CS3223 Project Report

Li Zi Ying (A0189497B)
Liaw Siew Yee (A0187880M)
Pang Jia Da (A0184458R)

1. Implementation

Our team has implemented the following operators for the given SPJ (Select-Project-Join) engine:

1. Block Nested Loops Join
2. Sort Merge Join (with External Sort Merge)
3. Distinct
4. Groupby
5. Orderby

1.1 Clarification on behaviour of operators when they differ from normal SQL

1.1.1 Orderby

Unlike traditional SQL engines, where the orderby ASC or DESC can be specified for each attribute in the attribute list, our SQL engine only supports 1 optional ASC or DESC, which is placed at the end of the attribute list. The sorting precedence follows the Orderby attribute list, where it is sorted on the first attribute if it differs, on the second if the first attributes are equal, etc. This Orderby type will apply to all attributes in the attribute list. If it is not specified, the default is ASC.

1.1.2 Groupby

As per clarification from Prof Tan, our implementation of Groupby without aggregates simply groups tuples in the output together. The grouping follows the Groupby attribute list, where it is grouped by the first attribute if it is equal among output tuples. Within that group, it is grouped on the second attribute, etc.

1.2 Discussion of Bugs

1.2.1. Erroneous assumption in Select cost calculation

The Select operator selects on a single attribute per select operation. We can thus define the selected attribute to be the attribute the selection condition is on, and the other attributes to be non-selected attributes. For example, if the output table size is 0.7 of the input table size, the PlanCost code assumes the number of distinct values of ***all*** attributes will be reduced by a factor of 0.7.

Fix [Implemented]: Only the selected attribute's number of distinct values will be scaled by a factor of 0.7. For the non-selected attributes, we assume they were uniformly distributed among the distinct values of the selected attribute, hence we only expect to see fewer occurrences of each distinct value, but the number of distinct values is unchanged. However, if the size of the output table is smaller than the number of distinct values, then the number of distinct values is bounded by and equals the number of records in the output table.

1.2.2 HashTable storing number of distinct values per attribute not updated

The hash table created to keep track of the number of distinct values per attribute in the Select cost calculation is not updated in PlanCost.java.
Fix [Implemented]: Added code to update hash table in PlanCost.java.

1.2.3 Program goes into infinite loop when tuple size exceeds page size

There is no error thrown when the tuple size for a given operator exceeds the page size and the program goes into an infinite loop.

Fix [Implemented]: Added code that prints out an error message and exits program when this occurs.

1.2.4 Other Limitations

a) RandomDB goes into an infinite loop when the number of records generated exceeds the range of the PK with no error thrown, thus the user may not be able to pinpoint why the program is hanging. *Proposed Fix*: Output an error message to prompt the user to generate values less than the given range.
b) RandomDB does not perform validation that each STRING primary key is unique. *Proposed Fix:* For a relatively small range of primary keys, an in-memory hash table can be created for the purpose of validating the uniqueness of each generated primary key.
c) The current SPJ engine does not accept composite primary keys for schemas.
d) The current SPJ engine does not process inequality joins between tables.

2. Experiments

In the experiments, we used a page size of 360B and 1800B. For the number of buffers, we used 10 and 1000.

Table	**No. of Records**	**Size of Tuple**	**<attribute name> <data type> <range> <key type> <column size>**
Flights	15000	72	flno INTEGER 20000 PK 4 from STRING 14 NK 28 to STRING 14 NK 28 distance INTEGER 100000 NK 4 departs INTEGER* 2360 NK 4 arrives INTEGER* 2360 NK 4
Aircrafts	15000	24	aid INTEGER 20000 PK 4 aname STRING 8 NK 16 cruisingrange INTEGER 10000 NK 4
Schedule	15000	8	flno INTEGER 20000 FK 4 aid INTEGER 20000 FK 4
Certified	15000	8	eid INTEGER 20000 FK 4 aid INTEGER 20000 FK 4
Employees	15000	36	eid INTEGER 20000 PK 4 ename STRING 14 NK 28 salary INTEGER 10000 NK 4

*As there is no data type for time, our group used the integer data type with a range not inclusive of 2360 to simulate timings from 0000 to 2359.

2.1 Experiment 1

Three 2-table joins were performed. The participating tables and SQL query used for each is indicated in the subsections below. For each join, we varied the join algorithm (only BNJ or SMJ), the page sizes (360 bytes, 1800 bytes) and the number of buffer pages (10, 1000) used, for a total of 8 timings.

2.1.1. Run joins on Employees and Certified tables (via eid)

SQL Query

```
SELECT *
FROM EMPLOYEES, CERTIFIED
WHERE EMPLOYEES.eid=CERTIFIED.eid
```

Page Size	No. of Buffers	Join	Time taken (s)
360	10	BNJ	59.488
		SMJ	21.767
360	1000	BNJ	6.153
		SMJ	7.877
1800	10	BNJ	41.692
		SMJ	11.463
1800	1000	BNJ	5.652
		SMJ	5.008

2.1.2 Run joins of Flights and Schedule (via flno)

SQL Query

```
SELECT *
FROM FLIGHTS, SCHEDULE
WHERE FLIGHTS.flno=SCHEDULE.flno
```

Page Size	No. of Buffers	Join	Time taken (s)
360	10	BNJ	87.185
		SMJ	32.244
360	1000	BNJ	8.423
		SMJ	12.104
1800	10	BNJ	19.723
		SMJ	16.908
1800	1000	BNJ	8.288
		SMJ	6.025

