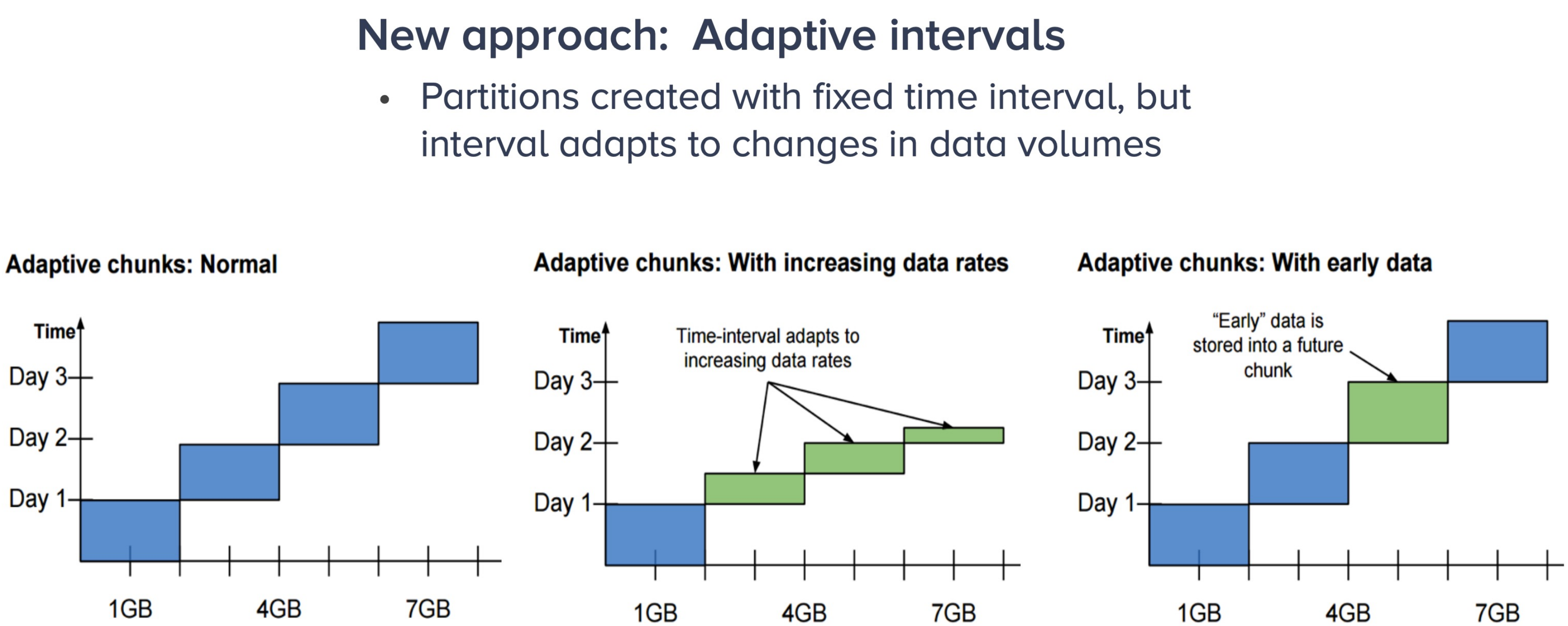

### Fixed target size?

- Early data can create too long intervals
- Bulk inserts expensive



# Adaptive time/space partitioning benefits



# Adaptive time/space partitioning benefits

- **Partitions spread across servers**
- **No centralized txn manager or special front-end**
  - Any node can handle any INSERT or QUERY
  - Inserts are routed/sub-batched to appropriate servers
  - Partition-aware query optimizations

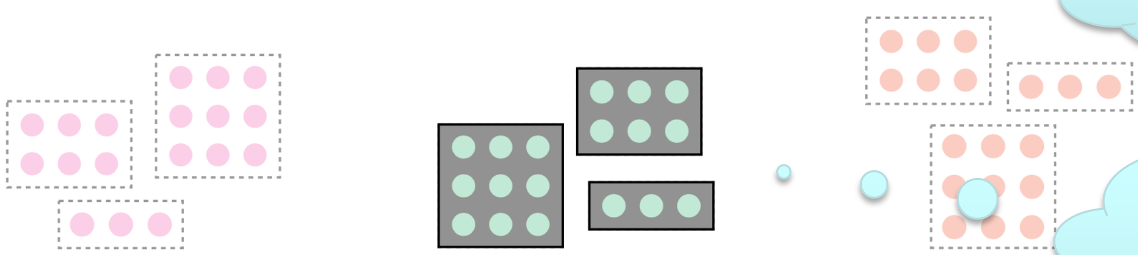
# Partition-aware Query Optimization

- Prevent unneeded chunks from entering a query plan
- Avoid querying chunks via **constraint exclusion analysis**

```
SELECT time, temp FROM data
    WHERE  time > now() - interval '7 days'
      AND device_id = '12345'
```

now() function makes the expression no longer accessible to plan time constraint exclusion

