Databases

Seminar 9
Indexes, Partitioning, Scaling, OLAP, ETL

Optimization: The life cycle of the request.

What happens after we write a query?

1. A database connection is created. The query is sent to the database in plain text.
2. The parser checks the correctness of the query syntax and creates a query tree.
3. The query rewriting system transforms the query – we get an updated query tree; a rule system is used)
4. The scheduler/optimizer creates a query plan.
5. The handler recursively traverses the query plan and gets the resulting set of strings

Optimization: Query Tree

The query tree is a special internal representation of SQL queries with its full analysis by key parameters:

- Command type (SELECT, UPDATE, DELETE, INSERT);
- List of relationships used;
- The target relationship where the result will be written;
- The list of fields (* is converted to a complete list of all fields);
- A list of restrictions (which are specified in WHERE);
- etc.

Optimization: how to read a query plan

The planner is a PostgreSQL component that tries to work out the most efficient way to execute an SQL query.
The execution plan contains information about how the database server will organize the viewing of the tables involved in the query.

The EXPLAIN operator:

- Outputs the execution plan generated by the PostgreSQL scheduler for the specified statement.
- Shows how tables affected by the operator will be scanned — just sequentially, by index, etc.
- Shows which join algorithm will be selected to combine rows read from tables.
- Shows the expected cost (in conventional units) of executing the request.
- MISSING from the SQL standard.

EXPLAIN operator example

EXPLAIN [(option [, ...])] statement
EXPLAIN [ANALYZE] [VERBOSE] statement

where option can be one of:
ANALYZE [boolean]
VERBOSE [boolean]
COSTS [boolean]
BUFFERS [boolean]
TIMING [boolean]
FORMAT { TEXT | XML | JSON | YAML }

INSERT INTO my_table ...;
EXPLAIN SELECT * FROM my_table;
- - - -
QUERY PLAN
Seq Scan on my_table
(cost=0.00..18334.00 rows=1000000
width=37)

What does it mean?

- The data is read by the Set Scan method (see the next question)
- The data is read from the my_table
- cost table — the cost (in some conventional units) of getting the first row..
- rows — the approximate number of rows returned when performing the Seq Scan operation (no rows are subtracted, the value is approximate)
- width — the average size of one row in bytes

When you use EXPLAIN again, it will show the old statistics, you need to call the ANALYZE command to update them.

Optimization: ANALYZE

The ANALYZE operator:

- Collects statistical information about the contents of tables in the database and stores the results in the pg_statistic system directory;
- Analyzes all tables in the current database without parameters.
- If the table name is passed in the parameters, it processes only the specified table.
- If a list of column names is passed in the parameters, statistics collection will start only for these columns.
- MISSING from the SQL standard.

Optimization: ANALYZE

This time the query will actually be executed.

- actual time: the real time in milliseconds taken to get the first row and all rows, respectively.
- rows: the actual number of rows received during the Seq Scan.
- loops: how many times the Seq Scan operation had to be performed.
- Planning time: the time spent by the scheduler to build a query plan.
- Execution time: the total execution time of the request.

ANALYZE [VERBOSE] [table_name
[(column_name [, ...])]]
ANALYZE my_table;
EXPLAIN SELECT * FROM my_table;

EXPLAIN (ANALYZE) SELECT * FROM my_table;
- - - -
QUERY PLAN
Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37)
(actual time=0.402..97.000 rows=1000010 loops=1)

Planning time: 0.042 ms
Execution time: 138.229 ms

Indexes: definition and properties

An index is a special database object that is stored separately from tables and provides quick access to data. These are auxiliary structures: any index can be deleted and restored from the information in the table. Indexes also serve to support some integrity constraints.
There are six different types of indexes built into PostgreSQL 9.6.
Index properties:

- All indexes are secondary, they are separated from the table. All information about them is contained in the system directory.
- When adding/changing data related to the index, the index is rebuilt each time (this slows down query execution).
- There may be different mathematical structures inside (B-tree, black-red tree...)
- Indexes can be multi-column (maintaining conditions for several fields).
- Indexes link keys and TID (tuple id - #page: #offset) — the number of the page and the lines on it.
- Updating table fields for which indexes were not created does not result in rebuilding indexes (Heap-Only Tuples, HOT).

