Kenny Yu
HUID: 30798260
CS165 Project 3 Performance Report

Join Performance

I tested the performance of loading the data from the CSV and then executing the following SQL queries (and its equivalent query plans):

SQL:

```
DROP TABLE IF EXISTS r;

CREATE TABLE r (
ra INT NOT NULL,
rc INT NOT NULL,
rd INT NOT NULL);

CREATE INDEX rc_index ON r (rc) USING BTREE;

DROP TABLE IF EXISTS s;

CREATE TABLE s (
sa INT NOT NULL,
sf INT NOT NULL,
sg INT NOT NULL);

CREATE INDEX sf_index ON s (sf) USING BTREE;
CREATE INDEX sa_index ON s (sa) USING BTREE;

LOAD DATA LOCAL INFILE 'r.csv' INTO TABLE r
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
(ra, rc, rd);

LOAD DATA LOCAL INFILE 's.csv' INTO TABLE s
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
(sa, sf, sg);

SELECT COUNT(r.rc)
```

```
FROM r
WHERE r.rc >= 1 and r.rc <= 9;

SELECT COUNT(s.sf)
FROM s
WHERE s.sf >= 31 and s.sf <= 99;

SELECT MAX(r.rd), MIN(s.sg), COUNT(r.rd), COUNT(s.sg)
FROM r,s
WHERE r.ra = s.sa
    AND r.rc >= 1
    AND r.rc <= 9
    AND s.sf >= 31
    AND s.sf <= 99;
```

QUERY PLAN:

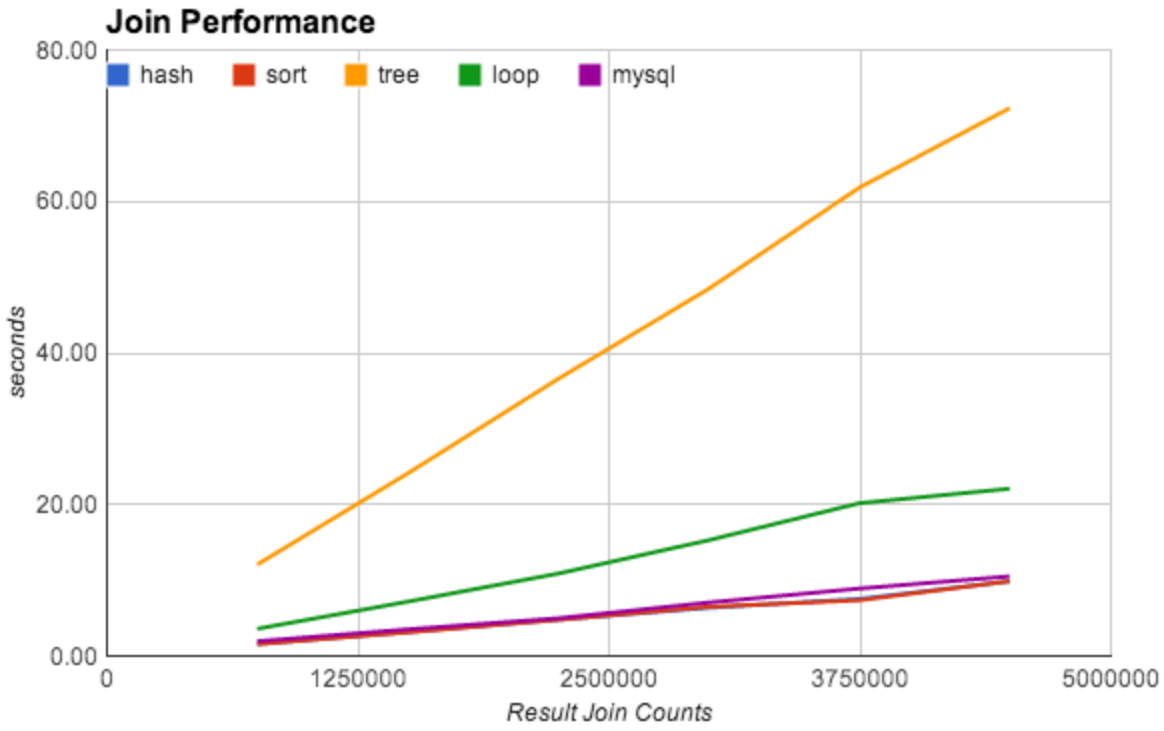
```
create(rc,"b+tree")
create(ra,"unsorted")
create(rd,"unsorted")
load("r.csv")
create(sf,"b+tree")
create(sa,"b+tree")
create(sg,"unsorted")
load("s.csv")
rc_inter=select(rc,1,9)
sf_inter=select(sf,31,99)
join_input1=fetch(ra,rc_inter)
join_input2=fetch(sa,sf_inter)
count(join_input1)
count(join_input2)
r_results,s_results={hash,sort,tree,loop}join(join_input1,join_input2)
rd_values=fetch(rd,r_results)
sg_values=fetch(sg,s_results)
maxr=max(rd_values)
mins=min(sg_values)
cr=count(rd_values)
cs=count(sg_values)
tuple(maxr,mins,cr,cs)
```

See the perftests directory for the python script gen.py to generate the csv data and scripts to run the performance tests. Here are the parameters I used:

- "selrater": 0.75, [selection rate on column rc]
- "numr": multiples of 100,000 [number of rows in r]
- "nums": 10000, [number of rows in s]

- "selrates": 1.0, [selection rate on column sf, 100% to allow treejoins]
- "seed": 42,
- "amax": 1000, [number of different keys for ra = sa to join on]

result join count	hash	sort	tree	loop	mysql
748374	1.58	1.57	12.07	3.59	1.94
1496583	3.15	3.12	24.09	7.17	3.51
2248328	4.80	4.78	36.61	10.88	4.98
2997238	6.37	6.49	48.48	15.29	7.06
3747569	7.53	7.36	61.85	20.17	8.90
4495205	9.86	9.88	72.32	22.08	10.51



In my database:

- sort and hash joins perform very similar to one another (when all the data fit in memory)
- tree joins are slow, presumably from all the IO when probing the btree
- loop joins are about twice as slow as sort and hash joins
- sort, hash perform just as well as mysql
- If I do NOT build an index on s.sa in mysql, the time for the queries take on the other of MINUTES instead of seconds.

Limitations

- I allow tree joins only on selections that select an entire column with a btree index.
- I can only support data that fit in memory

Challenges

The biggest challenge was debugging my btree implementation. After doing performance tests with ~1 million rows, I discovered bugs with my btree implementation, resulting in incorrect number of rows being selected. I solved this by carefully stepping through gdb to analyze my invariants. As an extension, I will implement a btree file dump utility to examine the contents of a btree index file programmatically from the command line without the need to step through it via gdb.

Future Extensions

- Introduce an iterator for iterating over select and fetch results
    - This will allow me to lazily materialize these values only when they are needed
    - This will also allow me to vectorize the processing of the data
- External sorts and external joins