CS3223 Report

Kerryn Eer (A0149960U)
Manickamalar Jothinandan Pillay (A0168954L)
Qi Ji (A0167793L)

9th April 2020

Contents

1 Introduction

For the project we implemented the following:

- Block Nested Loops Join
- Sort Merge Join (with external sort algorithm)
- `DISTINCT` operator (implemented with external sort)
- Aggregate functions (`MIN`, `MAX`, `COUNT`, `SUM`, `AVG`)
- Replaced the optimizer with another randomized algorithm [JS13]
- Addressed some miscellaneous limitations

2 Implementation Details

2.1 Block Nested Join

2.1.1 Overview

The Block Nested Join is a similar to the Page Nested Loop Join with the main difference being that multiple pages of the outer (left) table are loaded into buffers at once and compared with a page from the inner (right) table each time.

2.1.2 Algorithm

The algorithm can be summarized as follows:

1. Read as many outer (left) table pages as possible into $B-2$ buffers. If lesser than $B-2$ buffers filled, take note of how many buffers are actually filled.
2. Read one page of the inner (right) table into 1 buffer. Every time new outer pages are loaded into the $B-2$ buffers. The reading of pages from the inner file are reset. If the same set of outer pages are being scanned, then just read the next available page of the inner table.
3. Loop through each of the $B-2$ buffers, for each page from the outer table, loop through each tuple in the 1 page buffer holding tuples from inner table. If there is a match on the join condition, the 2 tuples are join and the new tuple is added to the output buffer. Essentially, for each page in the $B-2$ buffers, the inner table page is re-scanned to find tuples satisfying the join condition.
4. Once the entirety of the inner page has been scanned with all the pages in the $B-2$ buffers, a new inner page is loaded and the process is repeated until all inner pages have been loaded and scanned.

5. Once all inner pages have been scanned for a particular set of outer pages, a new set of outer pages (up to $B-2$) are loaded into buffers and the entire process is repeated. When a new set of outer pages are in buffers, the inner pages are read freshly from scratch. This entire process is repeated until all outer pages have been loaded into buffers and scanned.
6. The process terminates when there are no pages left to be loaded into the buffers. Essentially, the `ArrayList` holding the pages from the outer table is either `NULL` or of size 0.

2.1.3 Manipulation of Cursors while Scanning Tuples

If the output buffer is full after adding a tuple, a new output buffer needs to be initialized before we continue scanning. However to ensure that no tuples are duplicated or missed out, the pointers need to be carefully manipulated. There are a total of 3 pointers. The `bufferCursor`, which controls the looping of the B-2 buffers. The `outCursor` which loops through the tuples in each outer page and the `inCursor`, which loops through the tuples in each inner page.

- **Case 1.** If both the `outCursor` and `inCursor` are at the end of the their respective pages, it implies that the scanning of an outer page and that inner page is complete. Thus, the `bufferCursor` is incremented by 1 to retrieve the next page from the $B-2$ buffers (if available) and the `outCursor` and `inCursor` is reset to 0 to essentially start scanning a fresh outer page from the buffer.
- **Case 2.** If the `outCursor` is not at the end of the page but the `inCursor` is at the end of the page, this means that for the current tuple in the outer page, all the inner tuples have been scanned. So we simply maintain the `bufferCursor`, while incrementing the `outCursor` and resetting the `inCursor` to 0.
- **Case 3.** If the `outCursor` is at the end of the page but the `inCursor` is not, it means that the inner page has not been fully scanned. Thus, for this case, both the `bufferCursor` and `outCursor` are maintained while the `inCursor` is incremented by 1.
- **Case 4.** If both the `outCursor` and `inCursor` are not at the end of the page, it is the same as Case 3 as it implies that for the current tuple in the outer page, the inner page has not been fully scanned. Thus, similarly, both the `bufferCursor` and `outCursor` are maintained while the `inCursor` is incremented by 1.

2.2 Sort Merge Join

There are 2 parts to this algorithm.

1. External Sort - to sort the 2 tables
2. Perform merge - to join the tuples with the same index from the 2 tables

2.2.1 External Sort

It aims to sorts the given relation based on the index given. This index is specified by sort merge join condition.

There are 2 phases in the External Sort algorithm. Phase 1 to generate the sorted runs and phase 2 to merge the sorted runs.

