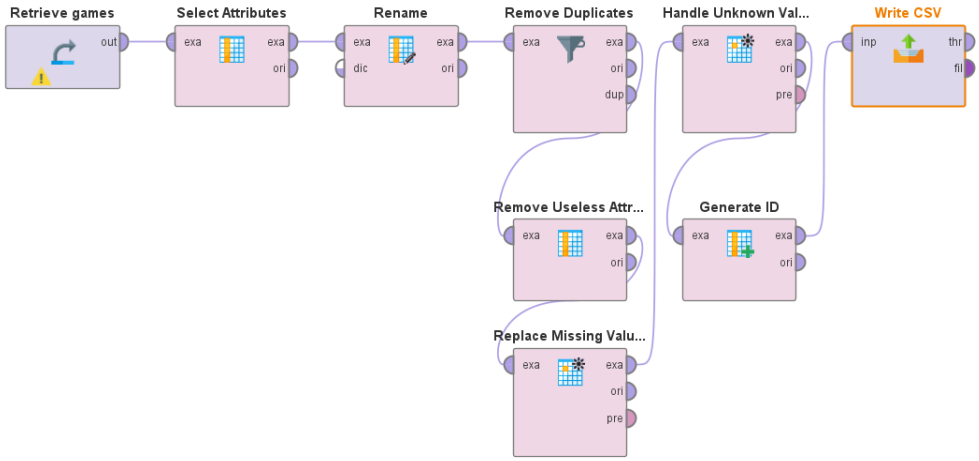
STADVDB MCO1 ETL Pipeline Using RapidMiner and Python Pandas

The ETL process involves using tools like Altair AI Studio (RapidMiner), and Python with libraries such as pandas and MySQL to extract, clean, transform, and load data into the OLAP application. Most of the extraction and cleaning is done in RapidMiner, as shown in the processing pipeline below.



The dataset is imported from a CSV file using RapidMiner's CSV operator for convenience. The Select Attributes operator is applied to choose relevant columns, followed by the Rename operator to align column names with the data warehouse schema. To clean the data, operators like Remove Duplicates, Remove Useless Attributes, and Replace Missing Values eliminate redundant or incomplete data. An ID is generated for each game record, and the cleaned data is exported as a CSV file.

This CSV file is then processed in Python, where it is loaded into a DataFrame using pandas. The Python script further prepares the data for a MySQL database by extracting unique genres and categories, splitting them into individual attributes to form dimension tables (dim_genre_set and dim_category_set).

The functions in the image below are the additional functions we used for data pre-processing. Function parse_date converts the *release_date* from a string to a datetime variable.

```python
# Load data from CSV file
csv_file_path = 'final_cleaned.csv'
df = pd.read_csv(csv_file_path, sep=';', encoding='latin1')

# Drop rows with missing data in important columns
df = df.dropna(subset=['user_score', 'genres', 'categories'])

# Clean user_score: remove non-numeric values and ensure no nulls
def clean_user_score(value):
    if pd.isna(value) or value > 100:  # Cap user_score at 100
        return 100
    return float(value) if isinstance(value, (int, float)) else None

# Clean playtime columns: convert to integers and handle non-numeric values
def clean_playtime(value):
    return 0 if pd.isna(value) else int(value)  # No nulls allowed

# Parse date format to YYYY-MM-DD for MySQL
def parse_date(value):
    try:
        # First try YYYY-MM-DD format
        return datetime.strptime(value, '%Y-%m-%d').strftime('%Y-%m-%d')
    except ValueError:
        try:
            # Try converting string format 'Oct 21, 2008' to '2008-10-21'
            return datetime.strptime(value, '%b %d, %Y').strftime('%Y-%m-%d')
        except ValueError:
            # If parsing fails, return None
            return None
```