Indexes: HOT optimization

HOT optimization: when updating a row, if possible, Postgres will put a new copy of the row immediately after the old copy of the line. Also, a special label is placed in the old copy of the row, indicating that the new copy of the row is located immediately after the old one. Therefore, it is not necessary to update all indexes.

HOT optimization can only be used when:

- The operator and argument types match.
- The index is valid.
- The order of the fields inside a multi-column index is important in order to impose conditions, expecting the optimizer to select the index.
- The plan with its use is optimal (minimum cost).
- Postgres takes all the information from the system directory.

HOT optimization: example

CREATE [UNIQUE] INDEX [CONCURRENTLY] [name] ON table_name [USING METHOD]...
CREATE INDEX ON my_table(column_2);
ALTER INDEX [IF EXISTS] name RENAME TO new_name
DROP INDEX [CONCURRENTLY] [IF EXISTS] name [, ...] [CASCADE|RESTRICT]

ALTER INDEX [IF EXISTS] name RENAME TO new_name
ALTER INDEX [IF EXISTS] name SET TABLESPACE tablespace_name
ALTER INDEX [IF EXISTS] name SET (storage_parameter = value [, ...])
ALTER INDEX [IF EXISTS] name RESET (storage_parameter [, ...])
DROP INDEX [CONCURRENTLY] [IF EXISTS] name [, ...] [CASCADE|RESTRICT]

Indexes: general information

- Indexes work better the higher the selectivity of the condition, that is, the fewer rows it satisfies. As the sample increases, the overhead of reading index pages also increases.
- The situation is aggravated by the fact that sequential reading is faster than reading pages "randomly". This is especially true for hard drives, where the mechanical operation of bringing the head to the track takes significantly longer than reading the data itself; in the case of SSD drives, this effect is less pronounced.

Indexes: Seq Scan

- Method
  Sequential, block by block, reading of table data.
- Advantages
  With a large amount of data for a single index field value, it works more efficiently than index scanning, since it usually works with large blocks of data, therefore, it can potentially select more data per access operation than index scanning, respectively, fewer access operations are needed, the speed is higher.
- Disadvantages
  It is usually performed much slower than an index scan, since it reads all the data in the table.

Indexes: Index Scan

- Method
  An index is used for WHERE conditions (selectivity of the condition), reads the table when selecting rows.
- Advantages
  With the selectivity of the condition, the time is reduced N → lnN. (As a result of executing the query, significantly fewer rows are selected than their number in the page)
- Disadvantages
  If we collect the index for all fields, it will often be much bigger than the data in the table. As the sample increases, the chances increase that you will have to return to the same tabular page several times. (In this case, the optimizer switches to Bitmap Scan)

Indexes: Bitmap Index Scan

- Method
  First Index Scan, then sample control by table. For the most part, working with strings (index by row, bitmap of pages, subsequent selection)
- Advantages
  Effective for a large number of rows.

Indexes: Bitmap Index Scan

- Disadvantages
  It does not speed up work if the condition is not selective. The sample may be too large for the amount of RAM, then only a bitmap of pages is built — it takes up less space, but when reading a page, you have to double-check the conditions for each row stored there.
  - In the case of almost ordered data, building a bitmap is an extra step, the usual index scan will be the same.
  - A bitmap is created, where we assume that, according to the collected statistics, our rows satisfy our condition; there are pages in it;
  - If conditions are imposed on several fields of the table, and these fields are indexed, scanning the bitmap allows (if the optimizer deems it advantageous) to use several indexes at the same time. Bitmaps of string versions are constructed for each index, which are then logically multiplied bitwise (if expressions are connected by the AND condition), or logically

Indexes: Index Only Scan

- Method
  We practically do not access the table, all the necessary values are in the index.
- Advantages
  Very fast operation.
- Disadvantages
  It can only be used when the index includes all the fields necessary for the selection, and additional access to the table is not required.

If the index already contains all the data required for the query, then the index is called a covering index.

Scalability in PostgreSQL: Partitioning

A problem often arises: one of the tables in the database has grown a lot and the query execution time to this table has increased.

One of the solutions to this problem in PostgreSQL is partitioning. Partitions can contain a different number of rows, which means that the size on the disk will be different. The table is partitioned line by line.

