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

Outline

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

Horizonal vs. Vertical Partitioning

Relational databases can be partitioned different ways:

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

Horizonal vs. Vertical Partitioning

Relational databases can be partitioned different ways:

- **Horizonal 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.

Horizonal vs. Vertical Partitioning

Relational databases can be partitioned different ways:

- **Horizonal 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

Horizonal Partitioning

- Partitioning is not covered by the current SQL standards.
- Most implementations nowadays have partitioning support, e.g., MySQL, Oracle, MariaDB.
  - 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

Horizonal Partitioning / Ranges

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

- **ranges**

```
PARTITION BY range(<partitionexpression>) (
    PARTITION <partitionname> VALUES LESS THAN (<partitionthreshold>)
  , PARTITION <partitionname> VALUES LESS THAN (<partitionthreshold>)
  ...
)
```

  - a row is assigned to the first partition below whos <partitionthreshold> the row's <partitionexpression> is.
  - the last <partitionthreshold> can be **MAXVALUE** to indicate no upper bound.
  - <partitionthreshold> should be simple and fast.
  - <partitionthreshold> can be just a column.

```
CREATE TABLE `kunde` (
    region int      NOT NULL
  , nr     int      NOT NULL
  , name   char(30)
  , ed     date     NOT NULL
)
PARTITION BY range(region) (
    PARTITION p0 VALUES LESS THAN (10)
  , PARTITION p1 VALUES LESS THAN (20)
  , PARTITION p2 VALUES LESS THAN (30)
```

Horizonal Partitioning / Ranges (2/2)

- example with slightly more complicated <partitionexpression>:

```
CREATE TABLE ‘kunde‘ (
    region int      NOT NULL
  , nr     int      NOT NULL
  , name   char(30)
  , ed     date     NOT NULL
)
PARTITION BY RANGE(year(ed)) (
    PARTITION p0 VALUES LESS THAN (1990)
  , PARTITION p1 VALUES LESS THAN (2000)
  , PARTITION p2 VALUES LESS THAN maxvalue
);
```

Horizonal Partitioning / Lists

- lists:
  - partitioning values are explicitly enumerated.

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

Horizonal 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>s all its <partitionexpression>s are.
  - limitation: only bare columns are allowed as expressions.

```
CREATE TABLE ‘kunde‘ (
   region int     NOT NULL
  , nr    int     NOT NULL
  , name  char(30)
  , ed    date    NOT NULL
)
PARTITION BY RANGE COLUMNS(region, nr) (
   PARTITION p0 VALUES LESS THAN (10, 10000)
  , PARTITION p1 VALUES LESS THAN (10, 20000)
  , PARTITION p2 VALUES LESS THAN (20, 10000)
  , PARTITION p3 VALUES LESS THAN (20, 20000)
);
```

Horizonal Partitioning / Hash Values

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

```
CREATE TABLE ‘kunde‘ (
    region int      NOT NULL
  , nr     int      NOT NULL
  , name   char(30)
  , ed     date     NOT NULL
)
PARTITION BY LIST(MOD(region, 4)) (
    PARTITION p0 VALUES IN (0)
  , PARTITION p1 VALUES IN (1)
  , PARTITION p2 VALUES IN (2)
  , PARTITION p3 VALUES IN (3)
);
```

```
CREATE TABLE ‘kunde‘ (
    region int      NOT NULL
  , nr     int      NOT NULL
  , name   char(30)
  , ed     date     NOT NULL
)
PARTITION BY HASH(region)
PARTITIONS 4;
```

Horizonal 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)

```
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

Outline

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

```
CREATE TABLE ‘kunde‘ (
    nr     int      NOT NULL
  , region int      NOT NULL
)
CREATE TABLE ‘kunde2‘ (
    nr     int      NOT NULL
  , name   char(30)
)
CREATE TABLE ‘kunde3‘ (
    nr     int      NOT NULL
  , ed     date     NOT NULL
)
```

Outline

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

JSON Format

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

JSON Format / Example

```
{
  FirstName: "Bob",
  Age: 35,
  Address: "5␣Oak␣St.",
  Hobby: "sailing"
}
```

```
{
  FirstName: "Jonathan",
  Age: 37,
  Address: "15␣Wanamassa␣Point␣Road",
  Languages: [ "English", "German" ]
}
```

JSON Datatypes in RDBMS: Sparse Data

- 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 Datatypes in RDBMS / Example Todo List

- define JSON columns:

```
CREATE TABLE cards (
  id integer NOT NULL,
  board_id integer NOT NULL,
  data jsonb
);
```

- insert JSON data:

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

JSON Datatypes in RDBMS / Example Todo List

- query JSON data:

```
SELECT data->>'name' AS name FROM cards
name
----------------
Paint house
Wash dishes
Cook lunch
Vacuum
Hang paintings
(5 rows)
```

- filtering JSON data:

```
SELECT * FROM cards WHERE data->>'finished' = 'true';
id | board_id |                              data
----+----------+-----------------------------------------------------------------------------
 1 |        1 | {"name": "Paint␣house", "tags": ["Improvements", "Office"], "finished": true}
(1 row)
```

JSON Datatypes in RDBMS / Example Todo List

- checking column existence:

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

- expanding data:

```
SELECT
  jsonb_array_elements_text(data->'tags') as tag
FROM cards
WHERE id = 1;
tag
--------------
Improvements
Office
(2 rows)
```

JSON Datatypes in RDBMS / Example Todo List

- indices: without indices:

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

- indices: with indices:

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

JSON Datatypes in RDBMS / Example Todo List

- query array elements:

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

- using indices:

```
CREATE INDEX idxgintags ON cards USING gin ((data->'tags'));
count
-------
1537
(1 row)
Aggregate (cost=20.03..20.04 rows=1 width=0) (actual time=2.665..2.666 rows=1 loops=1) -> Bitmap Heap Sc
   Recheck Cond: (((data -> 'tags'::text) ? 'Clean'::text) AND ((data -> 'tags'::text) ? 'Kitchen'::text
   Heap Blocks: exact=185
   -> Bitmap Index Scan on idxgintags (cost=0.00..16.01 rows=1 width=0) (actual time=0.750..0.750 rows=1
      Index Cond: (((data -> 'tags'::text) ? 'Clean'::text) AND ((data -> 'tags'::text) ? 'Kitchen'::te
Planning time: 0.088 ms
Execution time: 2.706 ms
(8 rows)

Time: 3.248 ms
```