creating_temporary_tables

```sql
-- Correlation between revenues and profit
SELECT CORR(revenues, profits) AS rev_profits,
    -- Correlation between revenues and assets
    CORR(revenues, assets) AS rev_assets,
    -- Correlation between revenues and equity
    CORR(revenues, equity) AS rev_equity
 FROM fortune500;
```

Explanation:

- This SQL query calculates the Pearson correlation coefficient between different financial metrics from a table named fortune500. The CORR() function computes the correlation, showing the linear relationship between two variables. The query returns three correlation values: the correlation between revenues and profits (rev_profits), revenues and assets (rev_assets), and revenues and equity (rev_equity). A value of 1 indicates a perfect positive correlation, -1 a perfect negative correlation, and 0 no linear correlation.

```sql
-- What groups are you computing statistics by?
SELECT
  sector,
  -- Select the mean of assets with the avg function
  AVG(assets) AS mean,
  -- Select the median
  percentile_disc(0.5) WITHIN GROUP (ORDER BY assets) AS median
FROM
  fortune500
-- Computing statistics for each what?
GROUP BY
  sector
-- Order results by a value of interest
ORDER BY
  mean;
```

Explanation:

- This SQL query calculates the mean and median assets for each sector in the fortune500 table. It groups the data by sector and then computes the average (AVG) and median (percentile_disc(0.5) WITHIN GROUP (ORDER BY assets)) assets within each group. The results are ordered by the mean assets in ascending order.

```
-- To clear table if it already exists; fill in name of temp table
DROP TABLE IF EXISTS profit80;

-- Create the temporary table
CREATE TEMP TABLE profit80 AS
 -- Select the two columns you need; alias as needed
 SELECT sector,
      percentile_disc(0.8) WITHIN GROUP (ORDER BY profits) AS pct80
  -- What table are you getting the data from?
  FROM fortune500
 -- What do you need to group by?
  GROUP BY sector;

-- See what you created: select all columns and rows from the table you created
SELECT *
 FROM profit80;
```

Explanation:

- This SQL code first drops a temporary table named profit80 if it already exists. Then, it creates a new temporary table profit80 by selecting the sector and the 80th percentile of profits (calculated using percentile_disc) for each sector from the fortune500 table. Finally, it selects all data from the newly created profit80 table to display the results. The percentile_disc function finds the value at the 80th percentile of the profits within each sector.

```
-- Drop the temporary table if it exists
DROP TABLE IF EXISTS profit80;

-- Create a temporary table to store the 80th percentile profit for each sector
CREATE TEMP TABLE profit80 AS
 SELECT sector,
      percentile_disc(0.8) WITHIN GROUP (ORDER BY profits) AS pct80
  FROM fortune500
  GROUP BY sector;

-- Select columns, aliasing as needed
SELECT f.title, f.sector,
     f.profits,
     f.profits / p.pct80 AS ratio
-- Join the tables
 FROM fortune500 f
     INNER JOIN profit80 p
     ON f.sector = p.sector
-- Select rows where profits are greater than the 80th percentile profit
 WHERE f.profits > p.pct80;
```

Explanation:

- This SQL code calculates the 80th percentile of profits for each sector in the fortune500 table and then identifies companies whose profits exceed that percentile. It does this by first creating a temporary table (profit80) containing the 80th percentile profit for each sector. Then, it joins this temporary table back to the original table to filter and display companies that surpass their sector's 80th percentile profit, along with their profit ratio relative to that percentile.

```sql
-- To clear table if it already exists
DROP TABLE IF EXISTS startdates;

-- Create temp table syntax
CREATE TEMP TABLE startdates AS
-- Compute the minimum date for each tag
SELECT tag,
     MIN(date) AS mindate
 FROM stackoverflow
 -- Group by tag to compute the min date for each tag
 GROUP BY tag;

 -- Look at the table you created
 SELECT *
  FROM startdates;
```

Explanation:

- This SQL code first drops a table named startdates if it already exists. Then, it creates a temporary table called startdates. This new table contains the minimum date (mindate) for each unique tag from the stackoverflow table. Finally, it selects all data from the newly created startdates table to display the results. The GROUP BY clause is crucial for finding the minimum date for each distinct tag.

```sql
-- To clear table if it already exists
DROP TABLE IF EXISTS startdates;

-- Create the temp table with minimum date for each tag
CREATE TEMP TABLE startdates AS
SELECT tag,
     MIN(date) AS mindate
 FROM stackoverflow
 GROUP BY tag;

-- Select the required columns, including the question counts on the min and max days
SELECT startdates.tag,
     startdates.mindate,
     so_min.question_count AS min_date_question_count,
     so_max.question_count AS max_date_question_count,
     -- Compute the change in question_count (max - min)
     so_max.question_count - so_min.question_count AS change
 FROM startdates
     -- Join startdates to stackoverflow with alias so_min for the minimum date
     INNER JOIN stackoverflow AS so_min
       ON startdates.tag = so_min.tag
      AND startdates.mindate = so_min.date
     -- Join to stackoverflow again with alias so_max for the maximum date
     INNER JOIN stackoverflow AS so_max
       ON startdates.tag = so_max.tag
      AND so_max.date = '2018-09-25';
```

