ADBIS: Project 2

Fabian Krause

July 31, 2022

Contents

1 Problem statement

The problem we face is to implement and analyze the performance of the hash join and sort-merge join algorithms based on a query to be performed on two datasets; we are also attempting to find a join algorithm with better performance.

The query is as follows:

$$\begin{aligned}\texttt{follows} \bowtie_{follows.object=friendOf.subject} & \texttt{ friendOf} \\ \bowtie_{friendOf.object=likes.subject} & \texttt{ likes} \\ \bowtie_{likes.object=hasReview.subject} & \texttt{ hasReview}\end{aligned}$$

2 Algorithm description

In the following, we are talking about a join $A \bowtie_{A.columnA=B.columnB} B$.

2.1 Hash Join

Hash join works by creating a hash table h that maps values of $A.columnA$ to their respective rows in A (building). After creating this map, we only have to iterate once through B and for each row r create the Cartesian product $h[r.columnB] \times r$ (probing).

If A is too big to fit in memory, we divide the process into several phases of building and probing: We iterate through A, adding elements to the hash table until the memory limit is reached. We then probe B as before. Now, we clear the hash table, and continue iterating through A, adding elements to the hash table, probing B again when the hash table is full again. We repeat this until we iterated through all of A.

2.2 Sort-Merge Join

Sort-merge join performs two steps: Sorting A and B, and then merging the results. When A and B are sorted, joining is trivial. We create two pointers r_A and r_B pointing at the beginning of both tables, and advance the pointers depending on the values of $columnA$ and $columnB$:

- If $r_A.columnA < r_B.columnB$, advance r_A.
- If $r_A.columnA > r_B.columnB$, advance r_B.
- If $r_A.columnA = r_B.columnB$:
    1. Advance r_A until $columnA$ differs from $r_A.columnA$, cache all rows in a set $A_{columnA}$.
    2. Advance r_B until $columnB$ differs from $r_B.columnA$, cache all rows in a set $B_{columnB}$.

3. Add $A_{columnA} \times B_{columnB}$ to result.

The difficult part of this algorithm is sorting tables that do not fit into memory. We need to use an external sorting algorithm. In this report we use k-way merge sort. Let us assume we want to sort A on column *columnA*.

1. Iterate through A, repeating:
   a) Fill up memory with rows, sort rows in memory, write sorted result to file, clear memory.
2. At the end of the previous process, we will have k separate files.
3. Until there is only one file left, do the following:
   a) Split files into pairs, merge pairs leaving $k/2$ files for the next iteration. Merging is done as follows:
      i. Create two pointers r_1 and r_2 pointing at the beginning of both intermediate results.
      ii. Do the following until the end of both documents is reached:
         - If $r_1.columnA < r_2.columnA$, add r_1 to result, advance pointer r_1.
         - If $r_1.columnA > r_2.columnA$, add r_2 to result, advance pointer r_2.
         - If $r_1.columnA = r_2.columnA$, add r_1 and r_2 to the result, advance both pointers.
4. The remaining file is the sorted result.

2.3 Parallel Sort-Merge Join

Parallel sort-merge join is a modification of sort-merge join. The merge part of the algorithm stays the same, as parallelization cannot overcome the bottleneck of disk I/O, but the sorting part can be improved. This applies to both the sorting done to whole tables that fit into memory and sorting done to chunks of tables in case the data does not completely fit into memory.

The idea is to use l-way merge sort where l is the available number of threads. The algorithm works as follows:

1. Read whole table into memory or fill up memory with chunk
2. Split data into l separate chunks of the same size
3. Sort chunks in parallel in l separate threads
4. Merge pairs in parallel until all chunks are merged
5. Write result to file

3 Dataset description

The dataset was provided in the RDF format and had to be partitioned into relational databases so that a join on them can be performed. Each property corresponds to a table.

Since the data contains strings, and comparison of strings is slow (and the results of the query take up more than 100 GB of space when using strings for `watdiv10M`), values are replaced with integers. A database `legend.csv` is created, which maps each integer to its original string value.

The data was generated from the Waterloo SPARQL Diversity Test Suite and describes a social network (e.g. `follows`, `friendOf`) which includes the ability to like and review products (e.g. `likes`, `hasReview`).

