Amery Chang

ac2925

PFP COMS-4995-02, Fall ‘22

Project Proposal: Parallel Database Joins

Abstract

I’m interested in implementing the textbook database join algorithms in Haskell, namely *nested-loop join*, *sort-merge join*, and *hash join*, and finding opportunities to perform parallelism. While the task of “join some data together on some column(s)” may seem fairly simple, it involves searching, filtering, and sorting as intermediary steps, and becomes more challenging when joins occur on multiple attributes. Moreover, I intend for the bulk of the complexity to come from structuring code into functions and combinators that can take advantage of Haskell’s parallelism constructs. My objective is to observe noticeable performance improvement when running the joins in parallel compared to in serial.

Implementation and Testing

My program will read input data (most likely several files representing normalized relational database tables) into Haskell data structures and perform the join algorithms. My data will be entirely in memory. While query optimization typically takes access paths (i.e. indexes) into account, I will not be considering the existence of indexed columns, since none of my data is on disk.

By “join” in this project, I am referring to the relational algebra operations for *conditional (theta) join* and *equijoin*. Formally,

$$R \bowtie_{\theta} S = \sigma_{(\theta)(R \times S)}$$

for relations R and S, and condition θ. The condition may be some predicate (theta join), or the condition can simply be an equal sign (equijoin). In SQL, joining on a predicate is essentially the inner join so my project will basically produce tuples that are the result of an inner join.

In database implementation, one set of techniques in query optimization is applying equivalence rules to relational algebra operators in order to create more efficient query plans (for example, by reducing the number of rows that need to be scanned). The focus of my project is not creating an optimal query plan, so I won't be doing much of this. Since my goal is to test the raw performance improvement of joins via parallelism, the actual efficiency of the query isn't something I'll be trying to optimize.

I plan to create several baseline queries of varying complexity (join conditions, joining on multiple columns, performing various filters and sorts). For each, I will perform my join algorithms without parallelism, and with parallelism, and compare the results. Additionally, I will highlight implementation details of the parallel approaches which improve performance.

Questions

I'd really appreciate feedback on a few things:

1. Is this problem "Haskell-y"? Does it make sense to do this in Haskell, and (the thing I really hope to be the case)- are there specific advantages to doing this type of work in Haskell?
2. Is this a good fit with Haskell's flavor of parallelism? I don't think this is too far removed from some of the parallelism examples we've studied, and for that matter we did some join-type stuff in homework 5.
3. Is this an appropriate level of complexity? I'm planning to work on my own. Is this a case where you'd expect to see a huge input data set? If so, I'd appreciate that feedback as soon as possible so that I can find or create a large enough data set.