Big Data Analytics

B. Distributed Storage / B.2 Partioning of Relational Databases

Lars Schmidt-Thieme

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

Syllabus

Tue.	9.4.	(1)	0. Introduction
			A. Parallel Computing
Tue.	16.4.	(2)	A.1 Threads
Tue.	23.4.	(3)	A.2 Message Passing Interface (MPI)
Tue.	30.4.	(4)	A.3 Graphical Processing Units (GPUs)
			B. Distributed Storage
Tue.	7.5.	(5)	B.1 Distributed File Systems
Tue.	14.5.	(6)	B.2 Partioning of Relational Databases
Tue.	21.5.	(7)	B.3 NoSQL Databases
			C. Distributed Computing Environments
Tue.	28.5.	(8)	C.1 Map-Reduce
Tue.	4.6.	—	— *Pentecoste Break* —
Tue.	11.6.	(9)	C.2 Resilient Distributed Datasets (Spark)
Tue.	18.6.	(10)	C.3 Computational Graphs (TensorFlow)
			D. Distributed Machine Learning Algorithms
Tue.	25.6.	(11)	D.1 Distributed Stochastic Gradient Descent
Tue.	2.7.	(12)	D.2 Distributed Matrix Factorization
Tue.	9.7.	(13)	Questions and Answers

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 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

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 `customer` (
    region  int       NOT NULL
  , cid     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 ‘customer‘ (
    region  int        NOT NULL
  , cid     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 'customer' (
      region  int        NOT NULL
    , cid     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 `customer` (
      region  int         NOT NULL
    , cid     int         NOT NULL
    , name    char(30)
    , ed      date        NOT NULL
)
PARTITION BY RANGE COLUMNS(region, cid) (
      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 'customer' (
     region  int        NOT NULL
   , cid     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 'customer' (
     region  int        NOT NULL
   , cid     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 customer 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
    - use **KEY** partitioning type
      — provides a hash value for any data type (of a list of columns)

Outline

Parallel Query Processing / Selects

- Assume tables are hash partitioned with hash function h
- How to answer a select query?

```
select * from T where C = c
```

```
if hash h depends only on attributes C:
  select matching local rows on partition h(c)
else:
  for all partitions in parallel:
    select matching local rows
  concatenate results
```

Parallel Query Processing / Joins / Example 1

- How to answer a join query?

```
select name, city from customer, address where customer.cid = address.cid
```

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

```
CREATE TABLE 'address' (
      aid     int           NOT NULL
    , street  varchar(256)  NOT NULL
    , city    varchar(256)  NOT NULL
    , zip     varchar(256)  NOT NULL
    , cid     int           NOT NULL
)
PARTITION BY LIST(MOD(cid, 4)) (
      PARTITION p0 VALUES IN (0)
    , PARTITION p1 VALUES IN (1)
    , PARTITION p2 VALUES IN (2)
    , PARTITION p3 VALUES IN (3)
);
```

Parallel Query Processing / Joins / Example 1

customer table:

cid	name
0	Miller
1	Martin
2	Adebayo
3	Schneider
4	Lopez
5	Wong
6	Zhang
7	Rossi

address table:

aid	cid	city
0	2	Lagos
1	7	Milano
2	1	Nizze
3	5	Bejing
4	4	Toronto
5	6	Taiwan
6	3	Hamburg
7	0	Boston

Parallel Query Processing / Joins / Example 1

partition p0:

customer:

cid	name
0	Miller
4	Lopez

address:

aid	cid	city
7	0	Boston
4	4	Toronto

partition p1:

customer:

cid	name
1	Martin
5	Wong

address:

aid	cid	city
2	1	Nizze
3	5	Bejing

partition p2:

customer:

cid	name
2	Adebayo
6	Zhang

address:

aid	cid	city
0	2	Lagos
5	6	Taiwan

partition p3:

customer:

cid	name
3	Schneider
7	Rossi

address:

aid	cid	city
6	3	Hamburg
1	7	Milano

Parallel Query Processing / Joins / Example 2

- How to answer a join query?

```
select name, city from customer, address where customer.cid = address.cid
```

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

```
CREATE TABLE 'address' (
      aid     int            NOT NULL
    , street  varchar(256)   NOT NULL
    , city    varchar(256)   NOT NULL
    , zip     varchar(256)   NOT NULL
    , cid     int            NOT NULL
)
PARTITION BY LIST(MOD(aid, 4)) (
      PARTITION p0 VALUES IN (0)
    , PARTITION p1 VALUES IN (1)
    , PARTITION p2 VALUES IN (2)
    , PARTITION p3 VALUES IN (3)
);
```

