

Big Data Analytics

5. Relational Databases for Big Data

Lars Schmidt-Thieme

Information Systems and Machine Learning Lab (ISMLL)
Institute of Computer Science
University of Hildesheim, Germany

Outline

1. Introduction
2. Horizontal Partitioning
3. Vertical Partitioning
4. Sparse Data in Relational Databases

Outline

1. Introduction
2. Horizontal Partitioning
3. Vertical Partitioning
4. Sparse Data in Relational Databases

Replication and Partitioning

- ▶ traditionally, relational databases have been hosted on a single server.
 - ▶ simple relational database implementations such as SQLite still do not offer partitioning today

Replication and Partitioning

- ▶ traditionally, relational databases have been hosted on a single server.
 - ▶ simple relational database implementations such as SQLite still do not offer partitioning today
- ▶ **replication**:
maintain several synchronized copies of a database
 - ▶ fault tolerance, availability
 - ▶ load balancing

Replication and Partitioning

- ▶ traditionally, relational databases have been hosted on a single server.
 - ▶ simple relational database implementations such as SQLite still do not offer partitioning today
- ▶ **replication**:
maintain several synchronized copies of a database
 - ▶ fault tolerance, availability
 - ▶ load balancing
- ▶ **partitioning**:
split a database table into parts (that can be distributed)
 - ▶ distributed computing

Horizontal vs. Vertical Partitioning

Relational databases can be partitioned different ways:

- ▶ **Horizontal Partitioning:** (row-wise)
 - ▶ a table is split into subtables of different rows.

Horizontal vs. Vertical Partitioning

Relational databases can be partitioned different ways:

- ▶ **Horizontal Partitioning:** (row-wise)
 - ▶ a table is split into subtables of different rows.

- ▶ **Vertical Partitioning:** (column-wise)
 - ▶ a table is split into subtables of different columns.

Horizontal vs. Vertical Partitioning

Relational databases can be partitioned different ways:

- ▶ **Horizontal Partitioning:** (row-wise)
 - ▶ a table is split into subtables of different rows.
 - ▶ **Sharding:**
 - ▶ a large table is partitioned horizontally.
 - ▶ small tables are replicated.
 - ▶ e.g., for fact and dimension tables in data warehouses.
- ▶ **Vertical Partitioning:** (column-wise)
 - ▶ a table is split into subtables of different columns.

Outline

1. Introduction
2. Horizontal Partitioning
3. Vertical Partitioning
4. Sparse Data in Relational Databases

Horizontal Partitioning

- ▶ Partitioning is not covered by the current SQL standard (SQL:2016).
- ▶ Most implementations nowadays have partitioning support, e.g., MySQL, Oracle, MariaDB, PostgreSQL.
 - ▶ for MySQL/MariaDB:
 - ▶ Tables can be partitioned using the **PARTITION BY** clause
 - ▶ at creation by **CREATE TABLE**
 - ▶ anytime by **ALTER TABLE**
 - ▶ Partitioning criteria:
 - ▶ RANGE
 - ▶ LIST
 - ▶ HASH
 - ▶ RANGE COLUMNS, LIST COLUMNS, HASH COLUMNS
 - ▶ KEY
 - ▶ LINEAR HASH, LINEAR KEY

Horizontal Partitioning / Ranges

Rows can be assigned to different partitions based on different criteria:

► ranges

```

1 PARTITION BY range(<partitionexpression>) (
2     PARTITION <partitionname> VALUES LESS THAN (<partitionthreshold>)
3     , PARTITION <partitionname> VALUES LESS THAN (<partitionthreshold>)
4     ...
5 )

```

- a row is assigned to the first partition below whos $\langle \text{partitionthreshold} \rangle$ the row's $\langle \text{partitionexpression} \rangle$ is.
- the last $\langle \text{partitionthreshold} \rangle$ can be **MAXVALUE** to indicate no upper bound.
- $\langle \text{partitionthreshold} \rangle$ should be simple and fast.
- $\langle \text{partitionthreshold} \rangle$ can be just a column.

```

1 CREATE TABLE 'kunde' (
2     region int     NOT NULL
3     , nr         int     NOT NULL
4     , name      char(30)
5     , ed        date    NOT NULL
6 )
7 PARTITION BY range(region) (
8     PARTITION p0 VALUES LESS THAN (10)
9     , PARTITION p1 VALUES LESS THAN (20)
10    , PARTITION p2 VALUES LESS THAN (30)
11 );

```

Horizontal Partitioning / Ranges (2/2)