For **phase 1**, `generateSortedRuns()` will:

1. Use all buffer pages available to read in as many pages as it can into memory.
2. Sorts the tuples in memory using in memory sorting algorithm
3. Write these tuples out into a file, forming 1 sorted run
4. Repeat this till all pages of the relation has been read, creating multiple sorted runs

For **phase 2**, perform the merge on these multiple sorted runs from phase 1. `performMerge()` will keep 1 page for writing out the sorted tuple and use the remaining available buffer pages to merge the sorted runs. It calls `mergeSortedRuns()` to perform the merge given the specified sorted runs, which returns 1 resultant sorted run. This resultant sorted run will be added to the remaining list of sorted runs to merge. This continues till we are left with only 1 sorted run (the sorted file).

The implementation of `mergeSortedRuns()` is similar to a k-way merge sort given k sorted runs. It looks for the smallest tuple from the beginning of the k sorted runs and add it to the output file. The pointer to the tuple in the sorted run which the smallest tuple was outputted is incremented. This process repeats till we have outputted all tuples from all k sorted runs.

2.2.2 Perform the merge

We have left and right pointers that traverse both the sorted left and sorted right tables at the same time. A list `matchingTuples` is used to store all the tuples of the right table with the same value as the current right tuple. It is to be used to be compared with the current left tuple.

Here is how we advance the left and right pointers of both tables:

- If the current left tuple has the same value as the right tuple, we can join the current left tuple with all the right tuples in the `matchingTuples` (which will all have the same value as the left tuple). We then increment the left pointer and perform the check again.
- If the left tuple is smaller than the right tuple, we keep incrementing the left pointer till they share the same value or left tuple is greater.

- If the left tuple is greater than the right tuple, we will clear the `matchingTuples` list and take the next right tuple with a different value and fill the `matchingTuples` list. Right pointer is incremented (possibly multiple times) if there are multiple right tuples with the same values.

2.3 Distinct

Distinct uses External sort algorithm. Instead of External sort sorting on the indexes of the join condition, it will sort on the indexes of the projected attributes instead. We then iterate through the sorted table and remove any tuples that is same as the previous tuple. Since external sort is involved, number of buffer pages inputted by the user is required to be at least 3.

2.4 Aggregate Functions

Aggregate functions basically perform a form of aggregation or mathematical function on the columns during the process of `Projection`. There are basically 5 main aggregate functions and these are `MAX`, `MIN`, `COUNT`, `SUM` and `AVG`.

The checking for the presence of Aggregation is basically done in the `Project` class as we only need to worry about aggregation after the columns to be projected have been determined. The following steps summarizes the algorithm used:

1. Loop through all the set of columns to be projected based on query.
2. For each column, the `Attribute` type is retrieved. There are 6 attribute types in total which are `Attribute.MAX` `Attribute.MIN`, `Attribute.COUNT`, `Attribute.SUM`,`Attribute.AVG` which represent the aggregated columns and `Attribute.NONE` which represent the non-aggregated columns.
3. Based on the type of aggregation, perform computation on all the tuples in the currently retrieved page and store these intermediary values in a data structure. There is also an `ArrayList` data structure that help to keep track of the order of columns in the tuple to be projected.
4. Repeat Step 3 over and over for each new page that is read from buffer. Compute the tuples in the page based on the attribute type and overwrite the old values previously stored.
5. Once there are no more pages to be read from buffer, this implies that the data that we have already aggregated is our final answer. This answer is then ordered based on the `ArrayList` we have been using to keep track of the order. The output tuple is then added to the output buffer and returned.

2.5 Optimizer

2.5.1 Existing work

In [IK90] the authors proposed a hybrid two-phase algorithm which runs Iterative Improvement [NSS86] then Simulated Annealing [KGV83]. The existing code however just executes Iterative Improvement a number of iterations (dependent on number of joins), picking a random starting query plan for each iteration. As iterative improvement only converges on a local minimum, running multiple iterations amplifies the probability that we converge on a global minimum instead.

2.5.2 New approach

We implement a hybrid approach using Cuckoo and Tabu Search [JS13]. Cuckoo search [YS09] is a biologically-inspired stochastic optimization algorithm which has displayed promising results in a variety of optimization problems. Tabu search [Glo89] is an aggressive local search which uses memory to avoid retracing random walks. As we are implementing an algorithm inspired by the breeding strategy of cuckoos [CB13], we sometimes refer to query plans as **eggs**.

2.5.3 Permutation functions

For this optimization algorithm we need two permutation functions P_1 and P_2. We choose $P_1(t)$ to be a random neighbour of the query plan tree t, obtained using a single/few application(s) of rules such as:

1. Join method - $R \bowtie_{\text{block nested}} S \equiv R \bowtie_{\text{sort merge}} S$,
2. Join commutes - $R \bowtie S \equiv S \bowtie R$,
3. Associativity - $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$.

P_2 however is a Lévy flight [Man82], that is a perturbation on the state space with a higher probability of greater step sizes.

