CS764 Final Project: External Merge Sort

Overview

We implemented external sort, simulating a real machine's environment with cache, memory and disk. These are our configurations, also in `./include/defs.h`:

```
ROW_LENGTH = 10
CACHE_CAPACITY = 16
MEMORY_CAPACITY = 1024
MEMORY_FAN_IN = 32
```

How to run our program

Setup

Make sure you have a `./disk` folder created, then run the `Makefile`:

```
make clean && make
```

Executables

```
./build/Main.exe
./build/Test.exe
```

`Main.exe` Usage

```
./build/Main.exe --row_number=100000
```

```
./build/Main.exe --row_number=66560 --value_range=100 --predicates=0:50,1:20
```

Command-Line Arguments

- **`--row_number=<num>`**:
  Specifies how many rows to generate and process.
  Optional. Default value is 100.
- **`--value_range=<num>`**:
  Specifies the range of generated values (0 to `<num>`).
  Optional. If not specified, the default is typically used.
- **`--predicates=index:value,index:value,...`**:
  Specifies a list of predicates to filter rows.
  Each predicate is given as `index:value`, where both are integers.
  Optional. If not specified, no filtering is applied.

Testing

1. Correctness

- Tree of Loser Priority Queue (used for both internal and external sorting): run Unit Test `./build/Test.exe`.
- You can run:
  - `./build/Main.exe --row_number=16` internal cache sort
  - `./build/Main.exe --row_number=1000` cache to memory sort
  - `./build/Main.exe --row_number=10000` (or larger) memory to disk sort
- A file `witness.output` should be created containing Witness Iterator's output (sort order check & parity check).
- You can also using test script `tools/experiment.py`
  - Write test settings in `experiment.yaml` under file folder `tools/experiment` or using the default `test.yaml`.

    ```
    row_num: [0, 1]                    -- A list of row_num paras
    value_range: [100, 1000]           -- A list of value_range paras
    predicates: ["1:100", "1:100,2:50"] -- Predicates para
    ```

  - Run test using command `$ python ./tools/experiment.py test`.

- Check the output's `parity` and `order` fields to see if they are correct.

2. Performance

- Row & Column Comparisions: We implemented and use Tree of Loser and OVC, and our sort operation functions correctly.
- Graceful Degradation: We make sure our optimization works as expected by checking the total amount of disk flushs. Here we chose how many rows in total was written to disk as the metric (because our fan-ins and memory size may vary). You can check this by looking at `Main.exe` output, there should be a line as follow: `src/Sort.cpp:105:~SortIterator total writes to disk (in rows unit): 1` . Below are some results we collected and reasonings, note that memory size is 1024 and memory fan-in is 32:
  - `./build/Main.exe --row_number=1` : 1 row writes.
  - `./build/Main.exe --row_number=1023` : 1023 rows writes.
  - `./build/Main.exe --row_number=1025` (over memory size, under max fan-in): 2050 rows writes. 1025 of these writes are initial writes from memory to disk (unsorted). So we consider the remaining: `ceil((2025-1025)/1024)=2` times memory writes.
  - `./build/Main.exe --row_number=32768` (W=F) (32 memory runs): 65536 row writes, `ceil((65536-32768)/1024)=32` times memory writes. This is correct.
  - `./build/Main.exe --row_number=64512` (W=2F-1) (63 memory runs): 161792 row writes, `ceil((161792-64512)/1024)=95` times memory writes. This is correct! `95 = 32 + (31+32)` .

Rubriks Reflection

We reflected the stuffs we have implemented with the given rubriks:

1. Internal Sort

- We initially implemented quick sort, but later switch to merge sort after verifying that our Priority Queue functions correctly.
- We can test the correctness by sorting a small number of records.

2. External Sort

- We implemented Tree of Loser Priority Queue and OVC in `./src/PriorityQueue.cpp` .
- We have Unit Tests for this in `./test/PriorityQueueTest.h` .

3. Graceful Degradation

Our implementation is as follow: Say we start with `W` runs, `F` fan-ins. Then we calculate the initial merge `R = (W-2) % (F-1) + 2` , and merge the first `R` runs, pushing that merged result to the end of run queue (so that we merge the bigger runs later). Then we iteratively repeat this calculation for `W-R+1` remaining runs and so on. This covers the following:

- Internal → External
- 1-step merge → n-step merge
- Merge Planning

4. Cache Runs

- We use internal sort for cache runs; the result is stored in memory.
- We use ceiling `(log2 (CACHE_CAPACITY))` as the priority queue size. We get output each time we push the `CACHE_CAPACITY` of rows. We store these cache runs.

5. Memory Runs

- If the runs from cache fills memory, then we do external merge sort (merge the cache runs currently stored in memomry) then flush them to disk.
- We use ceiling `(log2 (cache_run_cnt))` as the priority size. To simplify, we ignore `CACHE_CAPACITY` here and always assume the cache-to-memory priority queue can fit in the cache; thus, we do NOT apply multi-level external sort at this part.
- For memory output buffer, Normally, we should have one page (or several) as an output buffer and flush the content back to disk periodically. To simplify, we hold an enormous array in our code and flush this array when the whole sort for a memory run is done.

6. Recycling the Priority Queue

- Priority Queue is recycled in our code, we have one for cache→memory and one for memory→disk.

7. Minimum number of row comparisions

- We used Tree of Loser Priority Queue with OVC, and this helps minimize full row comparision.
- We simply follow the code of Priority queues for database query processing

8. Minimum number of column comparisions

- In case of similar OVC, we start comparing from the similar offset.
- We simply follow the code of Priority queues for database query processing

9. Test Cases for Correctness

- Mentioned above.

10. Test Cases for Performance

- Mentioned above.

11. Iterators

- Scan: Initialize random records
- Witness: Row count, parity check, order correctness check
- Filter: We did a simple filter that takes in `index` and `value` and returns only record with `row[index] > value`.