- ▶ example with slightly more complicated <partitionexpression>:

```
1 CREATE TABLE 'kunde' (  
2     region int     NOT NULL  
3     , nr          int     NOT NULL  
4     , name       char(30)  
5     , ed         date     NOT NULL  
6 )  
7 PARTITION BY RANGE(year(ed)) (  
8     PARTITION p0 VALUES LESS THAN (1990)  
9     , PARTITION p1 VALUES LESS THAN (2000)  
10    , PARTITION p2 VALUES LESS THAN maxvalue  
11 );
```

Horizontal Partitioning / Lists

► **lists:**

- partitioning values are explicitly enumerated.

```
1 CREATE TABLE 'kunde' (  
2   region int      NOT NULL  
3   , nr    int      NOT NULL  
4   , name  char(30)  
5   , ed   date      NOT NULL  
6 )  
7 PARTITION BY LIST(region) (  
8   PARTITION p0 VALUES IN (1, 3, 5 )  
9   , PARTITION p1 VALUES IN (2, 4, 6 )  
10  , PARTITION p2 VALUES IN (10, 11, 12 )  
11 );
```

Horizontal Partitioning /Column Ranges (or Lists)

▶ **range columns, list columns:**

- ▶ multiple expressions and thresholds (or value lists)
- ▶ a row is assigned to the first partition below whos <partitionvalue> all its <partitionexpression>s are.
- ▶ limitation: only bare columns are allowed as expressions.

```
1 CREATE TABLE 'kunde' (  
2     region int     NOT NULL  
3     , nr      int     NOT NULL  
4     , name   char(30)  
5     , ed     date    NOT NULL  
6 )  
7 PARTITION BY RANGE COLUMNS(region, nr) (  
8     PARTITION p0 VALUES LESS THAN (10, 10000)  
9     , PARTITION p1 VALUES LESS THAN (10, 20000)  
10    , PARTITION p2 VALUES LESS THAN (20, 10000)  
11    , PARTITION p3 VALUES LESS THAN (20, 20000)  
12 );
```

Horizontal Partitioning / Hash Values

► hash

- partition based on expression mod N.
- leads to uniform size distribution
(for expressions with many levels, e.g., IDs)

```

1 CREATE TABLE 'kunde' (
2     region int      NOT NULL
3     , nr      int   NOT NULL
4     , name   char(30)
5     , ed     date   NOT NULL
6 )
7 PARTITION BY LIST(MOD(region, 4)) (
8     PARTITION p0 VALUES IN (0)
9     , PARTITION p1 VALUES IN (1)
10    , PARTITION p2 VALUES IN (2)
11    , PARTITION p3 VALUES IN (3)
12 );

```

```

1 CREATE TABLE 'kunde' (
2     region int      NOT NULL
3     , nr      int   NOT NULL
4     , name   char(30)
5     , ed     date   NOT NULL
6 )
7 PARTITION BY HASH(region)
8 PARTITIONS 4;

```


Horizontal Partitioning /Queries

- ▶ the **PARTITION** clause of the **SELECT** statement can be used to query data from given partitions only
 - ▶ i.e., from the local partition (stored on the queried machine)

```
1 SELECT * FROM kunde PARTITION (p0)
```

Limitations

- ▶ indices are also partitioned
- ▶ all columns in the partitioning expression must be part of every key / unique column.
 - ▶ uniqueness constraint can be checked locally
 - ▶ for unique strings (e.g., email):
 - ▶ convert to int
 - ▶ **CHAR_LENGTH** length
 - ▶ **ASCII** code of first character

Outline

1. Introduction
2. Horizontal Partitioning
- 3. Vertical Partitioning**
4. Sparse Data in Relational Databases

Vertical Partitioning

- ▶ create a table for subsets of columns, linked by keys
- ▶ less useful for analytics as most often, if there are many columns, they are sparse
 - e.g., word indicators in texts, pattern indicators in images etc.
 - ▶ sparse data needs to be stored in a different way anyway in relational databases

```
1 CREATE TABLE 'kunde' (  
2   nr      int      NOT NULL  
3   , region int      NOT NULL  
4 )  
5 CREATE TABLE 'kunde2' (  
6   nr      int      NOT NULL  
7   , name  char(30)  
8 )  
9 CREATE TABLE 'kunde3' (  
10  nr      int      NOT NULL  
11  , ed     date     NOT NULL  
12 )
```