Need runtime exclusion



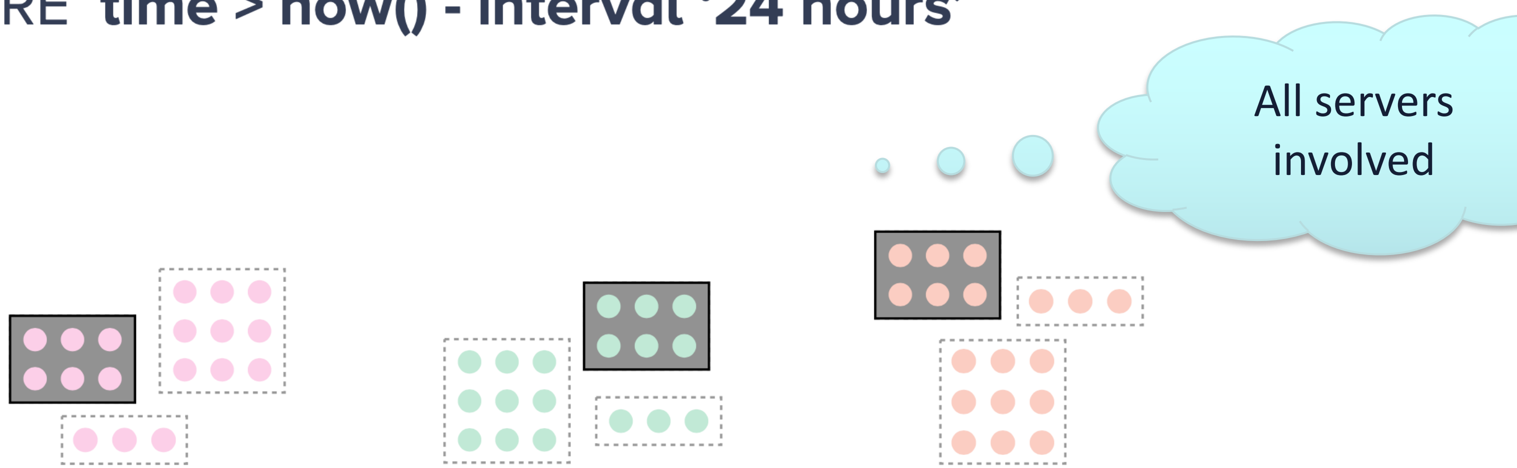
All data for '12345' on Server 2

https://www.timescale.com/blog/implementing-constraint-exclusion-for-faster-query-performance/

# Partition-aware Query Optimization

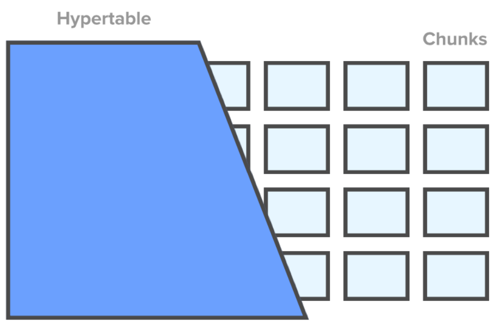
- Avoid querying chunks via **constraint exclusion analysis**

SELECT time, device_id, temp FROM data
WHERE **time > now() - interval '24 hours'**

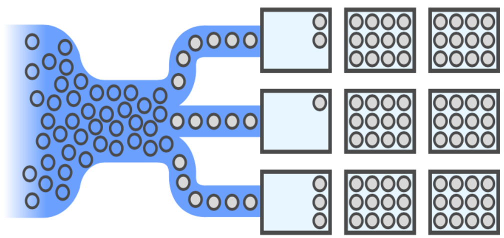



https://www.timescale.com/blog/implementing-constraint-exclusion-for-faster-query-performance/