Two separate instances of the dataset were provided, `watdiv100k` and `watdiv10M`. The former is only around 5MB in plain text, where as the latter is about 1.5GB in plain text. The query results in about 11.41 million rows for the smaller dataset, and about 1.03 billion rows for the bigger dataset.

4 Experiment and Analysis

4.1 Benchmark Setup

Two systems where used to measure the performance of the join algorithms:

CPU	AMD Ryzen 5800U, 1.9-4.4 GHz, 8 Cores, 16 Threads
RAM	14 GB DDR4 @ 3200 MHz
Storage	M.2 NVMe SSD

Table 1: System 1.

CPU	Intel i7-4790, 3.6-4.0 GHz, 4 Cores, 8 Threads
RAM	16 GB DDR3 @ 1600 MHz
Storage	SATA SSD

Table 2: System 2.

The query was executed 50 times on `watdiv100k` and 5 times on `watdiv10M` for each join algorithm and system. Real execution time was measured using `std::chrono`. Memory was limited to 8 GB. The algorithms were implemented in C++ using the C++20 standard. The implementation can be found at https://github.com/kcaliban/adbis-project2.

4.2 Hash Join

4.2.1 Time Complexity

Let n be the number of rows in A, m be the number of rows in B.

Creating the hash table requires $\mathcal{O}(n)$ time, as we simply iterate through A and hash table insertions are on average constant.

Let c_A be the maximum amount of rows in A with the same value in *columnA*. Probing takes $\mathcal{O}(m \cdot c_A)$ time: While iterating through B, hash table lookups are on average constant, and creating the Cartesian product may take c_A operations for each iteration. This leaves us with $\mathcal{O}(n + m \cdot c_A)$.

In the worst case, $c_A = n$ and we have $\mathcal{O}(n \cdot m)$. Usually, c_A is small enough to assume it is constant, hence we can assume an overall complexity of

$$\mathcal{O}(n + m)$$

If the hash table is too big to fit into memory, we have to iterate through B more than once. Let $k = \lceil \frac{\texttt{size\_of}(A)}{\texttt{mem}} \rceil$.

We still only have to iterate through A once, but through B each time we create a new hash table. The complexity becomes $\mathcal{O}(n + k \cdot m \cdot c_A)$, or

$$\mathcal{O}(n + k \cdot m)$$

for small c_A.

4.2.2 Benchmarks

For `watdiv100k`, the following real execution times were recorded:

	System 1	System 2
Avg.	14.055s	26.093s
Max.	14.235s	26.827s
Min.	13.928s	25.945s

Table 3: Average, maximum and minimum real execution times of hash join on `watdiv100k` on system 1 and 2.

For `watdiv10M`, the following real execution times were recorded:

	System 1	System 2
Avg.	23.21min	42.85min
Max.	23.25min	42.93min
Min.	23.16min	42.75min

Table 4: Average, maximum and minimum real execution times of hash join on `watdiv10M` on system 1 and 2.

4.3 Sort-Merge Join

4.3.1 Time Complexity

Let n be the number of rows in A, m be the number of rows in B.

Sorting both tables takes $\mathcal{O}(n \cdot \log n + m \cdot \log m)$. Merging the results requires iterating through (sorted) A and (sorted) B, so $\mathcal{O}(n + m)$.

Let c_A be the maximum amount of rows in A with the same value in *columnA*, let c_B be the analog for B and *columnB*. Taking into account the computation of Cartesian products, we arrive at a full complexity of $\mathcal{O}(n \cdot \log n + m \cdot \log m + n + m + \max(n, m) \cdot c_A \cdot c_B)$. We can assume again that c_A and c_B are constant, and arrive at

$$\mathcal{O}(n \cdot \log n + m \cdot \log m)$$

If the tables are too big to sort in memory, we have to use external sorting. Let us show that k-way merge sort also has complexity $\mathcal{O}(n \cdot \log n)$, therefore not changing overall complexity:

Let $k = \lceil \frac{\texttt{size\_of}(A)}{\texttt{mem}} \rceil$. We will sort k separate chunks of data in memory, each requiring $\mathcal{O}\left(\frac{n}{k} \cdot \log \frac{n}{k}\right)$. The time complexity is therefore

$$\mathcal{O}\left(k \cdot \frac{n}{k} \cdot \log \frac{n}{k}\right) = \mathcal{O}\left(n \cdot \log \frac{n}{k}\right)$$