2.5.4 Cost function

We largely reuse the existing cost function, which takes into account IO cost of the joins and the number of intermediate tuples produced. For the new methods that we implemented, their estimated I/O costs are given by the following equations, where B denotes number of buffers, and P_l and P_r be number of pages taken by left and right relations respectively.

2.5.4.1 Block Nested Join

$$\begin{aligned} n_{\text{outer blocks}} &= \left\lceil \frac{P_l}{B-2} \right\rceil \\ \text{total cost} &= P_l + n_{\text{outer blocks}} P_r \end{aligned}$$

2.5.4.2 Sort Merge Join

There are costs needed to sort the 2 relations and to perform the merge.

$$\begin{aligned} \text{sort}_l &= 2P_l \left(1 + \lceil \log_{B-1} \lceil P_l/B \rceil \rceil\right) \\ \text{sort}_r &= 2P_r \left(1 + \lceil \log_{B-1} \lceil P_r/B \rceil \rceil\right) \\ \text{merge} &= P_l + P_r \\ \text{total cost} &= \text{sort}_l + \text{sort}_r + \text{merge} \end{aligned}$$

2.5.4.3 Distinct

The cost for the distinct operator is the cost needed to sort the relation.

$$\begin{aligned} n_{\text{passes}} &= 1 + \lceil \log_{B-1} \lceil P/B \rceil \rceil \\ \text{cost} &= 2P n_{\text{passes}} \end{aligned}$$

where P is the number of pages taken up by the relation.

2.5.5 Algorithm

We follow the approach in [JS13] rather closely. It is parametrized by these values

- $n_1 = 4$ size of Tabu list,
- $n_2 = 2$ maximum Tabu variable,
- $n = 3$ number of eggs to generate initially.

We decided to just use these constant values proposed by the authors since we are implementing a stochastic genetic algorithm, and the runtime of each iteration is dependent on the magnitude of these parameters. As each iteration runs relatively quickly with these small parameters, we can afford to iterate more times than the existing Iterative Improvement algorithm. We decided to iterate this stochastic process $10 \cdot n_{\text{joins}}$ times for now.

To implement the Lévy flight permutation P_2, we simply iterated the neighbour function a random number of times (dependent on n_{joins}). Intuitively, as n_{joins} gets large, the space of query plans grows exponentially, and the space itself is not very well-connected (by the neighbor relation). As a result iterating the neighbour function from a query plan is unlikely to result somewhere near it, this satisfies the heavy tail condition.

3 Bugs and Limitations

3.1 Automated testing

To streamline our development progress we automated correctness checks. For the relation schemas and sample queries given in /testcases, we scripted (only tested on recent GNU/Linux)

- the generation of random databases (only for that schema),
- executing the queries,
- comparing output tuples against another SPJ engine implementation, such as PostgreSQL.

These scripts were automated using a continuous integration workflow.

3.2 Resolved issues

- Bug in `RandomDB` made it incapable of generating `REAL` values.
- Upon further inspection, cost of the implemented `NestedJoin` seems to be $P_l + P_l \cdot P_r$ instead.

3.3 Other identified issues

- `WHERE` clause only accepts a conjunction of conditions, and is implemented using iterated selections. The obvious way to handle it would be to edit the object model to deal with disjunctions accordingly, but this can be partially resolved by just implementing `UNION`/`UNION WITH`. Since every logical proposition is equivalent to one in disjunctive normal form, a query with `UNION` is same as one with `WHERE ... OR ... OR ...` modulo duplicates.
- When running experiments, we encountered the lack of dedicated `DATE`, `TIME` and `DATETIME` types. However `DATE` and `DATETIME` can be emulated using UNIX timestamps (which are `INTEGER`s), while `TIME` can be emulated using $\{0, \dots, 1339\}$.

4 Experiments

The generated tables and queries used are in /experiments.

Table 1: Sizes of generated relations used for experiments.

Relation	Size
Employees	50000
Certified	99997
Aircraft	40000
Schedule	99999
Flights	60000

For all experiments, we fixed $B = 25$ buffers each of size 4096.