# Hyper-Table Abstraction




- **Illusion of a single table**
- **SELECT against a single table**
  - Distributed query optimizations across partitions

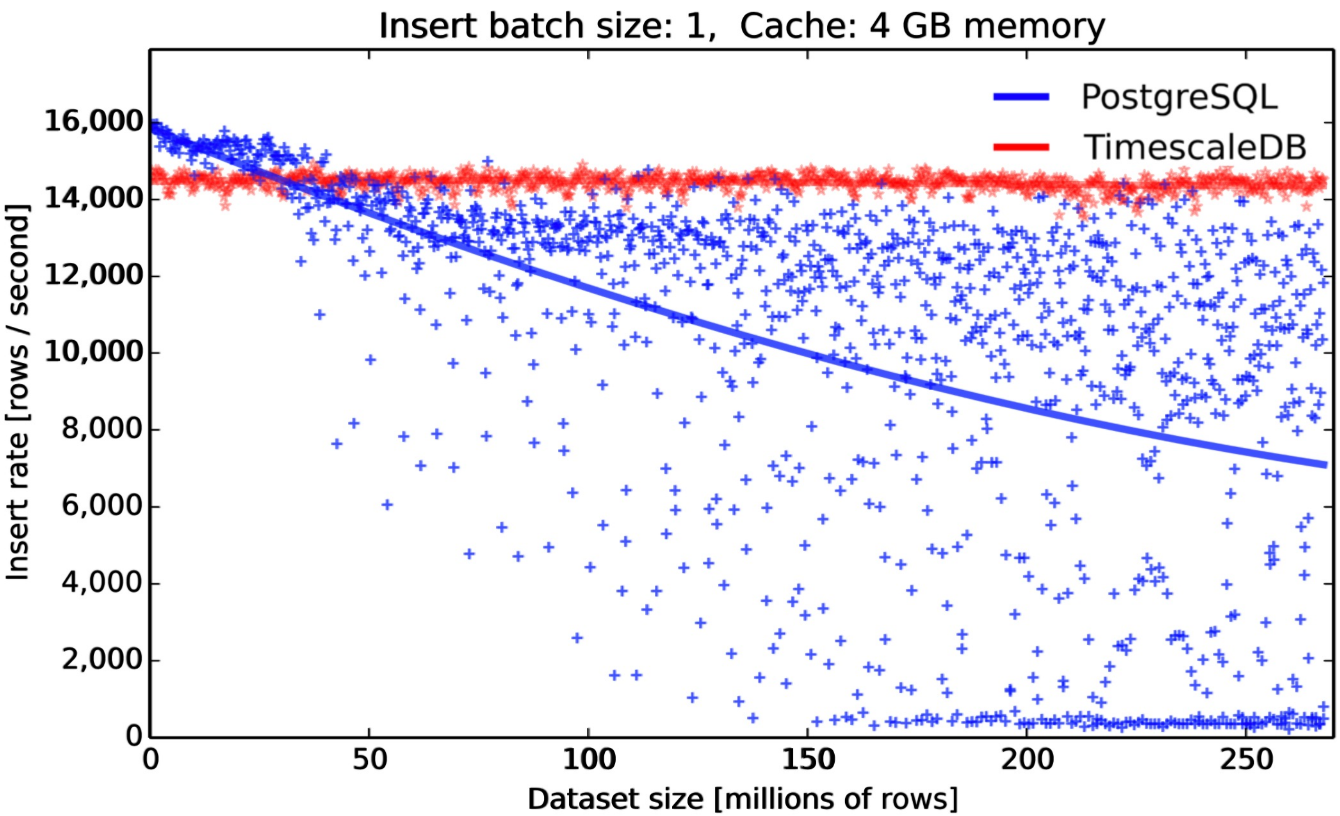


- **INSERT row / batch into single table**
  - Rows / sub-batches inserted into proper partitions
- **Engine automatically closes/creates partitions**
  - Based on both time intervals and table size

