P3 - Joins

Deadline: Wednesday, April 17, 2024

Accept Assignment: https://classroom.github.com/a/McUDU60i

Submit Assignment: https://autolab.cse.buffalo.edu/courses/cse410-s24/assessments/P3-Join-Algorithms

In this assignment, you will implement three disk-oriented join algorithms

This assignment is intended to: - Give you experience writing algorithms for memory-bound use cases - Explore the contrast between different algorithms - Implement a data structure

You should expect to spend approximately 20-30 hours on this assignment. Plan accordingly.

To complete this assignment, you should:

1. Accept this assignment through GitHub Classroom.
2. Modify the files listed below, implementing the functions labeled `todo!()`. Note that you may need to add additional fields to some structures.
    - `src/data_ops/nested_loop.rs`,
    - `src/data_ops/block_nested_loop.rs`,
    - `src/data_ops/on_disk_hash.rs`,
3. Commit your changes and push them to Github.
4. Go to Autolab, select your repository, acknowledge the course AI Policy, and click Submit.

You may repeat steps 2-4 as many times as desired. You may also modify any of the files in the `data_ops` module.

Overview

In this assignment you will implement three on-disk join algorithms:

- Nested Loop Join
- Block Nested Loop Join

- On-Disk Hash Join

`data_ops::Source<T>`

This the `Source` struct represents an abstract collection of records of type `T`. A `Source<T>` defines a single method: `iter()` that returns a fresh `Iterator<Item = T>` over the collection. Examples of `Source` can be found in:

- `test_utils::RangeSource`
- `tpch::CustomerFile`
- `tpch::OrdersFile`
- `data_ops::DataFile`

Join Algorithms

All of the join algorithms take a `Source<A>` and a `Source<B>` as input.

The join algorithms are composed of two parts:

- A `Source<(A, B)>` that can be used to create...
- An `Iterator<Item = (A, B)>` that actually 'materializes' the individual records

You will need to implement the `Iterator` for all three join algorithms.

Documentation

You may find the following documentation useful:

- The Rust Book
- std::fs::File
- Run `cargo doc --open`

The following utility classes are provided:

`data_ops::DataFile<T>`

A DataFile stores a collection of records using the compact Postcard serialization format, using the Rust `serde` library.

See the file `src/data_ops/data_file.rs` for documentation; In summary, `DataFile` can be used to create temporary files to store data on-disk as-needed.

For example:

```
let a = RangeSource(0..100);
let data_file = DataFile::temporary()?
                    .from_source(a)
                    .build();
```

or

```
let mut builder::DataFileBuilder<u32> = DataFile::temporary()?;
builder.write(1);
builder.write(2);
builder.write(3);
builder.write(4);
let data_file = builder.build()
```

Serializing new types of data

The `serde` library requires that the records being stored implement the `Serialize` and `Deserialize` traits. The `serde` library provides a macro that can be used to automatically derive these traits. See `tpch::Customer` and `tpch::Orders` for examples.

`data_ops::InMemHashJoin<A, B, GetKeyA, GetKeyB>`

This class implements an in-memory hash algorithm. It may be useful in implementing the On-Disk Hash Join algorithm. It follows the same general template as the other three join algorithms, of defining both a source and an iterator class.

Objectives

In this assignment, you will implement three join algorithms

`src/data_ops/nested_loop.rs`

This file defines two structs: `NestedLoopJoin` and `NestedLoopJoinIterator`.

Enumerating the entire iterator produced by `NestedLoopJoin` should have:

- $O(1)$ memory
- $O(|source\_A| * |source\_B|)$ IO
- $O(|source\_A| * |source\_B|)$ runtime

You will need to implement the method `NestedLoopJoinIterator::read_one`, which retrieves the next element from the cartesian product of the two input sources.

Recall that the nested loop join works by computing the cartesian product of the two sources as follows:

```
for a in source_a.iter()?
{
  for b in source_b.iter()?
  {
    emit (a, b);
  }
}
```

Note the memory requirement. You should **not** construct the entire result all at once. Instead, you'll need to implement a `read_one` method that constructs only the very next record.