Partitioning example

```
CREATE TABLE people_partitioned (
    person_id       SERIAL        PRIMARY KEY,
    first_name      VARCHAR(128)  NOT NULL,
    last_name       VARCHAR(128)  NOT NULL,
    birthday        DATE          NOT NULL,
    ...
) PARTITION BY RANGE (birthday);
```

```
CREATE TABLE
people_partitioned_birthdays_1800_to_1850
    PARTITION OF people_partitioned
    FOR VALUES FROM ('1800-01-01') TO ('1849-12-
31');
```

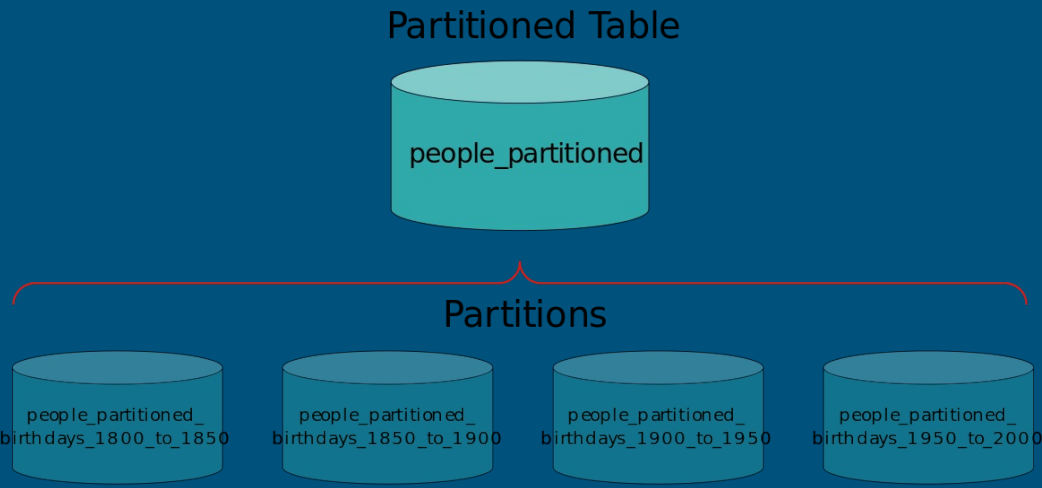
```
CREATE TABLE
people_partitioned_birthdays_1850_to_1900
    PARTITION OF people_partitioned
    FOR VALUES FROM ('1850-01-01') TO ('1899-12-
31');
```

```
CREATE TABLE
people_partitioned_birthdays_1900_to_1950
    PARTITION OF people_partitioned
    FOR VALUES FROM ('1900-01-01') TO ('1949-12-
31');
```

```
CREATE TABLE
people_partitioned_birthdays_1950_to_2000
    PARTITION OF people_partitioned
    FOR VALUES FROM ('1950-01-01') TO ('1999-12-
31');
```

Partitioning

The table that is partitioned is called the master table. The partition has a connection with the master table and is an ordinary table, that is, it can be accessed in the same way as the most ordinary table:
SELECT, INSERT (if it does not violate the restrictions imposed on the partition), UPDATE, DELETE.



Partitioning

What problems can partitioning solve?

- speeding up data sampling;
- speeding up data insertion;
- simplify the deletion of old data;
- simplify table maintenance.

It should be remembered that partitioning is not a panacea.

In which cases will partitioning not help or have no effect?

- backup creation time;
- recovery time from backup;
- disk space.

Partitioning

In the first two cases, the time will not change much, since a complete data search is performed. In the latter case, there were 1 billion rows and after partitioning we still have the same 1 billion rows. The occupied disk space will be almost the same.

Some limitations and possible problems based on the results of partitioning

- the partitioned table must be large enough; according to the documentation, it is recommended to partition if the table exceeds the memory size;
- the partitioned table cannot be referenced via FOREIGN KEYS (it is possible starting from PostgreSQL 12); however, the opposite (the partitioned table refers to others) is acceptable;
- In some cases, partitioning can degrade performance on read and write operations.;
- ideally, the query will be executed against one partition, but in the worst case, it will affect all partitions and, depending on the PostgreSQL settings, will increase the query execution time, as in the previous paragraph.

Types of partitioning

- by ranges

  The table is partitioned by "ranges" defined by a key column or set of columns, and not overlapping with each other (see the example above).