Outline

1. Introduction
2. Horizontal Partitioning
3. Vertical Partitioning
4. Sparse Data in Relational Databases

Sparse Data: Key-Value Tables

- ▶ column attribute representation:
email:

id	spam	buy	great	now	university	program	course	...
77dd	1	1	1	0	0	0	0	...
2314	0	0	0	1	0	1	1	...
⋮								

- ▶ **key/value representation:**

email:

id	spam
77dd	1
2314	0
⋮	

email_words:

email_id	word	value
77dd	buy	1
77dd	great	1
2314	now	1
2314	program	1
2314	course	1

Sparse Data: Key-Value Tables

column attribute representation:

- ▶ useful for dense data
- ▶ stores sparse data in a dense way
 - ▶ e.g., 99% sparsity \rightsquigarrow 100 times storage size

key/value representation:

- ▶ stores data in two tables
 - ▶ one table for the objects itself
 - ▶ one table for the attributes
 - ▶ object ID
 - ▶ attribute ID
 - ▶ attribute value
 - ▶ composite key (*objectID*, *attributeID*)
 - ▶ works OK if all / most attributes have the same type
 - ▶ requires joins to query information

JSON Format

- ▶ JSON — JavaScript Object Notation
- ▶ Data serialization format for dictionaries
 - ▶ = "objects consisting of attribute-value pairs"
- ▶ text format, human-readable
- ▶ schemaless
- ▶ programming-language independent (despite its name)
- ▶ alternatives: YAML — Yet Another Markup Language
- ▶ open standard (RFC 7159 and ECMA-404)

JSON Format / Example

Elementary data types:

- ▶ string: " " string
- ▶ number
- ▶ boolean: **true**, **false**
- ▶ value **null**

```
1 {  
2   "FirstName": "Bob",  
3   "Age": 35,  
4   "Address": "5_Oak_St.",  
5   "Hobby": "sailing"  
6 }
```

Composite data types:

- ▶ object: { }
- ▶ key/value pairs
- ▶ array: []

```
1 {  
2   "FirstName": "Jonathan",  
3   "Age": 37,  
4   "Address": "15_Wanamassa_Point_Road",  
5   "Languages": [ "English", "German" ]  
6 }
```

JSON Datatypes in RDBMS: Sparse Data

- ▶ Since SQL:2016 covered by the SQL standard.
- ▶ Modern RDBMS allow to store (parsed) JSON datatypes.
 - ▶ e.g, Postgres, Oracle
- ▶ JSON fields can be queried.
- ▶ JSON fields can be indexed.
- ▶ good tutorial:
<https://blog.codeship.com/unleash-the-power-of-storing-json-in-postgres/>

JSON Operators

- ▶ object: `->` : get value for specified key

```
1 '{ "id": 4, "name": "X202", "price": "199.99" }' :: json->'name'
2 "X202"
```

- ▶ array: `->` : get value at specified index

```
1 '[ "now", "program", "course" ]' :: json->1
2 "program"
```

- ▶ object/array: `#>` : get value at specified path

```
1 '{ "class": 0, "words": [ "now", "program", "course" ] }' :: json#>'{words, 2}'
2 "course"
```

- ▶ `->`, `#>` returns typed value (e.g, a json object),
`->>`, `#>>` returns a string.

JSON Datatypes in RDBMS / Example Todo List

► define JSON columns:

```
1 CREATE TABLE cards (  
2   id integer NOT NULL,  
3   board_id integer NOT NULL,  
4   data jsonb  
5 );
```

► insert JSON data:

```
1 INSERT INTO cards VALUES (1, 1, '{"name": "Paint house", "tags": ["Improvements", "Office"],  
2   "finished": true}');  
3 INSERT INTO cards VALUES (2, 1, '{"name": "Wash dishes", "tags": ["Clean", "Kitchen"],  
4   "finished": false}');  
5 INSERT INTO cards VALUES (3, 1, '{"name": "Cook lunch", "tags": ["Cook", "Kitchen", "Tacos"],  
6   "ingredients": ["Tortillas", "Guacamole"], "finished": false}');  
7 INSERT INTO cards VALUES (4, 1, '{"name": "Vacuum", "tags": ["Clean", "Bedroom", "Office"],  
8   "finished": false}');  
9 INSERT INTO cards VALUES (5, 1, '{"name": "Hang paintings", "tags": ["Improvements", "Office"],  
10  "finished": false}');
```

JSON Datatypes in RDBMS / Example Todo List

► query JSON data:

```

1 SELECT data->>'name' AS name FROM cards
2 name
3 -----
4 Paint house
5 Wash dishes
6 Cook lunch
7 Vacuum
8 Hang paintings
9 (5 rows)

