Advanced Databases Project 2 Report

Team 7

Diggory Hardy
Diwakar Sapan
Elmas Ferhat
Goyal Pulkit
Tran Le Hung
Vidmantas Zemleris

Table of Contents

1. Data Distribution

Data partitioning

To optimise the performance of several of the equijoins in queries 7 and 8, we determined that partitioning should respect the following rules:

1. CUSTOMER and ORDER tuples with the same CUSTKEY should appear on the same node; since each LINEITEM tuple corresponds to a single CUSTKEY, line-item tuples should also be stored on the same node as CUSTOMER and ORDER tuples with this CUSTKEY
2. PART and PARTSUPP tuples with the same PARTKEY should appear on the same node
3. Data within each of the largest six relations should be distributed evenly across all eight nodes, while the smallest two relations (NATION and REGION) could be stored exclusively on a single node.

Enumerating the nodes from 0 through 7, we used the following, which matched our requirements well:

1. Store the NATION and SERVER relations exclusively on server 7
2. For each CUSTOMER, ORDER and LINEITEM tuple, find the relevant CUSTKEY, then assign the tuple to node (CUSTKEY mod 8)
3. For each PART and PARTSUPP tuple, assign to node (PARTKEY mod 8)
4. For each SUPPLIER tuple, assign to node (SUPPKEY mod 8)

We found this to distribute data extremely evenly. Of course, this took advantage of the artificial way in which the data was generated, but even with real data, if keys are allocated randomly, so long as there are a large number of unique keys (as in this case), such a partitioning should be fairly even.

Distributing data

Sample data sets are generated by the TPC-H dbgen tool. This generates our eight tables. To partition the data as above, we considered three possible approaches:

1. Generate all data on one node, then copy.
2. Generate and import all data on each node, then delete unwanted data.
3. Generate only the required data on each node.

The first approach involved copying a lot of data across the network, which was undesirable. We considered the second, but since we wished to be able to generate data sets for large scale factors and disk space was insufficient to generate all data locally beyond a scale factor of 20, this approach was considered suboptimal.

The third approach involved modifying the dbgen sources, but proved quite straightforward. Our modified sources are attached. The modification enables a -p option controlling which partition data should be generated for. The included scripts make (re)generation of data sets with custom scale factor simple.

2. Bloom Join

Generating BitVector in SQL

The bloom join could be quite easily implemented solely in SQL to give good performance. There are at least two ways of generating the bit-vector:

- store bitvector in environment variable, and modify bits with INSERT(buffer, pos, 1, val)[1]. Example: if *hashes* contained hashes that have some tuples and @bf is the final buffer:

```
SELECT @cn:=@cn+1 AS cnt,
    @bf:=INSERT(@bf, hash+1, 1, '1') AS bf FROM hashes AS t
ORDER BY cnt DESC LIMIT 1;
```

- use **group_concat** to concatenate the rows representing list of hash states (0/1)

After evaluating the performance of both method, we noticed that the **group_concat** variant is much more performant for larger bit-vector sizes (it seems that setting a character in a string buffer with INSERT is quadratic, probably mysql is making a copy of the string each time). So we stick to this method, and explain it further below.

For the second method we need a couple of simple tricks:

- group_concat by default is limited 1024 characters, however this is easily fixable (even in runtime) by setting *group_concat_max_len* system variable[2]
- then to get bitmap we need to compute hash states within all the hashing range:

```
SELECT MAX(active), hash
FROM  (select 1 as active, hash from hashes)
UNION (select 0 as active, hash from all_numbers)
group_by hash
```

  however, here we need to generate list of all numbers within the given range (0, hashmax), however MySQL do not not contain any similar function, but we can just create a table containing list of all numbers. This can also be easily done in sql with many cross-products, union all and a row counter:

```
insert into all_numbers select
  @i := @i + 1 as number
from
  (select 0 union all select 1 … union all select 9) as t0,
  ….
  (select @i:=0) as t_init;
```

Hash function

We used a simple hash function based on primes (we also always force the hash length to be prime), that seem to give fairly good distribution and is easy to implement on SQL:

---

[1] based on idea of using environment variables from http://www.astorm.ch/blog/index.php?post/2011/06/26/BloomJoin-avec-MySQL

[2] http://dev.mysql.com/doc/refman/5.0/en/server-system-variables.html#sysvar_group_concat_max_len

```
SELECT DISTINCT ((JOINKEY*107+1009) MOD @hashlen) AS hash, 1 AS active
FROM <local table(s)>
WHERE <filtering_criteria>
```

@hashlen is chosen to be a prime=99,881 so to optimize both the performance of bloom filter calculation and not to have much of collisions (based on selectivity calculations for join tables for feasible scale ratios). For larger scale ratios one may wish to select a larger hash length or smaller for very small tables.

