W3: Join Algorithm Performance

- **Due:** April 28; 11:59 PM
- **Summary:** 4 parts, for a total of 12 points.
- **Submission:** https://autolab.cse.buffalo.edu/courses/cse410-s24/assessments/W3-Join-Algorithm-Performance

Submission

Only PDF-formatted files will be accepted by autolab.

- You may typeset your submission in LaTeX, Typst, or a similar tool.
- You may use a word processor to prepare your submission, but you must export the document to PDF before submitting.

Note that the instructor must be able to read your answer. Submissions that are unintelligible will receive no points.

Overview

Database systems frequently implement a range of different join algorithms to cope with different situations. In this assignment, you will

Documentation

Your documentation should include:

- An overview of your experimental setup.
- A set of hypotheses for when each join algorithm is preferable.
- Experiments supporting (or rejecting) your initial hypothesis.

Part 1: Hypothesis Generation [3pt; 1 task]

You implemented three join algorithms (and the assignment came packaged with a fourth):

1. Nested Loop Join
2. Block Nested Loop Join (parameter: Block Size)
3. On-Disk (2-pass) Hash Join (parameter: Number of Partitions)
4. In-Memory (1-pass) Hash Join

Task: Review these algorithms and develop a set of hypothesis, for each algorithm, stating under what conditions the algorithm is the 'best' (and why). For those algorithms that include parameters, state how the parameter is to be selected.

In addressing these questions, you may wish to consider properties including:

- The algorithm's runtime complexity.
- The algorithm's memory complexity.
- The algorithm's IO complexity.
- The algorithm's capabilities (e.g., the join condition)
- Features of the data (e.g., size, number of rows)
- Features of the computer (e.g., memory, disk type)

Part 2: Experimental Setup [3pt; 1 task]

Experimental outcomes can depend heavily on the environment in which the experiment is performed. For example, AMD and Intel CPUs adopt very different memory management models. Specific CPU models (e.g., the Intel P5) have hardware glitches that can break experiments. Different operating systems adopt different disk caching and virtual memory policies.

All of this falls under the heading of "Reproducibility." To ensure that others can re-create your experiments, and that inconsistencies can be properly attributed, you need to properly document the context(s) in which you conducted the experiment.

Task: Your write-up should include a roughly 2 paragraph summary of the experimental setup, including, in particular:

- Details of the hardware you're running the experiment on
  - CPU model (Intel, AMD, ARM; What specific CPU model)
    - On linux, you can get this information by running `cat /proc/cpuinfo` (the `model name` field)
  - RAM (Amount)
    - On linux, you can get the total memory information by running `cat /proc/meminfo`
  - Disk (eMMC, SSD, or HDD)
- Details of the software you're using
  - Operating System (Windows, Linux, MacOS, BeOS; Version/Kernel Revision)
    - On Linux, you can get this information by running `cat /proc/version` (The kernel version will be at the front. e.g., `Linux version 6.6.10-76060610-generic`)
  - Compiler Version
    - On most platforms you can get this information by running `rustc --version`
  - Scripting tools used to automate the testing process
- Details of general experimental procedure
  - Did you test Hot or Cold performance? (see "Hot vs Cold Systems" below)
  - How did you account for mechanical differences between the systems?
    - e.g., Block nested loop join supports arbitrary tests between tuples, while hash join supports only equality tests between tuples.
- Details of where to find your experimental code
  - You may use the P3 github repository for this; Provide a link.

Part 3: Experimental Design [4pt; 2 tasks]

Design one or more experiments to assess whether each of your hypotheses from part 1 are correct.

A good experimental design considers a range of variables. For analyzing a join between tables R and S, these can include (but are not limited to):

1. The number of records in R and S
2. The size of each record in R and S
3. The join condition between R and S.
4. How many records in S (resp. R) does each record in R (r. S) join with? (this is known as the 'Fan-out' of the join)
5. Is there a skew in the number of records that any given record in R (resp. S) joins with (e.g., some records join with a large number of other records)
6. The runtime of the algorithm
7. The memory use of the algorithm

Task 1 For each experiment you plan to conduct, list all variables you plan to consider, and for each identify whether the variable will be:

- A **Control Variable** (held constant); Indicate what value you will be holding the variable constant at, and how you plan to do so.
- A **Independent Variable** (varied); Indicate the range of values you will be exploring.
- A **Dependent Variable** (measured); Indicate how you will measure the variable.