- by list

  The table is partitioned using a list that explicitly specifies which key values should relate to each partition.

Partitioning by list example

```
CREATE TABLE traffic_violations_p_list(
    seq_id         TEXT,
    violation_type TEXT,
    ...
)
PARTITION BY LIST (violation_type);

CREATE TABLE traffic_violations_p_list_warning
    PARTITION OF traffic_violations_p_list
    FOR VALUES IN ('Warning');

CREATE TABLE traffic_violations_p_list_sero
    PARTITION OF traffic_violations_p_list
    FOR VALUES IN ('SERO');
```

```
CREATE TABLE traffic_violations_p_list_Citation
    PARTITION OF traffic_violations_p_list
    FOR VALUES IN ('Citation');

CREATE TABLE traffic_violations_p_list_ESERO
    PARTITION OF traffic_violations_p_list
    FOR VALUES IN ('ESERO');

CREATE TABLE traffic_violations_p_list_default
    PARTITION OF traffic_violations_p_list
DEFAULT;
```

Postgres does not create a "default partition" automatically.
Think about the issues connected to having a "default partition".

Partitioning by hash

- by hash

The table is partitioned by specific modules and residuals, which are specified for each section. Each section contains rows for which the hash value of the partition key divided by the module is equal to the specified remainder.

```
CREATE TABLE traffic_violations_p_hash(
    seqid     TEXT,
    councils  SMALLINT,
    ...
)
PARTITION BY HASH (councils);

CREATE TABLE traffic_violations_p_hash_p1
    PARTITION OF traffic_violations_p_hash
    FOR VALUES WITH (MODULUS 5,
REMAINDER 0);
```

```
CREATE TABLE traffic_violations_p_hash_p2
    PARTITION OF traffic_violations_p_hash
    FOR VALUES WITH (MODULUS 5, REMAINDER 1);

CREATE TABLE traffic_violations_p_hash_p3
    PARTITION OF traffic_violations_p_hash
    FOR VALUES WITH (MODULUS 5, REMAINDER 2);

CREATE TABLE traffic_violations_p_hash_p4
    PARTITION OF traffic_violations_p_hash
    FOR VALUES WITH (MODULUS 5, REMAINDER 3);

CREATE TABLE traffic_violations_p_hash_p5
    PARTITION OF traffic_violations_p_hash
    FOR VALUES WITH (MODULUS 5, REMAINDER 4);
```

Partitioning

- The types of partitioning described above are generically called declarative partitioning.
- It is not possible to combine types of partitioning or multi-column partitioning at the level of a single table. On the other hand, the partition can already be partitioned according to another condition, since the partition is also a table. This is how nested partitioning is implemented.
- Although all partitions must have the same columns as the partitioned parent table, each partition can have its own indexes, constraints, and default values independently of the others.
- All rows inserted into a partitioned table are redirected to the appropriate sections, depending on the column values of the partitioning key. If, when changing the values of the partition key in a row, it ceases to meet the constraints of the original section, this row is moved to another section.
- You cannot convert a regular table to a partitioned one and vice versa.

Partitioning

```
-- this is impossible
CREATE TABLE measurement (
    city_id         INT NOT NULL,
    logdate         DATE NOT NULL,
    peaktemp        INT,
    unitsales       INT
) PARTITION
    BY RANGE (logdate),
    BY RANGE (unitsales);
```

Partitioning using inheritance

Although embedded declarative partitioning is useful in many frequently occurring situations, there are circumstances that require a more flexible approach. In this case, partitioning can be implemented by applying a table inheritance mechanism, which will provide a number of features unsupported by declarative partitioning, for example:

- With declarative partitioning, all partitions must have exactly the same set of columns as the partitioned table, whereas normal table inheritance allows for additional columns in child tables that are missing from the parent.
- The table inheritance mechanism supports multiple inheritance.
- With declarative partitioning, only list, range, and hash partitioning is supported, whereas with table inheritance, data can be divided according to any criteria selected by the user.

