---

## Definitions

- **Transaction**: A set of items.
- **Transactional Database**: A collection of transactions.
- **Temporal Transaction**: A set of items with a temporal occurrence ID, which specifies when the transaction occurs.
- **Temporal Database**: A collection of temporal transactions.

## Temporal Database File Structure

A temporal database file contains:

- A **temporal occurrence ID** followed by the items in the transaction, all separated by a delimiter.
- Transactions are separated by newline (`\n`) characters.
- Eg: `10 1 2 3\n` where `10` is the `temporal id`

If the database file only pads at the end, the **Compressed Sparse Row (CSR)** representation would still serve as a space-efficient mechanism to avoid processing or storing unnecessary padding. Here's how it aligns with your context:

---

## CSR Format for Temporal Databases (with End Padding)

1. **What is Stored**:
   - **Data (`data`)**: Contains all items from the transactions (excluding padding).
   - **Row Pointers (`row_ptr`)**: Points to the starting index of each transaction in the `data` array.

---

## Example Transactions Database

The file contains the following transactions:

```
1 2\n
1 2 3\n
2 3 4 5\n
```

---

## CSR Representation

### 1. Data (`data`)

The `data` array contains all items across transactions, listed sequentially:

```
data = [1, 2, 1, 2, 3, 2, 3, 4, 5]
```

### 2. Row Pointers (`row_ptr`)

The `row_ptr` array points to the starting index of each transaction in the `data` array.

- Transaction 1: Starts at index `0` and has 2 items.
- Transaction 2: Starts at index `2` and has 3 items.
- Transaction 3: Starts at index `5` and has 4 items.

So, the `row_ptr` array is:

```
row_ptr = [0, 2, 5, 9]
```

#### 3. Column Indices (`col_index`)

Since this is a temporal database and not a matrix, the column indices are unnecessary. Each row (transaction) is sequential.

---

### Summary of CSR Representation

- **Data**: `[1, 2, 1, 2, 3, 2, 3, 4, 5]`
- **Row Pointers**: `[0, 2, 5, 9]`

This format efficiently removes any need for padding and allows quick access to any transaction.

---

### Reconstructing Transactions

Using the `row_ptr` array:

- Transaction 1: `data[0:2] = [1, 2]`
- Transaction 2: `data[2:5] = [1, 2, 3]`
- Transaction 3: `data[5:9] = [2, 3, 4, 5]`

---

## GPU Parser

---

#### Assumptions

1. The temporal database file is UTF-8 encoded. Each byte represents a single ASCII character.
    - For example, the transaction `10 1 2 3\n` contains 9 characters/bytes: `1`, `0`, , `1`, , `2`, , `3`, and `\n`.
2. Values are sequences of characters separated by a delimiter ( ) or newline (`\n`).
3. A newline (`\n`) marks the end of a transaction; all preceding values belong to that transaction.
4. All items are numerical and range from 1 to *n* (where *n* is the total number of unique items).
5. Temporal occurrence IDs are integers immediately preceding the first space in each transaction.

---

#### Implementation Steps

1. **Load Dataset**
   Load the file into GPU-accessible memory using **Kvikio** for GPU Direct Storage (GDS).
   - If GPU Direct Storage is not supported, fallback to **POSIX I/O** for transferring data from the file to GPU memory.

2. **Count Transactions**
   - Assign each GPU thread to process a single byte of the file.
   - Each thread checks if its assigned byte is a newline (`\n`).
   - Increment a global counter for each newline detected. This gives the total number of transactions.

3. **Allocate Memory for Indexes**
   - Allocate memory to store the start and end indexes of each transaction (newline positions).
   - Each GPU thread identifies newline characters and stores their positions in the index array.

4. **Identify Temporal Occurrence IDs**
   - After identifying the start of each transaction (positions of newlines), extract the temporal occurrence ID as the first sequence of digits before the first space.
   - For example:

     ```
     Input: 1 1 2 3\n
     Temporal ID: 1
     ```

5. **Count Delimiters per Transaction**
   - Each thread is assigned a transaction (using indexes from Step 3).
   - Count the number of spaces ( ) within the transaction to determine the number of items it contains.
   - Use a prefix sum to calculate cumulative item counts across all transactions.

6. **Allocate Memory for Integer Database**
   - Based on the total item count from Step 5, allocate memory to store:
     - **Data array (`data`)**: Stores all items in a flat array.
     - **Row pointers (`row_ptr`)**: Stores starting positions of each transaction in the `data` array (CSR format).
   - Allocate space for storing temporal occurrence IDs separately.

7. **Convert to Integer Database**
   - Each thread processes one transaction from the character database using its `thread id` and transaction start/end indexes from Step 3.
   - Steps for conversion:
     1. Parse the temporal occurrence ID as an integer.
     2. Extract each item from the transaction and convert it to an integer.

3. Store the integers in the `data` array and update `row_ptr`.

---

### Example Database

Input:

```
1 1 2 3\n
2 1 2 4 5\n
4 2 4\n
5 1 2 3 4 5\n
```

#### Step-by-Step Processing

1. **Transaction Indexes**
   - Newline positions: `[7, 18, 26, 40]`
   - Transaction starts: `[0, 8, 19, 27]`
2. **Temporal Occurrence IDs**
   - Extracted IDs: `[1, 2, 4, 5]`
3. **Count Delimiters**
   - Delimiter counts: `[3, 4, 2, 5]`
   - Total items: `14`
4. **Allocate Memory**
   - `data` size: `14`
   - `row_ptr`: `[0, 3, 7, 9, 14]`
5. **Converted Integer Database**
   - **Data (`data`)**: `[1, 2, 3, 1, 2, 4, 5, 2, 4, 1, 2, 3, 4, 5]`
   - **Row Pointers (`row_ptr`)**: `[0, 3, 7, 9, 14]`
   - **Temporal IDs**: `[1, 2, 4, 5]`

---

### Final Result (CSR Format)

- **Temporal IDs**: `[1, 2, 4, 5]`
- **Row Pointers**: `[0, 3, 7, 9, 14]`
- **Data**: `[1, 2, 3, 1, 2, 4, 5, 2, 4, 1, 2, 3, 4, 5]`