```

► filtering JSON data:

```

1 SELECT * FROM cards WHERE data->>'finished' = 'true';
2 id | board_id | data
3 ---+-----+-----
4 1 | 1 | {"name": "Paint_house", "tags": ["Improvements", "Office"], "finished": true}
5 (1 row)

```

JSON Datatypes in RDBMS / Example Todo List

▶ checking column existence:

```
1 SELECT count(*) FROM cards WHERE data ? 'ingredients';
2 count
3 -----
4      1
5 (1 row)
```

▶ expanding data:

```
1 SELECT
2   jsonb_array_elements_text(data->'tags') as tag
3 FROM cards
4 WHERE id = 1;
5 tag
6 -----
7 Improvements
8 Office
9 (2 rows)
```

JSON Datatypes in RDBMS / Example Todo List

► indices: without indices:

```

1 SELECT count(*) FROM cards WHERE data->>'finished' = 'true';
2 count
3 -----
4 4937
5 (1 row)
6 Aggregate (cost=335.12..335.13 rows=1 width=0) (actual time=4.421..4.421 rows=1 loops=1) -> Seq Scan on
7   Filter: ((data ->> 'finished'::text) = 'true'::text)
8   Rows Removed by Filter: 5062
9   Planning time: 0.071 ms
10  Execution time: 4.465 ms

```

► indices: with indices:

```

1 CREATE INDEX idxfinished ON cards ((data->>'finished'));
2 count
3 -----
4 4937
5 (1 row)
6 Aggregate (cost=118.97..118.98 rows=1 width=0) (actual time=2.122..2.122 rows=1 loops=1) -> Bitmap Heap
7   Recheck Cond: ((data ->> 'finished'::text) = 'true'::text)
8   Heap Blocks: exact=185
9   -> Bitmap Index Scan on idxfinished (cost=0.00..4.66 rows=50 width=0) (actual time=0.671..0.671 rows=1)
10   Index Cond: ((data ->> 'finished'::text) = 'true'::text)
11 Planning time: 0.084 ms
12 Execution time: 2.199 ms

```

JSON Datatypes in RDBMS / Example Todo List

► query array elements:

```

1 SELECT count(*) FROM cards WHERE data->>'finished' = 'true';
2 count
3 -----
4 4937
5 (1 row)
6 Aggregate (cost=335.12..335.13 rows=1 width=0) (actual time=4.421..4.421 rows=1 loops=1) -> Seq Scan on
7   Filter: ((data ->> 'finished'::text) = 'true'::text)
8     Rows Removed by Filter: 5062
9 Planning time: 0.071 ms
10 Execution time: 4.465 ms

```

► using indices:

```

1 CREATE INDEX idxgintags ON cards USING gin ((data->'tags'));
2 count
3 -----
4 1537
5 (1 row)
6 Aggregate (cost=20.03..20.04 rows=1 width=0) (actual time=2.665..2.666 rows=1 loops=1) -> Bitmap Heap Sc
7   Recheck Cond: (((data -> 'tags'::text) ? 'Clean'::text) AND ((data -> 'tags'::text) ? 'Kitchen'::text)
8     Heap Blocks: exact=185
9   -> Bitmap Index Scan on idxgintags (cost=0.00..16.01 rows=1 width=0) (actual time=0.750..0.750 rows=1
10     Index Cond: (((data -> 'tags'::text) ? 'Clean'::text) AND ((data -> 'tags'::text) ? 'Kitchen'::text)
11 Planning time: 0.088 ms
12 Execution time: 2.706 ms
13 (8 rows)
14
15 Time: 3.248 ms

```


Summary

- ▶ For relational databases **partitioning** and **replication** are considered separately.
- ▶ Relational databases can be partitioned:
 - ▶ horizontally: row-wise
 - ▶ vertically: column-wise
 - ▶ sharded: row-wise for large tables, small tables are replicated.
- ▶ The SQL standard describes neither replication nor partitioning.

Summary

- ▶ MariaDB tables can be partitioned based on a **partition expression**:
 - ▶ assigning value **ranges** to a partition
 - ▶ assigning value **lists** to a partition
 - ▶ assigning **hash** values to a partition.
- ▶ **Sparse data** can be represented in relational databases using a separate **key/value attribute table**.
 - ▶ efficient for storage
 - ▶ expensive to query due to joins
- ▶ SQL:2016 and most modern RDBMs support (parsed) JSON columns
 - ▶ type **jsonb**
 - ▶ access fields/elements with `->` and `#>`
 - ▶ supports indexing by json keys