2.1.3 Run joins of Schedule and Aircrafts (via aid)

SQL Query

```
SELECT *
FROM SCHEDULE, AIRCRAFTS
WHERE SCHEDULE.aid=AIRCRAFTS.aid
```

Page Size	No. of Buffers	Join	Time taken (s)
360	10	BNJ	50.207
		SMJ	20.208
360	1000	BNJ	5.683
		SMJ	16.204
1800	10	BNJ	12.98
		SMJ	10.856
1800	1000	BNJ	5.877
		SMJ	5.205

2.1.4 Explanation of Results for Experiment 1

In general, it is observed that BNJ's performance improves as the number of buffer pages supplied increases. By increasing the number of buffer pages, the block size increases, thus reducing the number of times the right page is being read when performing joins with tuples from the left pages. SMJ's execution time also improves as the number of buffer pages supplied increases, since there are fewer passes required for the sort phase of the operation. For both BNJ and SMJ, the increase in page size also allows more tuples to be stored within a page, which means that there are fewer disk accesses required since fewer pages need to be fetched and written to for the same size of tuples.

When there is a limited number of buffers (10), SMJ outperforms BNJ, since the right pages have to be fetched more frequently for a small block size for BNJ compared to SMJ. However, as the number of buffer pages increase, the time taken for BNJ starts to match and even beat that of SMJ. In the worst case scenario where there are several duplicate values, BNJ may perform better than SMJ which has to backtrack partitions generated from the right table.

2.2 Experiment 2

In this query, we want to find the list of pilots who have been scheduled for a flight. We use 360 byte pages and 10 buffer pages.

SQL Query

```
SELECT *
FROM EMPLOYEES,CERTIFIED,SCHEDULE
WHERE EMPLOYEES.eid=CERTIFIED.eid,CERTIFIED.aid=SCHEDULE.aid
```

No.	Plan Used	Time taken(s)
1	(Employees $\bowtie_{BNJ}$ Certified) $\bowtie_{BNJ}$Schedule	996.579
2	Employees $\bowtie_{BNJ}$ (Certified $\bowtie_{BNJ}$ Schedule)	627.274
3	Employees $\bowtie_{BNJ}$ (Certified $\bowtie_{SMJ}$ Schedule)	512.711
4	Employees $\bowtie_{SMJ}$ (Certified $\bowtie_{SMJ}$ Schedule)	58.229

2.2.1 Explanation of Results of Experiment 2*

*Note: for all calculations that follow, the result of all divisions is rounded *up* to the nearest integer, i.e. we take the ceiling of the result. We assume that the intermediate number of output tuples is 15,000 for each join.

In experiment 2, Plans 1 and 2 differ in the order in which the joins occur. The following is an estimated calculation of the costs of these 2 plans. In this calculation, we assume that all 10 buffer pages are allocated for the joins.

$|Employees| = 1500$, $|Certified| = 334$, $|Schedule| = 334$
$Cost(Employees \bowtie_{BNJ} Certified) = 1500 + (1500/8)*334 = 64292$
$Cost(Certified \bowtie_{BNJ} Schedule) = 334 + 334/8 * 334 = 14362$
$Size\ of\ (Employees \bowtie_{BNJ} Certified)\ tuple = 40 \Rightarrow |\ Employees \bowtie_{BNJ} Certified\ | = 1667$
$Size\ of\ (Certified \bowtie_{BNJ} Schedule)\ tuple = 12 \Rightarrow |Certified \bowtie_{BNJ} Schedule| = 500$
$Cost(Plan\ 1) = Cost((Employees \bowtie_{BNJ} Certified) \bowtie_{BNJ} Schedule) = 64292 + 1667 + 1667/8 * 334 = 135{,}765$
$Cost(Plan\ 2) = Cost(Employees \bowtie_{BNJ} (Certified \bowtie_{BNJ} Schedule)) = 14362 + 1500 + 1500/8 * 500 = 109{,}862$

As seen from the calculation above, there is a huge difference in the I/O costs of plans 1 and 2, thus demonstrating that for large data sets, the order in which joins occur make a significant difference in query execution time and I/O costs.

The following is an estimated calculation for plan 4:
$Cost\ of\ sorting\ Certified = 2 * 334 * (1 + ceil(\log_9(334/10))) = 2004$
$Cost\ of\ sorting\ Schedule = Cost\ of\ sorting\ Certified = 2004$
$Assuming\ all\ partitions\ fit\ in\ memory\ during\ merge,\ Cost(Plan\ 4) = 2004 + 2004 + 334 + 334 = 4676$

There is a difference of more than 100,000 I/O for Plans 2 and 4, thus demonstrating that BNJ performs much more poorly compared to SMJ with a limited number of buffer pages. Plan 3, which uses both BNJ and SMJ, has an execution time that sits between Plans 2 and 4.

In large data sets such as the above, this difference in number of I/Os is significant as seen from our calculations, and thus shows that the choice of join algorithm in query execution plans is crucial for faster query execution.

2.3 Conclusion

The difference in execution speed for plans with different I/O complexities is magnified by the size of the input data sets. An inefficient execution plan might be able to run in reasonable time with small input, but quickly becomes intractable with large inputs. This is especially important in real world systems which typically use data sets on the order of millions to tens of million records, spanning hundreds of tables. An inefficient query plan in such a scenario is intractable.

In improving the I/O efficiency of query plans, the algorithm used for and order in which certain operators are called matters since it affects intermediate output table size and thus I/O costs involved in reading and writing those tables. Furthermore, for interactive (as opposed to batch) processing, execution plans can be ordered to reduce initial response time by quickly finding and returning the first few matching tuples to the user, while the rest of the query carries on in parallel.