National University of Singapore
School of Computing

CS3223: Database Systems Implementation
Semester 2, 2015/16

Assignment 3: Query Evaluation (9 marks)

Due: 11:59pm April 13, 2016 (Wednesday)

In this assignment, we examine the evaluation of join queries. As in the previous assignments, this assignment is to be done using your assigned compg compute node.
Late submission penalty: There will be a late submission penalty of 1 mark per day up to a maximum of 3 days. If your assignment is late by more than 3 days, it will not be graded and you will receive 0 credit for the assignment.

1 Query Evaluation in PostgreSQL

This section presents information on some of `PostgreSQL`'s evaluation operators and configuration parameters that are relevant for this assignment.

1.1 Sort Operator

In `PostgreSQL`, the amount of main memory allocated for performing a sort operation is controlled by a configurable parameter `work_mem` which has a default value of 4MB. If the set of tuples to be sorted can fit entirely in the allocated memory, the tuples will be sorting using a main-memory quicksort algorithm; otherwise, `PostgreSQL` evaluates the sort operation using a disk-based external sorting algorithm.

1.2 Unique Operator

`PostgreSQL`'s Unique operator is used to eliminate duplicate records. The operator takes as input a sorted sequence of records R that is ordered with respect to some sort key K and outputs a duplicate-free (with respect to K) subset of R. That is, if R' is the output relation produced by the Unique operator on an input relation R ordered by sort key K, then (1) $\pi_K(R') = \pi_K(R)$ and (2) $t.K \neq t'.K \ \forall \ t, t' \in R', t \neq t'$.

1.3 HashAggregate Operator

`PostgreSQL`'s HashAggregate operator is used for partitioning an input set of records into a collection of groups via hashing. This operator could be used for computing group-by aggregation, eliminating duplicate records with respect to some key, etc.

1.4 Semi-Joins and Anti-Joins

There are two special types of join operator, termed semi-join and anti-join, that are useful for evaluating EXISTS subqueries and NOT EXISTS subqueries, respectively. The *semi-join of R and S* (denoted by

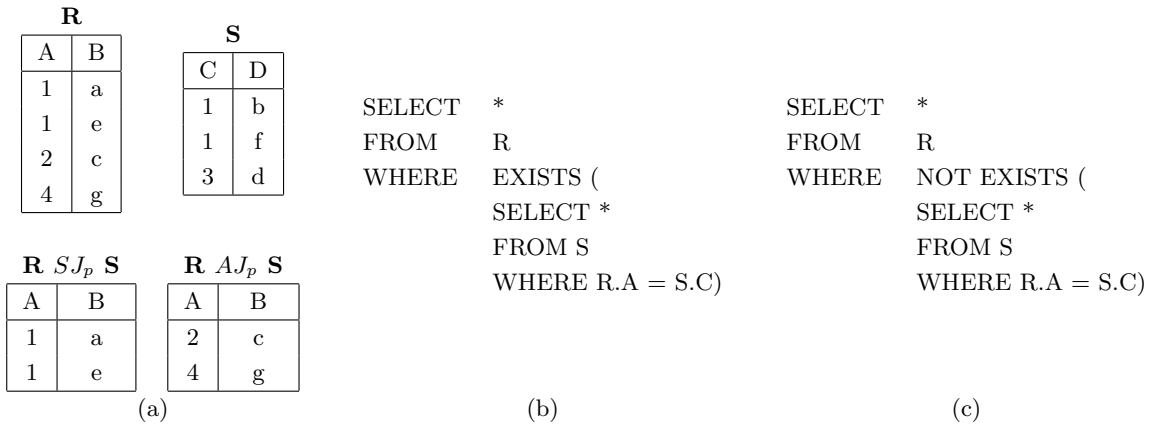


Figure 1: Semi-join & Anti-join (a) SJ_p & AJ_p with join predicate $p : R.A = S.C$ (b) Query Q_1 with EXISTS-subquery (c) Query Q_2 with NOT EXISTS-subquery

R SJ S) returns the set of tuples in R that join with some tuple in S. The *anti-join of R and S* (denoted by R AJ S) returns the set of tuples in R that do not join with any tuple in S.

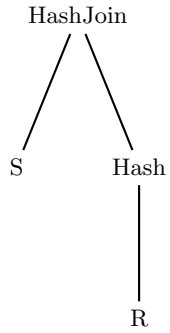
An illustration of these two join operators (with join predicate $R.A = S.C$) is shown in Figure 1(a). Observe that the query Q_1 in Figure 1(b) is equivalent to $R\ SJ_p\ S$ and the query Q_2 in Figure 1(c) is equivalent to $R\ AJ_p\ S$ with the join predicate p given by $R.A = S.C$.