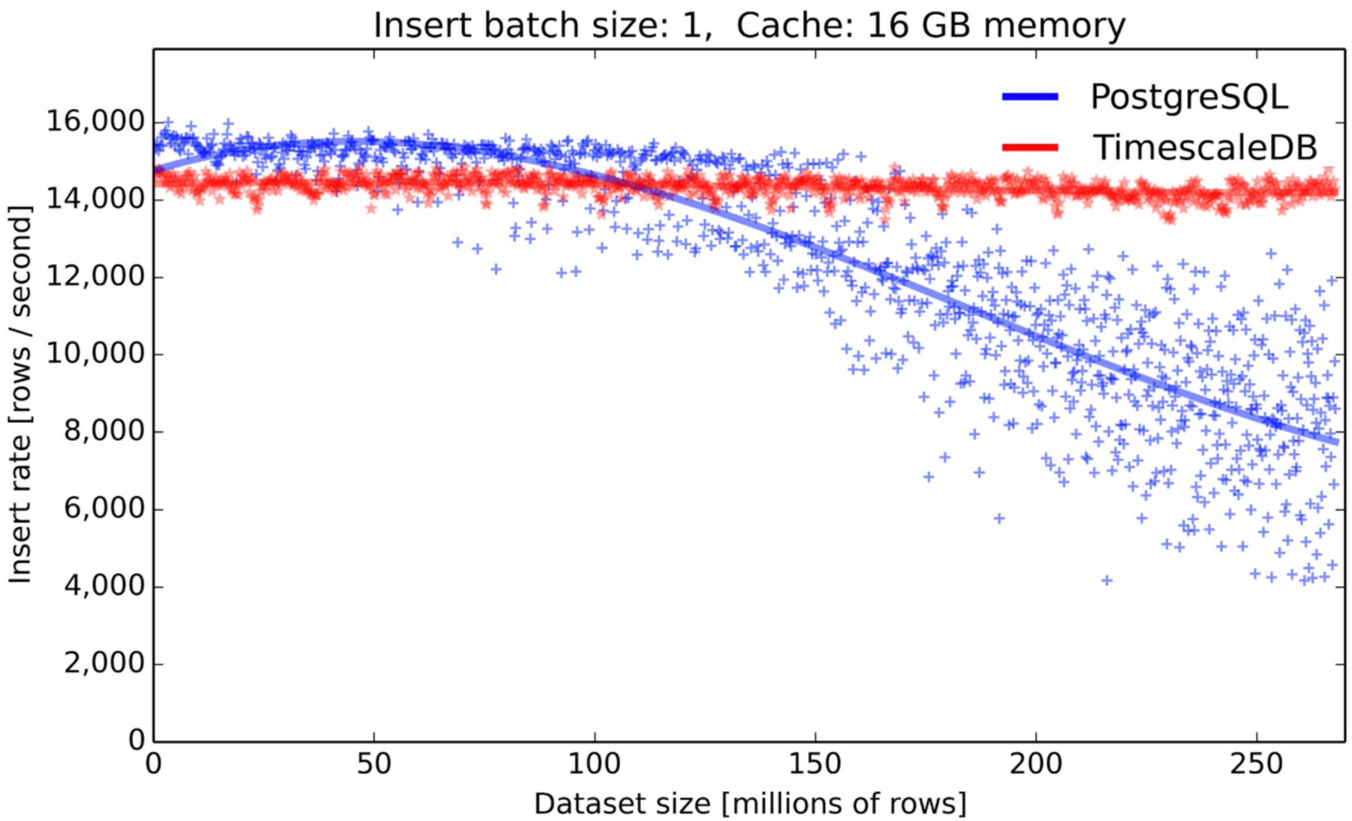
# Example: Importing Bulk Data

```
$ psql
psql (9.6.2)
Type "help" for help.

tsdb=#  CREATE TABLE data (
            time TIMESTAMP WITH TIME ZONE NOT NULL,
            device_id TEXT NOT NULL,
            temperature NUMERIC NULL,
            humidity NUMERIC NULL
        );

tsdb=#  SELECT create_hypertable ('data', 'time', 'device_id', 16);

tsdb=#  INSERT INTO data (SELECT * FROM old_data);
```

