```
import sqlparse
from sqlparse.sql import IdentifierList, Identifier
from sqlparse.tokens import Keyword, DML
import json

def is_subselect(parsed):
    if not parsed.is_group:
        return False
    for item in parsed.tokens:
        if item.ttype is DML and item.value.upper() == 'SELECT':
            return True
    return False

def extract_from_clauses(parsed):
    """
    Collect all table references (FROM and JOIN) to build an alias-
>full_table map.
    """
    from_seen = False
    alias_map = {}

    def parse_alias(token_text):
        """
        Given something like 'scd_prod.performance_seclvl_latest psl',
        return ('scd_prod.performance_seclvl_latest', 'psl').
        If there's no alias, return (that_text, None).
        """
        # Safeguard: skip if token_text is empty
        if not token_text:
            return None, None

        parts = token_text.split()
        if len(parts) == 2:
            return parts[0], parts[1]
        elif len(parts) == 1:
            return parts[0], None
        else:
            # More or fewer parts than expected.
            # e.g. if it included conditions or was a weird token
            return None, None

    for token in parsed.tokens:
        # Once we've encountered FROM...
        if from_seen:
            if is_subselect(token):
                # If there's a subselect, recursively process it
                for sub_alias_map in extract_from_clauses(token):
                    alias_map.update(sub_alias_map)

            # If we see the next major clause (WHERE, GROUP, ORDER,
```

```
etc.), we stop
            elif token.ttype is Keyword and token.value.upper() in [
                'WHERE', 'GROUP', 'ORDER', 'HAVING', 'LIMIT', 'UNION'
            ]:
                break

            else:
                # Parse JOIN tokens or direct references
                token_str = token.value.strip()

                # Skip if it's empty or pure whitespace
                if not token_str:
                    continue

                if token_str.upper().startswith(('INNER JOIN', 'LEFT
JOIN', 'RIGHT JOIN', 'FULL JOIN')):
                    # e.g. "INNER JOIN scd_prod.model_portfolio_latest
mp"
                    # We'll remove the "JOIN" portion
                    try:
                        join_type, table_expr =
token_str.split('JOIN', 1)
                    except ValueError:
                        # If the string doesn't split cleanly, skip it
                        continue
                    table_expr = table_expr.strip()
                    t, a = parse_alias(table_expr)
                    if t:
                        if a:
                            alias_map[a] = t
                        else:
                            alias_map[t] = t

                elif token_str.upper().startswith('ON'):
                    # skip the ON clause
                    continue

                elif token_str.upper().startswith(('AND', 'OR')):
                    # skip conditions in the ON clause
                    continue

                else:
                    # This might be the main FROM table reference
                    t, a = parse_alias(token_str)
                    if t:
                        if a:
                            alias_map[a] = t
                        else:
                            alias_map[t] = t
```

```
        elif token.ttype is Keyword and token.value.upper() == 'FROM':
            from_seen = True

    # yield so we can recursively accumulate alias maps
    yield alias_map

def extract_select_columns(parsed):
    """
    Extract columns from the SELECT clause. Return a list of
identifiers
    (each might have .get_parent_name(), .get_real_name(),
and .get_alias()).
    """
    select_seen = False
    columns = []

    for token in parsed.tokens:
        if select_seen:
            # If we hit a major keyword, stop parsing SELECT
            if token.ttype is Keyword:
                break

            if isinstance(token, IdentifierList):
                for identifier in token.get_identifiers():
                    columns.append(identifier)
            elif isinstance(token, Identifier):
                columns.append(token)

        elif token.ttype is DML and token.value.upper() == 'SELECT':
            select_seen = True

    return columns

def parse_sql(sql):
    """
    Main function:
      1) parse the SQL
      2) build alias map from FROM/JOIN clauses
      3) read each column in SELECT
      4) replace alias with full table name
      5) handle AS aliases if present
    Returns a list of tuples: (final_label, fully_qualified_column)
    """
    statements = sqlparse.parse(sql)
    if not statements:
        print("No SQL statements found.")
        return []

    # We'll parse just the first statement
    statement = statements[0]
```

```
    alias_map = {}

    # Build a merged alias->table dictionary from all yields
    for partial_map in extract_from_clauses(statement):
        alias_map.update(partial_map)

    # Now find columns in the SELECT
    columns = extract_select_columns(statement)

    results = []
    for col in columns:
        # e.g. col might represent "psl.split_type_description AS
split_desc"
        real_name = col.get_real_name()      # e.g.
"split_type_description"
        parent_alias = col.get_parent_name() # e.g. "psl"
        as_alias = col.get_alias()           # e.g. "split_desc"

        if not real_name:
            # It's possibly a literal (e.g. 'SCD' AS something). We'll
skip or store it as is.
            # You can skip or handle differently. We'll store it as
final_label -> 'SCD'
            if as_alias:
                results.append((as_alias, col.value))
            else:
                results.append((col.value, col.value))
            continue

        # If there's a parent alias, convert it to a full table
        if parent_alias:
            full_table = alias_map.get(parent_alias, parent_alias)
            fully_qualified = f"{full_table}.{real_name}"
        else:
            # Possibly a literal or function call with no table
reference
            fully_qualified = real_name

        # The final label is either the "AS" alias or the real column
name
        final_label = as_alias if as_alias else real_name
        results.append((final_label, fully_qualified))

    return results

if __name__ == "__main__":
    # 1) Read your entire 'cpp.sql' from disk
    sql_file_path = "cpp.sql"  # Adjust if needed
    with open(sql_file_path, "r", encoding="utf-8") as f:
        sql_content = f.read()
```

```
    # 2) Parse the SQL and get a list of (label,
fully_qualified_column)
    mappings = parse_sql(sql_content)

    # 3) Print the results to console
    print("Field Mapping:")
    if not mappings:
        print("No columns found or no mapping possible.")
    for label, fq_col in mappings:
        print(f"{label} -> {fq_col}")

    # 4) Write results to a text file
    with open("mapping.txt", "w", encoding="utf-8") as txt_file:
        for label, fq_col in mappings:
            txt_file.write(f"{label} -> {fq_col}\n")

    # 5) Write results to JSON
    mapping_dict = {label: fq_col for (label, fq_col) in mappings}
    with open("mapping.json", "w", encoding="utf-8") as json_file:
        json.dump(mapping_dict, json_file, indent=2)
```