Both of these join operators are implemented in `PostgreSQL` using variants of its join algorithms (e.g., nested-loop, sort-merge, and hash join).

1.5 Hybrid Hash Join Algorithm

The join algorithms available in `PostgreSQL` include the index nested-loop join, sort-merge join, and hash join.

`PostgreSQL` implements a variant of the Grace Hash Join algorithm called the Hybrid Hash Join algorithm. Conceptually, this is implemented in terms of two operators: *Hash* and *HashJoin* as illustrated by the following query plan for $R \bowtie S$.



The *Hash* operator builds a hash table for the build relation R. The amount of main memory allocated for the hash table is also controlled by the `work_mem` parameter. If R is too large for its hash table to fit

entirely in the allocated main memory, the *Hash* operator will partition the tuples in R into partitions (referred to as batches in `PostgreSQL`'s terminology) and stores only the first partition of tuples in the hash table during the partitioning of the build relation. Tuples belonging to other partitions are written to temporary files on disk, and the hash join operation will be performed in partitions. Thus, at the end of scanning R, R is partitioned into partitions $R = R_1 \cup \cdots \cup R_k$, where the tuples in partition R_1 are stored in a main-memory hash table and each of the remaining partitions is written to a temporary file on disk.

The *HashJoin* operator performs a scan of the tuples in the probe relation S. For each scanned tuple t in S, if t belongs to the same partition as the build tuples in the hash table, t will be will be used to probe the hash table; otherwise, t will be written to an appropriate temporary file on disk. Thus, at the end of scanning S, S is partitioned into partitions $S = S_1 \cup \cdots \cup S_k$ with $R_1 \bowtie S_1$ computed (by probing the hash table with tuples belonging to S_1), and each partition S_i, $i > 1$, written to a temporary file on disk.

If $k > 1$, `PostgreSQL` will compute $R_i \bowtie S_i$ for each remaining partition i by reloading the main-memory hash table with tuples from R_i and scanning S_i to probe the hash table.

In contrast to the Grace Hash Join algorithm, which computes $R_1 \bowtie S_1$ during the probing phase, the Hybrid Hash Join algorithm computes $R_1 \bowtie S_1$ during the partitioning phase by building a hash table for R_1 as R is being partitioned.

1.6 Query Plan Configuration Parameters

`PostgreSQL` provides several optimization-related parameters that can be configured at run-time to influence the choice of query plans selected by the query optimizer. For this assignment, the following four boolean parameters are relevant:

- **enable_indexscan**: enables or disables the query optimizer's use of index-scan plan types. The default value is on.
- **enable_nestloop**: enables or disables the query optimizer's use of nested-loop join plans. It is impossible to suppress nested-loop joins entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default value is on.
- **enable_mergejoin**: enables or disables the query optimizer's use of merge-join plan types. The default value is on.
- **enable_hashjoin**: enables or disables the query optimizer's use of hash-join plan types. The default value is on.

These parameters can be configured with the `SET` command; e.g., SET enable_hashjoin = off.

2 Getting Started

For this assignment, ensure that you are using the original version of `PostgreSQL`. Perform the following setup for this assignment:

```
$ cd $HOME
$ wget http://www.comp.nus.edu.sg/~cs3223/assign/assign3.zip
$ unzip assign3
$ cd assign3
$ pg_ctl start
$ createdb assign3
$ bash setup.sh
```

The `assign3` sub-directory created in your home directory contains the following files:

- `restore-pgsql-linux.sh`: This script re-installs `PostgreSQL`. You do not need to run this script if you had already installed the original `PostgreSQL` from the last assignment.
- `setup.sh`: The `setup.sh` script creates and installs an SQL extension named `dropdbbuffers`. With this extension, you can use the SQL command "select dropdbbuffers('assign3')" to clear all disk blocks belonging to the `assign3` database from the buffer pool.
- `dropdbbuffers/`: This is a directory containing the `dropdbbuffers` extension to be installed by `setup.sh`.
- `loaddata.sh`: This is a script to load data into relations r and s, and creates indexes on them. This script requires two input parameters: (1) the number of tuples in relation r and (2) the join factor. For example, the command "./loaddata.sh 100000 10" will set up the database for Experiment 1 with 100,000 tuples in relation r and a join factor of 10.
- `sample.sh`: This is a sample script to run Experiment 1A with the indexed nested-loop join algorithm.
- `expt1a.pdf`: This is a sample performance graph for Experiment 1A that is provided for your reference.