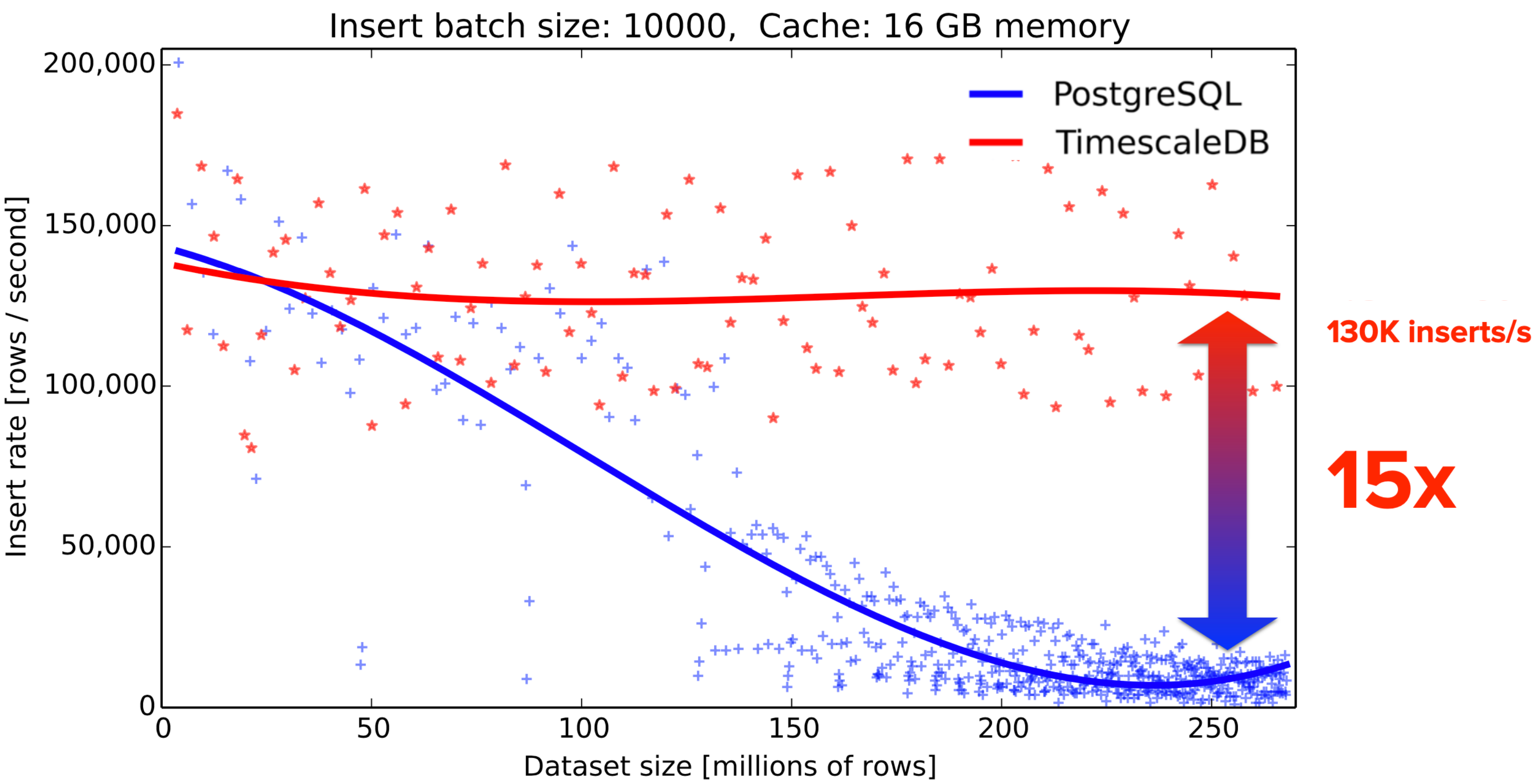
Partitioning column and number of partitions




**14.4K inserts/s**




14.4K inserts/s


Insert batch size: 10000, Cache: 16 GB memory
PostgreSQL
TimescaleDB
Insert rate [rows / second]
200,000
150,000
100,000
50,000
0
0
50
100
150
200
250
Dataset size [millions of rows]
130K inserts/s
15x

# Summary

- Time series support in databases (e.g., sensor data)
  - Example: TimescaleDB.com
- Time/space partitioning to ensure scalability
  - Example: TimescaleDB „hypertables“
- Bulk reading performance improved
- Partitioning-aware query optimization
  - TimescaleDB also much more efficient than Postgres for some queries referring to time data

**e.g., query “max per minute for all hosts with limit” is SQL:**

```
SELECT date_trunc('minute', time) as minute, max(usage) FROM cpu
   WHERE time < '2017-03-01 12:00:00’
   GROUP BY minute
   ORDER BY minute DESC
   LIMIT 5
```

# Add on: Timeseries Visualization (Grafana)