Parallel Query Processing / Joins / Example 2

partition p0:

customer:

cid	name
0	Miller
4	Lopez

address:

aid	cid	city
0	2	Lagos
4	4	Toronto

partition p1:

customer:

cid	name
1	Martin
5	Wong

address:

aid	cid	city
1	7	Milano
5	6	Taiwan

partition p2:

customer:

cid	name
2	Adebayo
6	Zhang

address:

aid	cid	city
2	1	Nizze
6	3	Hamburg

partition p3:

customer:

cid	name
3	Schneider
7	Rossi

address:

aid	cid	city
3	5	Bejing
7	0	Boston

Parallel Query Processing / Joins

- How to answer a join query?

```
select * from T,S where T.A = a, S.B = b, T.C = S.C
```

- both tables need to be partitioned w.r.t. C the same way
  - they either are already partitioned w.r.t. C
  - or they need to be **repartitioned** that way:
    define new hash function:

$$\tilde{h} : \mathcal{C} \rightarrow \{1, \ldots, P\}$$

```
if hash h depends only on attributes C:
  for all partitions in parallel :
    join matching local rows of T and S over C
  concatenate results
else :
  for all partitions in parallel :
    send all local rows x of T with x.A = a to partition h̃(x.C)
    send all local rows x of S with x.B = b to partition h̃(x.C)
  for all partitions in parallel :
    join matching received rows of T and S over C
  concatenate results
```

Note: Here $\mathcal{C}$ denotes the domain of attribute C.

Parallel Query Processing / Cartesian Products

- How to answer a cartesian product query?

```
      select * from T,S
```

- naive method: broadcast the smaller table (say S):

```
for all partitions in parallel :
  send all rows of S to all partitions
for all partitions in parallel :
  combine all local rows of T with all (received) rows of S
concatenate results
```

  - communication cost: $P \cdot N_S$
    — for P partitions, N_T rows in table T and N_S rows in table S

Parallel Query Processing / Cartesian Products (2/2)

- more efficient method:
  - arrange the P partitions in a $P_T \times P_S$ grid: $P = P_T \cdot P_S$
  - define new hash functions:

$$h_T : \mathcal{X} \rightarrow \{1, \ldots, P_T\}, \quad h_S : \mathcal{X} \rightarrow \{1, \ldots, P_S\}$$

```
for all partitions in parallel :
  send all local rows x of T to all partitions (h_T(x), *)
  send all local rows x of S to all partitions (*, h_S(x))
for all partitions in parallel :
  combine all received rows of T with all received rows of S
concatenate results
```

  - communication cost: $P \cdot (\frac{N_T}{P_T} + \frac{N_S}{P_S})$
    - minimal for $\frac{N_T}{P_T} = \frac{N_S}{P_S}$, thus $P_T = \sqrt{P \frac{N_T}{N_S}}, \quad P_S = \sqrt{P \frac{N_S}{N_T}}$
  - $\rightsquigarrow$ communication cost: $2\sqrt{P \cdot N_T \cdot N_S}$
    - smaller than naive costs $P \cdot N_S$ for tables of similar size by a factor of $\sqrt{P}/2$, i.e., for large P

Note: Here $\mathcal{X}$ denotes the union of domains of tables T and S.

Parallel Query Processing / Multiway Joins

- How to answer a multiway join query?

```
select * from T,S,R where T.A = S.A, S.B = R.B
```

- naive method:
  - first join T and S
  - then join result of the first step with R
- naive method possibly could produce large intermediate results
- better method:
  - Shares / HyperCube algorithm [Afrati and Ullman, 2010]

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 ‘customer‘ (
      nr      int        NOT NULL
    , region  int        NOT NULL
)
CREATE TABLE ‘customer2‘ (
      nr      int        NOT NULL
    , name    char(30)
)
CREATE TABLE ‘customer3‘ (
      nr      int        NOT NULL
    , ed      date       NOT NULL
)
```

- no explicit support by MySQL

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

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 $(\mathit{objectID}, \mathit{attributeID})$
  - 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**

Composite data types:

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

```
{
  "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

- Since SQL:2016 covered by the SQL standard.
- Modern RDBMS allow to store (parsed) JSON datatypes.
  - e.g, MySQL, 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

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

- array: -> : get value at specified index

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

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

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

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

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

- query JSON fields without indices (slow):

```
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 o
      Filter : ((data ->> 'finished'::text) = 'true'::text)
      Rows Removed by Filter: 5062
Planning time: 0.071 ms
Execution time: 4.465 ms
```

- query JSON fields with indices (faster):

```
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 He
      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 JSON arrays/dictionaries without indices (slow):

```
SELECT count(*) FROM cards
WHERE
  data->'tags' ? 'Clean'
  AND data->'tags' ? 'Kitchen';
count
--------
1537
(1 row)
...
Planning time: 0.063 ms
Execution time: 6.710 ms
(6 rows)

Time: 7.314 ms
```

- query JSON arrays/dictionaries with indices (gin = generalized inverted index):

```
CREATE INDEX idxgintags ON cards USING gin ((data->'tags'));
count
--------
1537
(1 row)
...
Planning time: 0.088 ms
Execution time: 2.706 ms
```

Summary (1/2)

- For relational databases **partioning** 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 partioning.

Summary (2/2)

- 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

References

Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM, 2010.