2.1 Required Readings

The following `PostgreSQL` documentation should be read before you start on this assignment.

1. How to read query plans generated by `EXPLAIN` command.
   `http://www.postgresql.org/docs/9.4/static/using-explain.html`
2. Manual page for `EXPLAIN` command.
   `http://www.postgresql.org/docs/9.4/static/sql-explain.html`
3. Details of run-time configuration parameters to influence `PostgreSQL`'s selected query plans.
   `http://www.postgresql.org/docs/9.4/static/runtime-config-query.html`
4. Description of statistics collector for monitoring database activity.
   `http://www.postgresql.org/docs/9.4/static/monitoring-stats.html`

3 What to do

In this assignment, you will compare the performance of evaluating the following three queries, where V denote a constant value.

```
QA: SELECT  r.c, s.z
    FROM    r JOIN s ON r.a = s.y
    WHERE   r.b <= V
```

```
QB: SELECT  z
    FROM    S
    WHERE   EXISTS
            (SELECT *
            FROM    r
            WHERE   r.a = s.y
            AND     r.b <= V)
```

```
QC: SELECT  z
    FROM    S
    WHERE   NOT EXISTS
            (SELECT *
            FROM    r
            WHERE   r.a = s.y
            AND     r.b <= V)
```

The database consists of two relations, $r(a, b, c)$ and $s(x, y, z)$, with the following properties:

1. $\pi_a(r) = \{1, 2, \cdots, ||r||\}$. For each tuple t in r, we have t.a = t.b.
2. $\pi_y(s) = \pi_a(r)$, and each tuple in r joins with exactly JF tuples in s. We refer to JF as the join factor. Thus, $||s|| = JF \times ||r||$.
3. There are two B$^+$-tree indexes: *ra_idx* is an index on attribute *r.a*, and *sy_idx* is an index on attribute *s.y*.

You are to conduct the following experiments to measure the performance (in terms of execution time) of different join algorithms for evaluating the three queries. Experiment 1 examines the effect of the selectivity factor of the selection predicate, Experiment 2 examines the effect of the join factor, and Experiment 3 examines the effect of the *work_mem* parameter. Each experiment is to be run on the three queries QA, QB, and QC. Table 1 shows the parameter values to be used for the nine performance experiments.

Expt	**Query**	$\|\|\mathbf{r}\|\|$ ($\times$**1000**)	**JF**	**work_mem**	**V** ($\times$**1000**)
1A	A	100	10	4MB	$\{12.5, 25, 37.5, 50, 75, 100\}$
1B	B	100	10	4MB	$\{12.5, 25, 37.5, 50, 75, 100\}$
1C	C	100	10	4MB	$\{12.5, 25, 37.5, 50, 75, 100\}$
2A	A	100	$\{5, 10, 20, 40, 80\}$	4MB	25
2B	B	100	$\{5, 10, 20, 40, 80\}$	4MB	25
2C	C	100	$\{5, 10, 20, 40, 80\}$	4MB	25
3A	A	100	10	$\{2, 4, 8, 16, 32\}$ MB	25
3B	B	100	10	$\{2, 4, 8, 16, 32\}$ MB	25
3C	C	100	10	$\{2, 4, 8, 16, 32\}$ MB	25

Table 1: Parameter values for performance experiments

For each experiment, you are to measure the execution time for each of the following four join algorithms:

- **INLJ**: Indexed Nested-Loop Join
- **SMJ**: Sort-Merge Join without using any index (i.e., with index scan disabled)
- **SMJ-I**: Sort-Merge Join with index scan enabled
- **HJ**: Hash Join

You should clear the database buffer (using SELECT dropdbbuffers('assign3')) and OS cache (using /opt/bin/dropcache) between query executions. Refer to sample.sh for how to do this within a script.

4 What & How to Submit

For this assignment, you are to submit a report (in pdf format) with the following information:

1. A performance graph for each of the nine experiments (Experiments 1A, 1B, $\cdots$, 3C) comparing the performance of the four join algorithms (INLJ, SMJ, SMJ-I, HJ). with the x-axis representing the parameter being varied and the y-axis representing the execution time. Refer to expt1a.pdf for an example.
2. A short discussion (at most half a page) of the key experimental observations.

Name your report pdf file using the student number of one of the team members (e.g., "a0123456x.pdf") and upload it to IVLE's `Submission-Assignment-3` workbin. Each team should upload only one submission.