Partitioning using inheritance example

```
CREATE TABLE measurement(
    city_id    INT NOT NULL,
    logdate    DATE NOT NULL,
    peaktemp   INT,
    unitsales  INT
);

CREATE TABLE measurement_y2006m02(
    CHECK(logdate >= DATE '2006-02-01'
AND logdate < DATE '2006-03-01')
) INHERITS (measurement);


CREATE TABLE measurement_y2006m03(
    CHECK(logdate >= DATE '2006-03-01'
AND logdate < DATE '2006-04-01')
) INHERITS (measurement);
...
```

```
CREATE TABLE measurement_y2007m11(
    CHECK(logdate >= DATE '2007-11-01' AND
logdate < DATE '2007-12-01')
) INHERITS (measurement);

CREATE TABLE measurement_y2007m12(
    CHECK(logdate >= DATE '2007-12-01' AND
logdate < DATE '2008-01-01')
) INHERITS (measurement);

CREATE TABLE measurement_y2008m01(
    CHECK(logdate >= DATE '2008-01-01' AND
logdate < DATE '2008-02-01')
) INHERITS (measurement);
...
CREATE TABLE measurement_city_id1(
    CHECK(city_id = 1)
) INHERITS (measurement);
```

Replication

Replication is a mechanism for synchronizing the contents of multiple copies of an object (for example, the contents of a database). Replication is a process that refers to copying data from one source to another (or to many others) and vice versa.

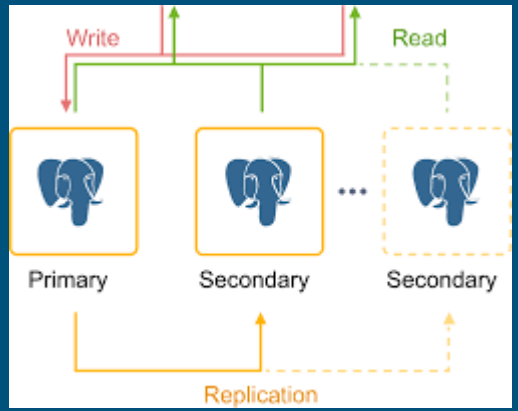
When can replication be useful?

- Fault tolerance. If you have several replicas and one server fails, the application/service/bank/global financial infrastructure/Google/whatever will not crash with a bang, but will continue to work.
- Efficiency. If one service will process 1000 requests per second or 3 for 1000, there is a difference.

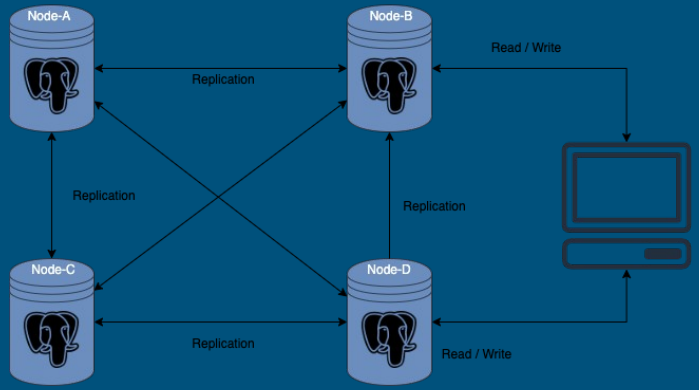
Usually, when replicating, they talk about master and slave. In the master-slave replication mode, there is one master server that processes requests for data changes, and several slave servers that process read requests.

Replication

Usually, when replicating, they talk about master and slave. In the master-slave replication mode, there is one master server that processes requests for data changes, and several slave servers that process read requests.



If the master fails, one of the slaves becomes the master. In master-master mode, all servers are equivalent and can handle any requests. But it is more complicated and if one of the servers "dies", you can get data corruption (inconsistency).



Horizontal scalability

Disadvantages of the partitioning solution

Partitioning eliminates the problem of working with big data, but does not completely solve it. For example, there may be so much data that it does not fit on one machine. Or there can be a lot of requests for writing/reading data and the requests themselves can be very time-consuming, and vertical scaling has its limits (RAM expansion, more spacious disks, more powerful processor).

Accessing tables on other servers

Not all data distributed across servers may need to be replicated or updated in real time. The data can be mostly static, intended for reference, search, or historical data, etc. Such data can be accessed from the main OLTP/OLAP servers using foreign data wrappers (FDW).

FDWs allow you to work with "other people's data", which can be located anywhere outside the Postgres server. The ability to work with data from another Postgres server is implemented using the postgres_fdw extension available in PostgreSQL.