Task 2: For each hypothesis from part 1, state what effects you would expect to see in the experimental results if your hypothesis is correct.

See “Experimental Design,” below for further guidance.

Part 4: Performance Evaluation [2pt; 1 task]

Task: Conduct the experiments you developed in Part 3 and provide the results as a set of plots and **brief** summary text.

Experimental Design

Measuring Runtime

Good experimental practice is to isolate the component being evaluated to the greatest extent possible. For example, if you use a python script to test runtime by running the `binary_search` or `bplus_tree` binaries directly, your measurements will include overheads from launching the binary (which is often far more expensive than just doing a single lookup).

Advice: For this project, to measure time directly in Rust. For example you can use the Rust `std::time::Instant` struct, and in particular `Instant::now()` to measure the time:

```
use std::time::Instant

let start = Instant::now()
run_your_test()
let end = Instant::now()
let runtime = (end - start).as_secs_f32()
```

Measuring Short Times

Most time measurements are at least a little noisy. When the thing you’re measuring is sufficiently fast, the noise may dominate the runtime you measure. Worse, you may end up with mostly zero runtimes, as the timer’s increment is coarser than the experiment.

Advice: If the time you’re measuring is short, consider running a large number of trials of the experiment (e.g., so that all of the trials together take at least 1s). Measure the **total** time to run all the trials, and then divide by the number of trials to get the average time per trial.

Error Bars

Experimental results can be noisy: Perhaps you’re watching a video while you run your experiments (don’t do this, btw); Perhaps you’re unlucky and the system picks the exact instant that you run your test to do some system maintenance task. Alternatively, perhaps your data structure has random behavior that can significantly affect its performance on a case-by-case basis.

In addition to reporting the measured runtime, you should also report how ‘variable’ your measurements are, typically by performing the experiment multiple times, and then reporting the mean and variance of the set of experiments. **Note**: This is in on top of running the experiment multiple times if you’re trying to measure a short time.

Advice: Run the experiment multiple times (5 or 10 is my usual rule of thumb) and report the mean and variance of the trials.

Generating Data/Workloads

The cornerstone of a good evaluation is a carefully thought-out workload. This includes both the base data that you use to populate your data structures, as well as the workloads (i.e., which keys do you look up)?

Advice: Think through a few questions when designing your experiment:

- How big should the dataset(s) you use be?

- How are you generating the test workload (e.g., which keys are you looking up)?
- Are you looking exclusively for keys that are present, or are you looking for any key?
  - If the latter, what is the expected ratio of keys that are/are not present.

Hot vs Cold Systems

A common source of noise in experimental evaluations comes from caches in the system, where the system itself tries to optimize away (using caches) the thing that you're testing. For example, most operating systems and language runtimes will cache disk accesses. This is great for users, but makes it hard to properly attribute performance improvements. For example, the same test may run substantially faster just by running it a second time.

As a result, typically evaluations are performed after resetting as much of the system as possible (termed a 'cold cache' experiment), or by only starting measurement after the system has already been running for a period (termed a 'hot cache' experiment).

You'll never get a fully cold cache, but you can often get a reasonable approximation by restarting the binary. e.g., you can assume that every time you call `cargo test` or `cargo run`, the cache will be cleared (flushed).

To get a hot cache, it's common to set aside a portion of the workload (the specific amount varies) as a 'burn-in' workload; start measurement after the burn-in workload completes.

Advice: In the write-up, clearly indicate whether you are conducting hot- or cold-cache experiments (or both). For cold-cache experiments, clearly indicate the steps you take to reset the cache. For hot-cache experiments, clearly indicate the steps you take to warm up the cache.