Merging requires $\mathcal{O}(\log k)$ iterations, as each iteration halves the number of pairs. In each iteration, we traverse the data exactly once, since all chunks are traversed exactly once. Merging therefore has complexity

$$\mathcal{O}(n \cdot \log k)$$

For the overall external k-way merge sort, we get the complexity:

$$\begin{aligned}\mathcal{O}\left(n \cdot \log \frac{n}{k} + n \cdot \log k\right) &= \mathcal{O}\left(n \cdot \left(\log \frac{n}{k} + \log k\right)\right) \\ &= \mathcal{O}(n \cdot (\log n - \log k + \log k)) \\ &= \mathcal{O}(n \cdot \log n)\end{aligned}$$

4.3.2 Benchmarks

For `watdiv100k`, the following real execution times were recorded:

	System 1	System 2
Avg.	15.701s	28.129s
Max.	16.000s	29.829s
Min.	15.566s	27.946s

Table 5: Average, maximum and minimum real execution times of sort-merge join on `watdiv100k` on system 1 and 2.

For `watdiv10M`, the following real execution times were recorded:

	System 1	System 2
Avg.	29.16min	51.54min
Max.	29.3min	51.7min
Min.	29.02min	51.47min

Table 6: Average, maximum and minimum real execution times of sort-merge join on `watdiv10M` on system 1 and 2.

4.4 Parallel Sort-Merge Join

4.4.1 Time Complexity

As explained in Section 2.3, only the sorting part of the algorithm is changed.

Let l be the number of available threads. For tables that fit in memory, we perform parallel l-way merge sort for the tables in memory. In the previous chapter we have shown that the sorting step of non-parallel l-way merge sort has complexity $\mathcal{O}\left(l \cdot \frac{n}{l} \cdot \log \frac{n}{l}\right)$. Since we sort in l threads in parallel, this complexity becomes

$$\mathcal{O}\left(\frac{n}{l} \cdot \log \frac{n}{l}\right)$$

The merge part of parallel l-way merge sort is also done in parallel, leaving an overall complexity for sorting of

$$\mathcal{O}\left(\frac{n}{l} \cdot \log \frac{n}{l} + \frac{n}{l} \cdot \log l\right) = \mathcal{O}\left(\frac{n}{l} \cdot \log n\right)$$

And therefore an overall complexity of parallel sort-merge join for tables that fit in memory of

$$\mathcal{O}\left(\frac{1}{l}\left(n \cdot \log n + m \cdot \log m\right) + (n + m)\right)$$

For tables that do not fit in memory, the analysis gets a bit more involved. Let $k = \lceil\frac{\texttt{size\_of}(A)}{\texttt{mem}}\rceil$. The complexity of sorting for non-parallel sort-merge join for data that does not fit in memory was

$$\mathcal{O}\left(k \cdot \frac{n}{k} \cdot \log \frac{n}{k}\right)$$

The difference for parallel sort-merge join is that the complexity of sorting of each chunk changes from $\mathcal{O}\left(\frac{n}{k} \cdot \log \frac{n}{k}\right)$ to $\mathcal{O}\left(\frac{n}{kl} \cdot \log \frac{n}{k}\right)$, as we can derive from the case where table data fits into memory. The complexity of the sorting step of parallel sort-merge join for data that does not fit into memory is then

$$\mathcal{O}\left(k \cdot \left(\frac{n}{kl} \cdot \log \frac{n}{k}\right) + k' \cdot \left(\frac{m}{k'l} \cdot \log \frac{m}{k'}\right)\right) = \mathcal{O}\left(\frac{n}{l} \cdot \log \frac{n}{k} + \frac{m}{l} \cdot \log \frac{m}{k'}\right)$$

where $k' = \lceil\frac{\texttt{size\_of}(B)}{\texttt{mem}}\rceil$. Combining this result with the complexity of the merging step yields the following overall complexity

$$\mathcal{O}\left(\frac{n}{l} \cdot \log \frac{n}{k} + \frac{m}{l} \cdot \log \frac{m}{k'} + n + m\right)$$

4.4.2 Benchmarks

For `watdiv100k`, the following real execution times were recorded:

	System 1	System 2
Avg.	15.730s	28.199s
Max.	16.115s	28.981s
Min.	15.612s	28.034s

Table 7: Average, maximum and minimum real execution times of parallel sort-merge join on `watdiv100k` on system 1 and 2.