Postgres FDW

```
srcdb=# create table srct (a int primary key);
CREATE TABLE

srcdb=# insert into srct (a) select generate_series(1,
100);
INSERT 0 100
```

At a destination server, you can setup a foreign table srct, which acts a proxy table for the actual srct table that lives in our source database:

```
destdb=# create extension postgres_fdw;
CREATE EXTENSION

destdb=# create server src foreign data wrapper
postgres_fdw options (host '/tmp', port '6000',
dbname 'srcdb');
CREATE SERVER

destdb=# create user mapping for
current_user server src;
CREATE USER MAPPING

destdb=# import foreign schema public
limit to (srct) from server src into public;
IMPORT FOREIGN SCHEMA

destdb=# select count(*) from srct;
 count
-------
   100
(1 row)
```

Postgres FDW

Suppose there is a source database with a table, from example in the last slide.
The external table does not take up space and does not contain data — it just serves as a wrapper to link to a real table located somewhere else. The postgres_fdw extension of the target Postgres server will establish and maintain a connection with the source Postgres server, converting each request that includes an external table into appropriate network calls.

Postgres FDW

- An external table can work seamlessly with regular local tables, as in this connection:

```
destdb=# create table destt (b int primary key, c text);
CREATE TABLE

destdb=# insert into destt (b,c) values (10,'foo'),
(20,'bar');
INSERT 0 2

destdb=# select a,b,c from srct join destt on srct.a =
destt.b;
 a  | b  |  c
----+----+-----
 10 | 10 | foo
 20 | 20 | bar
(2 rows)
```

Postgres FDW

The main task of FDW is to transfer work to a remote server as much as possible and minimize the amount of data transferred back and forth between the two servers. For example, you want the remote server to process the data constraint rather than extract all rows and then apply the data constraint locally. However, given the complexity of SQL, as well as the PostgreSQL query scheduler and executor, this is not an easy task. Efficiency continues to improve with each release, but some queries may take too long or more working memory than you expect.

Materialized Views + Foreign Data Wrappers

Depending on your use case, combining materialized views with FDW can offer a reasonable balance between replicating the full table and having it completely remote (external) access. A materialized view can effectively function as a local cache, which, in turn, can be used together with local tables to ensure local-level performance.

```
destdb=# create materialized
view destmv as select a,b,c from
srct join destt on srct.a = destt.b;
SELECT 2

destdb=# select * from destmv;
 a  | b  |  c
----+----+-----
 10 | 10 | foo
 20 | 20 | bar
(2 rows)
```

The "cache" can be updated at any time, periodically or otherwise, using the usual "REFRESH MATERIALIZED VIEW" command. As a bonus, you can define (local) indexes in the view to further speed up queries.

Distribution of rows on servers

- Sharding rows of a single table on multiple servers while simultaneously providing SQL clients with a unified interface for a regular table is perhaps the most popular solution for working with large tables. This approach simplifies applications and makes database administrators work harder!
- Splitting tables into parts so that queries work only with the corresponding rows, preferably in parallel, is the basic principle of segmentation. PostgreSQL v10 introduced the partitioning feature, which has since undergone many improvements and has become widespread.
- Vertical scaling using partitioning involves creating partitions in different tablespaces (on different disks). Horizontal scaling involves combining partitioning and FDW

Partitioning + FDW

Going with the example from the Postgres documentation, let's create the partition root table measurement having one local partition table and one foreign partition table:
Using the example from the Postgres documentation, let's create a root partitioned measurement table having one local table and one external table:

Partitioning + FDW

```
destdb=# CREATE TABLE measurement
(
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);
CREATE TABLE

destdb=# CREATE TABLE
measurement_y2023
PARTITION OF measurement
FOR VALUES FROM ('2023-01-01') TO
('2024-01-01');
CREATE TABLE
```

```
destdb=# CREATE FOREIGN TABLE
measurement_y2022
PARTITION OF measurement
FOR VALUES FROM ('2022-01-01') TO ('2023-01-01')
SERVER src;
CREATE FOREIGN TABLE
```

The foreign table is only a proxy, so the actual table itself must be present on the foreign server:

```
srcdb=# CREATE TABLE measurement_y2022 (
    city_id         int not null,
    logdate         date not null
        CHECK (logdate >= '2022-01-01' and logdate <=
'2023-01-01'),
    peaktemp        int,
    unitsales       int
);
CREATE TABLE
```