Explanation:

- This SQL code analyzes a stackoverflow table (presumably containing tag, date, and question count data). It finds the minimum date for each tag, then compares the question count on that minimum date to the question count on '2018-09-25' for each tag, showing the difference. It does this by creating a temporary table (startdates) to store the minimum dates and then joining this table back to the original table twice to get the question counts for both the minimum date and '2018-09-25'.

```
DROP TABLE IF EXISTS correlations;

-- Create a temporary table to store correlation results
CREATE TEMP TABLE correlations AS
SELECT
  'profits'::varchar AS measure,  -- Column specifying the base measure ('profits')
  corr(profits, profits) AS profits,        -- Correlation of profits with itself (will be 1)
  corr(profits, profits_change) AS profits_change, -- Correlation of profits with its change
  corr(profits, revenues_change) AS revenues_change -- Correlation of profits with revenue change
FROM fortune500;
```

Explanation:

- This SQL code snippet calculates the correlation between the 'profits' column and other columns ('profits_change', 'revenues_change') in a table named fortune500. It uses the corr() function to compute the correlation coefficients. The results are stored in a temporary table called correlations. The ::varchar casts the string 'profits' to a varchar type for consistency. The temporary table is dropped automatically at the end of the session.

```
DROP TABLE IF EXISTS correlations;

CREATE TEMP TABLE correlations AS
SELECT 'profits'::varchar AS measure,
    corr(profits, profits) AS profits,
    corr(profits, profits_change) AS profits_change,
    corr(profits, revenues_change) AS revenues_change
 FROM fortune500;

-- Add a row for profits_change
INSERT INTO correlations
SELECT 'profits_change'::varchar AS measure,
    corr(profits_change, profits) AS profits,
    corr(profits_change, profits_change) AS profits_change,
    corr(profits_change, revenues_change) AS revenues_change
 FROM fortune500;

-- Repeat the above, but for revenues_change
INSERT INTO correlations
SELECT 'revenues_change'::varchar AS measure,
    corr(revenues_change, profits) AS profits,
    corr(revenues_change, profits_change) AS profits_change,
    corr(revenues_change, revenues_change) AS revenues_change
 FROM fortune500;
```

Explanation:

- This SQL code calculates the correlation between three variables (profits, profits_change, and revenues_change) from a table named fortune500. It creates a temporary table called correlations to store the results. The code iteratively inserts rows, each representing a measure and its correlation with the other measures. The corr() function computes the Pearson correlation coefficient. The ::varchar casts the measure names to the correct datatype.

```
DROP TABLE IF EXISTS correlations;

CREATE TEMP TABLE correlations AS
SELECT 'profits'::varchar AS measure,
    corr(profits, profits) AS profits,
    corr(profits, profits_change) AS profits_change,
    corr(profits, revenues_change) AS revenues_change
 FROM fortune500;

INSERT INTO correlations
SELECT 'profits_change'::varchar AS measure,
    corr(profits_change, profits) AS profits,
    corr(profits_change, profits_change) AS profits_change,
    corr(profits_change, revenues_change) AS revenues_change
 FROM fortune500;

INSERT INTO correlations
SELECT 'revenues_change'::varchar AS measure,
    corr(revenues_change, profits) AS profits,
    corr(revenues_change, profits_change) AS profits_change,
    corr(revenues_change, revenues_change) AS revenues_change
 FROM fortune500;

-- Select each column, rounding the correlations
SELECT measure,
    ROUND(profits::numeric, 2) AS profits,
    ROUND(profits_change::numeric, 2) AS profits_change,
    ROUND(revenues_change::numeric, 2) AS revenues_change
 FROM correlations;
```

Explanation:

- This SQL code calculates and displays the correlation matrix for three variables: profits, profits_change, and revenues_change from a table named fortune500. It first creates a temporary table correlations to store the results. Then, it populates this table with correlation coefficients calculated using the corr() function for all pairs of variables. Finally, it selects the results, rounding the correlation coefficients to two decimal places for better readability. The ::varchar casts are used to ensure the measure column has the correct data type.