A typical approach involves maintaining two iterators as state: One over the elements of `source_a`, and one over the elements of `source_b`.

- Advance the `b` iterator by one element.
- If there are no more elements, reset the iterator (generate a new one) and advance the `a` iterator by one step.

You will need to make changes beyond just the one function you're implementing; for example, you will likely need to add fields to `NestedLoopJoinIterator`

Two test cases are provided:

- nested_loop_simple
- nested_loop_2way_join (note: This test is *waaaay* too slow to generate the full result; it is configured to only generate the first 10 results)

`src/data_ops/block_nested_loop.rs`

This file defines two structs: `BlockNestedLoopJoin` and `BlockNestedLoopJoinIterator`.

Enumerating the entire iterator produced by `NestedLoopJoin` should have:

- $O(block\_size)$ memory
- $O(|source\_A| * |source\_B|/block\_size)$ IO
- $O(|source\_A| * |source\_B|)$ runtime

You will need to implement the method `NestedLoopJoinIterator::next()`, which retrieves the next element from the **join** of the two input sources.

Recall that the block nested loop join works by computing the cartesian product of the two sources as follows:

```
let mut iter_a = source_a.iter()?;
let mut buffer_a = /* read buffer_size elements from iter_a */
while buffer_a.len() > 0
{
  for b in source_b.iter()?
  {
    for a in buffer_a
    {
      emit (a, b)
    }
  }
  buffer_a = /* read buffer_size elements from iter_a */
}
```

As before, you will need to refactor the code above into an iterator.

Note the memory requirement. You should **not** construct the entire result all at once.

Additionally, note that, unlike the regular Nested Loop Join, the missing method here is `Iterator::next`. This means you will need to emit only records that pass the provided `test(a, b)`. See `NestedLoopJoinIterator` for ideas on how to do this.

You will need to make changes beyond just the one function you're implementing; for example, you will likely need to add fields to `BlockNestedLoopJoinIterator`

Two test cases are provided:

- block_nested_loop_simple
- block_nested_loop_2way_join (note: This test is a bit too slow to generate the full result; it is configured to only generate the first 100 results)

`src/data_ops/on_disk_hash.rs`

This file defines two structs: `OnDiskHashJoin` and `OnDiskHashJoinIterator`.

Enumerating the entire iterator produced by `NestedLoopJoin` should have:

- Expected $O(|source\_A|/partitions + |source\_B|/partitions)$ memory
- $O(|source\_A| + |source\_B|)$ IO
- Expected $O(|source\_A| + |source\_B|)$ runtime

You will need to implement the body of the methods `OnDiskHashJoin::new()` and `OnDiskHashJoinIterator::next()`.

Recall that the on-disk hash join works by building a series of hash-based partitions.

```
let data_files:Vec<(_, _)> = /* Create 2 data files for each partition */

//////// Handle this part in OnDiskHashJoin::new() ///////////
```

```
for a in source_a.iter()?
{
  /* append a to */ data_files[hash(get_a_key(a))]
}
for b in source_b.iter()?
{
  /* append b to */ data_files[hash(get_b_key(b))]
}

//////// Handle this part in OnDiskHashJoinIterator::next() ///////////

for i in 0 .. partitions
{
  /* do an in-memory hash join on data_files[i].0 and data_files[i].1 */
}
```

Note that this algorithm works in two phases:

1. Loading data into partitions on disk. Do this phase in `OnDiskHashJoin::new()`
2. Performing a hash join over each successive partition. Do this phase in `OnDiskHashJoinIterator::next()`

Note the memory requirement. You should **not** construct the entire result all at once.

Also note the presence of two other classes that should help you: - You can use `data_ops::DataFile::temporary()` (see docs above) to create temporary files that you can store each partition in. - You can use `data_ops::InMemHashJoin` **instead of implementing an in-memory hash join yourself**

Two test cases are provided:

- on_disk_hash_simple
- on_disk_hash_2way_join (note that this test can take a minute or three, depending on how fast your disk is)

Strategy

1. Implement the regular nested loop join
2. Implement the block nested loop join
3. Implement the on-disk hash join

Additional Notes

- You may **not** add new crates without permission.