For `watdiv10M`, the following real execution times were recorded:

	System 1	System 2
Avg.	27.96min	49.36min
Max.	28.02min	49.93min
Min.	27.92min	48.97min

Table 8: Average, maximum and minimum real execution times of parallel sort-merge join on `watdiv10M` on system 1 and 2.

4.5 Analysis

The following table shows average execution times for each join algorithm, dataset and system.

	Hash Join	Sort-Merge Join	Parallel Sort-Merge Join
`watdiv100k` @ Sys. 1	14.055s	15.701s	15.730s
`watdiv10M` @ Sys. 1	23.21min	29.16min	27.96min
`watdiv100k` @ Sys. 2	26.093s	28.129s	28.199s
`watdiv10M` @ Sys. 2	42.85min	51.54min	49.36min

Table 9: Average real execution time of hash join, sort-merge join and parallel sort-merge join on `watdiv10M` and `watdiv100k` on system 1 and 2.

Relations and intermediate query results on `watdiv100k` fit completely into memory. Hash join has the best performance in that case. It is 12% faster than sort-merge join on system 1, and 7% faster than sort-merge join on system 2. As our complexity analysis has shown, it has complexity $\mathcal{O}(n+m)$, which is as good as it can get (each table has to be iterated at least once).

The performances of sort-merge join and parallel sort-merge join are similar to another when data fits into memory. As the multithreading overhead of sorting such a small amount of data in parallel cancels out its performance gain, the implementation of parallel sort-merge join uses only a very small amount of threads. The slight difference ($<1\%$) in performance may be attributed to spinning up these threads.

For the dataset `watdiv10M`, where data does not fit into memory, hash join still outperforms the other algorithms: It is 25% and 20% faster than sort-merge join on system 1 and 2 respectively, 20% and 15% faster than parallel sort-merge join on system 1 and 2 respectively. This is not immediately apparent from our complexity analysis.

As we have shown, the time complexity of hash join and sort-merge join when data does not fit into memory is $\mathcal{O}(n+k\cdot m)$ and $\mathcal{O}(n\cdot\log n+m\cdot\log m)$, respectively. The first summand is smaller for hash join. For the second summand, it matters whether $k=\lceil\frac{\texttt{size\_of}(A)}{mem}\rceil$ is less than or greater than $\log m$, which depends on how much memory is available, the size of A, and the number of rows in B $(=m)$. As an example, if $m=10000$, $\texttt{size\_of}(A)$ must be $\log_2 m\approx 13.29$ bigger than the available memory. If the available memory is 8GB, then A must take 106GB of space for the second summand to be equal.

The key difference between hash join and both sort-merge algorithms when data does not fit into memory is that the latter two have to repeatedly read from and write data to disk (or in this case, flash storage). Hence, even though sort-merge join has the same theoretical complexity when spilling to disk as when being able to keep all data in memory, the relative speedup of hash join with regard to sort-merge join is higher for `watdiv10M` than for `watdiv100k` (12% and 25% respectively on system 1).

Sorting in parallel when there is enough data slightly improves sort-merge join. The speedup is about 4.2% on system 1 and 4.4% on system 2. The central bottleneck of any sort-merge join algorithm is the sorting itself, plus disk I/O when sorting externally.

5 Conclusion

The time complexity analysis has shown that hash join performs better than sort-merge join and parallel sort-merge join when data fits into memory, and the benchmarks have confirmed this. The improved sort-merge join algorithm employing a parallel sorting scheme does not improve performance for small amounts of data.

When data does not fit into memory, it is not directly apparent from the complexity analysis which algorithm is fastest. The benchmarks have shown that hash join is faster than sort-merge join in this case as well, and that parallel sorting improves sort-merge join by about 4%, still not reaching the performance of hash join. The main bottleneck of any sort-merge join algorithm is having to sort the data, and disk I/O when sorting externally. Even though SSDs are getting faster, they are still not at the level of the speed of RAM.

List of Tables