4.1 Experiment 1

1. Employees $\bowtie$ Certified
   - Plan: `BlockNested(Certified [Certified.eid==Employees.eid] Employees)`
     Execution time: 76 seconds
   - Plan: `SortMerge(Certified [Certified.eid==Employees.eid] Employees)`
     Execution time: 11 seconds
2. Flights $\bowtie$ Schedule
   - Plan: `BlockNested(Schedule [Schedule.flno==Flights.flno] Flights)`
     Execution time: 99 seconds
   - Plan: `SortMerge(Flights [Flights.flno==Schedule.flno] Schedule)`
     Execution time: 12.5 seconds
3. Schedule $\bowtie$ Aircrafts
   - Plan: `BlockNested(Schedule [Schedule.aid==Aircrafts.aid] Aircrafts)`
     Execution time: 61 seconds
   - Plan: `SortMerge(Aircrafts [Aircrafts.aid==Schedule.aid] Schedule)`
     Execution time: 9.1 seconds

4.2 Experiment 2

For experiment two, we want the list of pilots who have been scheduled for a flight, say flight with flno = 35777, in relational algebra we query

$$\pi_{\text{eid,ename}}\sigma_{\text{flno}=35777}\,(\text{Employees} \bowtie \text{Certified} \bowtie \text{Schedule})\,.$$

- Plan: `Project(BlockNested(BlockNested(Select(Schedule 'Schedule.flno==35777) [Schedule.aid==Certified.aid] Certified) [Certified.eid==Employees.eid] Employees))`
  Execution time: 6.2 seconds
- Plan: `Project(SortMerge(Employees [Employees.eid==Certified.eid] SortMerge(Certified [Certified.aid==Schedule.aid] Select(Schedule 'Schedule.flno==35777))))`
  Execution time: 14.5 seconds
- Plan: `Project(NestedJoin(Employees [Employees.eid==Certified.eid] NestedJoin(Select(Schedule 'Schedule.flno==35777) [Schedule.aid==Certified.aid] Certified)))`
  Execution time: 5.0 seconds

4.3 Summary of results

In experiment 1 we performed joins on pairs of large relations. The results suggest that sort merge join is dramatically faster than block nested loops join. Our cost model predicted that sort merge would be faster but not by this much.

For experiment 2, in all cases the optimizer was able to select a plan which starts the join from the smallest relation first. As the intermediate relations using such a plan are all relatively small, we happen to have enough buffers for block nested join and plain nested-loop join to perform similarly. This suggests that sort merge join falls behind as the number of tuples involved gets smaller,we think this is mainly because that the overhead to sort is a much bigger percentage of the overall cost when the table is small, but when the tables get large, the sorting overhead becomes comparatively minimal. The number of passes required to sort the relation would be relatively small due to the logarithm property in the equation. The savings in the merging cost outweighs the overhead from the sorting cost. Hence, sort merge join is expected to perform better for larger relations than smaller relations.

From these experiments runned, we realised how query execution plan orderings can make a significant difference in a real-world system with larger data sets.

Data sets in the real world can be very asymmetrical. Setting a relation as the left or right relation to compute the join can have a large difference in the cost. Especially when computing the join of multiple relations, the intermediate table sizes have to be considered as well. Join is an expensive operation and ordering the query plans can help us to avoid cartesian products and keep the cost low as much as possible.

References

[CB13] Pinar Civicioglu and Erkan Besdok. “A conceptual comparison of the Cuckoo-search, particle swarm optimization, differential evolution and artificial bee colony algorithms”. In: *Artificial intelligence review* 39.4 (2013), pp. 315–346.

[Glo89] Fred Glover. “Tabu Search—Part I”. In: *ORSA Journal on Computing* 1.3 (1989), pp. 190–206. DOI: 10.1287/ijoc.1.3.190.

[IK90] Y. E. Ioannidis and Younkyung Kang. “Randomized Algorithms for Optimizing Large Join Queries”. In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’90. Atlantic City, New Jersey, USA: Association for Computing Machinery, 1990, pp. 312–321. ISBN: 0897913655. DOI: 10.1145/93597.98740. URL: https://doi.org/10.1145/93597.98740.

[JS13] Mukul Joshi and Praveen Ranjan Srivastava. “Query Optimization”. In: *International Journal of Intelligent Information Technologies* 9.1 (2013), pp. 40–55. DOI: 10.4018/jiit.2013010103.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680. ISSN: 0036-8075. DOI: 10.1126/science.220.4598.671. eprint: https://science.sciencemag.org/content/220/4598/671.full.pdf. URL: https://science.sciencemag.org/content/220/4598/671.

[Man82] Benoit B. Mandelbrot. *The fractal geometry of nature*. W.H. Freeman, 1982.

[NSS86] Surendra Nahar, Sartaj Sahni, and Eugene Shragowitz. “Simulated Annealing and Combinatorial Optimization”. In: *Proceedings of the 23rd ACM/IEEE Design Automation Conference*. DAC ’86. Las Vegas, Nevada, USA: IEEE Press, 1986, pp. 293–299. ISBN: 0818607025.

[YS09] X. Yang and Suash Deb. “Cuckoo Search via Lévy flights”. In: *2009 World Congress on Nature Biologically Inspired Computing (NaBIC)*. 2009, pp. 210–214.