Partitioning + FDW

Now we can insert rows into the root table and direct them to the appropriate section. You can see that the SELECT query performs both local and external scans and combines the results.

```
destdb=# insert into measurement (city_id, logdate, peaktemp, unitsales)
values (1, '2022-01-03', 66, 100), (1, '2023-01-03', 67, 300);
INSERT 0 2

destdb=# select * from measurement;
 city_id |  logdate   | peaktemp | unitsales
---------+------------+----------+-----------
       1 | 2022-01-03 |       66 |       100
       1 | 2023-01-03 |       67 |       300
(2 rows)

destdb=# explain select * from measurement;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Append  (cost=100.00..219.43 rows=3898 width=16)
   ->  Foreign Scan on measurement_y2022 measurement_1  (cost=100.00..171.44 rows=2048 width=16)
   ->  Seq Scan on measurement_y2023 measurement_2  (cost=0.00..28.50 rows=1850 width=16)
(3 rows)
```

However, both partitioning and external tables still have implementation limitations in PostgreSQL, which means that this method works satisfactorily only for simple tables and basic queries.

OLAP

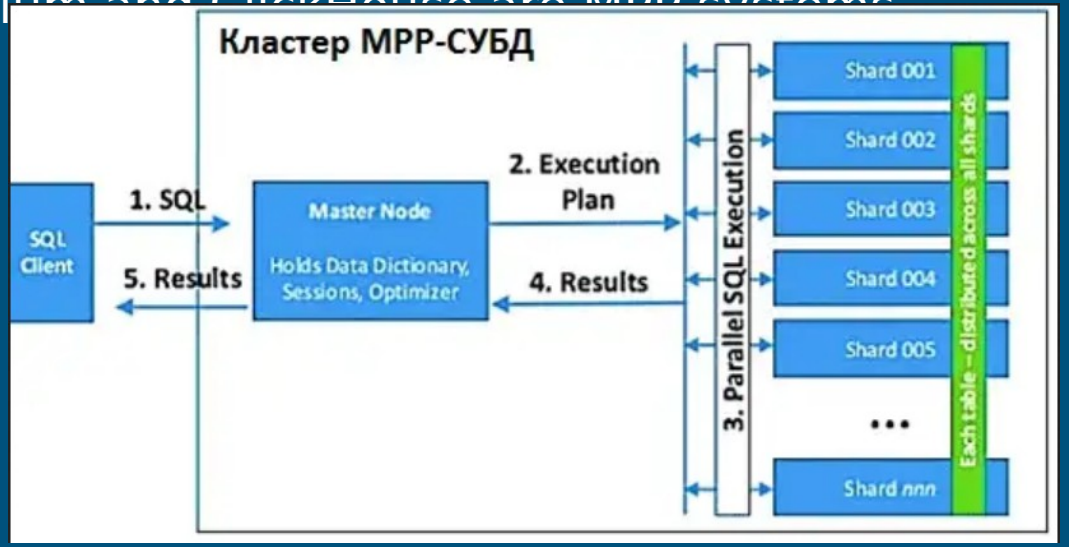
Postgres was developed with an eye on scenarios of frequent small data rates and multiple point reads (several rows from tables). This usage scenario is called OLTP (OnLine Transaction Processing).
On the other hand, it may be necessary to build reports for a long period with aggregations and connections of tens, hundreds, and even thousands of tables. Such usage scenarios are called OLAP (OnLine Analytical Processing). They have their own storages.

De facto standards in the industry now:

- Greenplum
- ClickHouse
- Vertica (proprietary, not considered)

MPP systems

MPP (massive parallel processing) is a massively parallel architecture. The main feature of this architecture is that the memory is physically divided. Roughly speaking, data is broken into pieces, stored and processed on different machines, and, if necessary, data is exchanged between each other. Greenplum and ClickHouse are MPP systems



Greenplum

- Official website
- Code
- Documentation
- Short reference
- Tutorial for running a cluster locally

GREENPLUM DATABASE®

Clickhouse

- Official website
- Code
- Documentation
- Online compiler
- Short reference
- Tutorial for running a cluster locally