The functions *normalize_name()* and *get_unique_entries()* are used to transform and pivot unique entries from the genre and category fields into table columns, ensuring they follow a lowercase, underscore-separated format.

```python
# Normalize genres and categories
def normalize_name(name):
    return re.sub(r'[^a-zA-Z0-9_]', '_', name.strip().lower())

def get_unique_entries(df, column):
    unique_entries = set()
    for entry in df[column].dropna():
        entries = entry.split(',')
        for item in entries:
            unique_entries.add(item.strip())
    return list(unique_entries)

# Extract and normalize unique genres and categories
unique_genres = get_unique_entries(df, 'genres')
unique_categories = get_unique_entries(df, 'categories')

print("Unique Genres: ", unique_genres)
print("Unique Categories: ", unique_categories)
```

This loop in our code iterates through the cleaned and transformed data frames to insert each row into their respective tables.

```python
for idx, row in df.iterrows():
    genre_flags = {}  # Initialize an empty dictionary
    genres_in_row = row['genres'].split(',')  # Split genres in the current row
    normalized_genres_in_row = sorted(normalize_and_check_duplicates(genres_in_row))
    for genre in normalized_genres:
        if genre in normalized_genres_in_row:
            genre_flags[genre] = 1  # Set flag to 1 if genre is present
        else:
            genre_flags[genre] = 0  # Set flag to 0 if genre is not present
    genre_values = ', '.join(str(genre_flags[genre]) for genre in normalized_genres)

    category_flags = {}  # Initialize an empty dictionary
    categories_in_row = row['categories'].split(',')  # Split categories from the row
    normalized_categories_in_row = sorted(normalize_and_check_duplicates(categories_in_row))
    for category in normalized_categories:
        if category in normalized_categories_in_row:
            category_flags[category] = 1  # Set flag to 1 if category is present
        else:
            category_flags[category] = 0  # Set flag to 0 if category is not present

    category_values = ', '.join(str(category_flags[category]) for category in normalized_categories)

    # Insert into dim_genre_set and dim_category_set with the same ID
    cursor.execute(f"INSERT INTO dim_genre_set (id, {', '.join(f'`{genre}`' for genre in normalized_genres)}) VALUES ({idx+1}, {genre_values})")
    cursor.execute(f"INSERT INTO dim_category_set (id, {', '.join(f'`{category}`' for category in normalized_categories)}) VALUES ({idx+1}, {category_values})")

    # Insert into fact_games
    cursor.execute("""INSERT INTO fact_games
                      (category_id, genre_id, release_date, price, positive_reviews, negative_reviews,
                      user_score, metacritic_score, average_playtime_forever, average_playtime_2weeks,
                      median_playtime_forever, median_playtime_2weeks)
                      VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)""",
                   (idx+1, idx+1, row['release_date'], row['price'], row['positive_reviews'],
                    row['negative_reviews'], row['user_score'], row['metacritic_score'],
                    row['average_playtime_forever'], row['average_playtime_2weeks'],
                    row['median_playtime_forever'], row['median_playtime_2weeks']))

# Commit the transaction
connection.commit()
```

Following the data insertions, physical columns and indexes were created for query optimization using the code below.

```
# Add new columns for release year and release month
cursor.execute("""
    ALTER TABLE fact_games
    ADD COLUMN release_year INT,
    ADD COLUMN release_month INT;
""")

# Update release_year and release_month based on release_date
cursor.execute("""
    UPDATE fact_games
    SET release_year = YEAR(release_date),
        release_month = MONTH(release_date);
""")

# Create indexes on fact_games
cursor.execute("""
    CREATE INDEX idx_games_user_playtime_id ON fact_games (user_score, average_playtime_forever, id);
""")
cursor.execute("""
    CREATE INDEX idx_games_release_year_month ON fact_games (release_year, release_month);
""")

# Create indexes on all columns in dim_genre_set
for genre in normalized_genres:
    index_name = f"idx_genre_{genre}"
    cursor.execute(f"CREATE INDEX {index_name} ON dim_genre_set (`{genre}`)")

# Create indexes on all columns in dim_category_set
for category in normalized_categories:
    index_name = f"idx_category_{category}"
    cursor.execute(f"CREATE INDEX {index_name} ON dim_category_set (`{category}`)")

# Commit these changes to the database
connection.commit()
print("Columns added, updated, and indexes created successfully.")
```