Bloom Filter in SQL

Once we have the bitvector, we want to do the bloom-filtering on the remote machine. This is also fairly simple in SQL, using SUBSTRING to extract a certain bit-vector position from the input string joined with all_numbers from the hashing range. Once we have a list of active hashes, we can then check if a certain row have to be filtered or not (hash_function may be inlined, but we omit this for simplicity).

```
SELECT *
FROM <tables>
WHERE <explicit_filtering_criteria> AND
      hash_function(JOINKEY)  in (
        -- decode bit vector into a set of rows
        select hash from
        (
          SELECT SUBSTRING(s, hash+1, 1) AS bit, number AS hash
          FROM all_numbers, (SELECT uncompress(@bit_vect) AS s) bvect
          WHERE number < char_length(s)
        ) as bit_vector
        where bit='1');
```

Bit-vector compression

Further for larger bit-vectors one may compress the resulting string representation of bit vector into binary form and use Gzip-like compression by using compress/uncompress functions.

Note: we are not using this for simplicity as the bit-vectors from multiple remote machines are OR'ed in the Java code on the main machine and they are 100K characters only.

3. Query Q7

3.1 Query Plan

Figure 1 illustrates the query plan for Q7. The objective of the query plan is to make the process run parallelly as much as possible. The first step is filter the nation based on the input parameters. Then, we run the query on node 7 to obtain the nation keys for two input parameters. We can then send this “small” data to the different servers to compute further joins. As described above, line item, customer and order are distributed by by customer key and therefore we can directly join them on each nodes without having to worry about data from other nodes. Since Supplier is distributed differently, we have to perform bloom join to obtain the final query results. The first step for this bloom join computation is to obtain bitvectors for *LineItem* |><| *filter_date(Order)* |><| *Customer* |><| *IntdN1Region* join on suppkeys. Since the data is distributed on all servers, we obtain this bit vector from all servers parallelly and then finally OR all the bit vectors. We then send the bitvectors to all nodes to obtain supplier keys. This is also done in parallel for all nodes. For the final step in bloom join processing, we insert the supplier and nation data into temporary tables at each server (again in parallel for all nodes). Finally, we perform the complete join on each node and obtain partial aggregates for the data which are then combined on the server. Each level in the query plan denotes one level of computation which is performed in parallel.

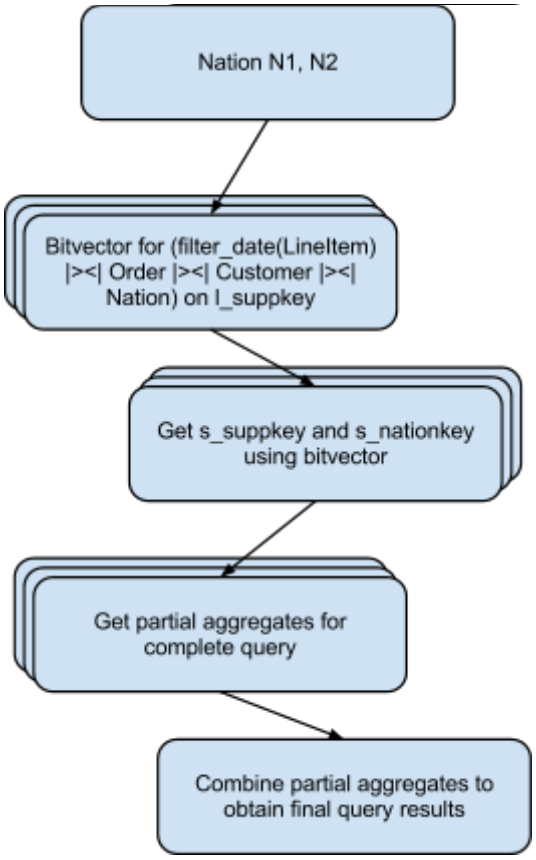

Figure 1: Query plan for Q7

3.2 Experiment Results

Scaling Factor	Running time (milliseconds)
0.01	1 479
0.1	2 637
1	8 939
10	61 880
20	104 608
29	174 186

4. Query Q8

4.1 Query Plan

The query plan for Q8 is as described in figure 2. In order to attain good efficiency, we try to parallelize our calls. Our first step is to get the nation keys using the join between Nation and Region. Since we have a filter on region and nation and region are stored on the same node and the size is quite small, we perform the join and download the results to the gateway. We can then send this “small” data to the different servers to compute further joins. Since we distribute line item, customer and order by customer key, we can easily perform local joins on these three tables without having to worry about performing bloom joins. We however need to perform bloom join between part and lineitem and supplier and lineitem because these relations are quite large. The first step for this bloom join computation is to obtain bitvectors for *LineItem* |><| *filter_date(Order)* |><| *Customer* |><| *IntdN1Region* join on partkeys and suppkeys. We do this task parallely on all servers (We run parallel threads for obtaining bit vector for suppkey and for part key and within these tasks, we parallely run queries on all servers to get the bitvectors from each site) and then OR the obtained bit vectors. We then send the bitvectors to all nodes to obtain supplier keys and part keys. This is also done in parallel for suppliers and parts and also in parallel for all nodes for each relation. For the final step in bloom join processing, we insert the data into temporary tables at each server (again in parallel for all nodes for both relations). Finally, we perform the complete join on each node and obtain partial aggregates for the data which are then combined on the server. Each level in the query plan denotes one level of computation which is performed in parallel.

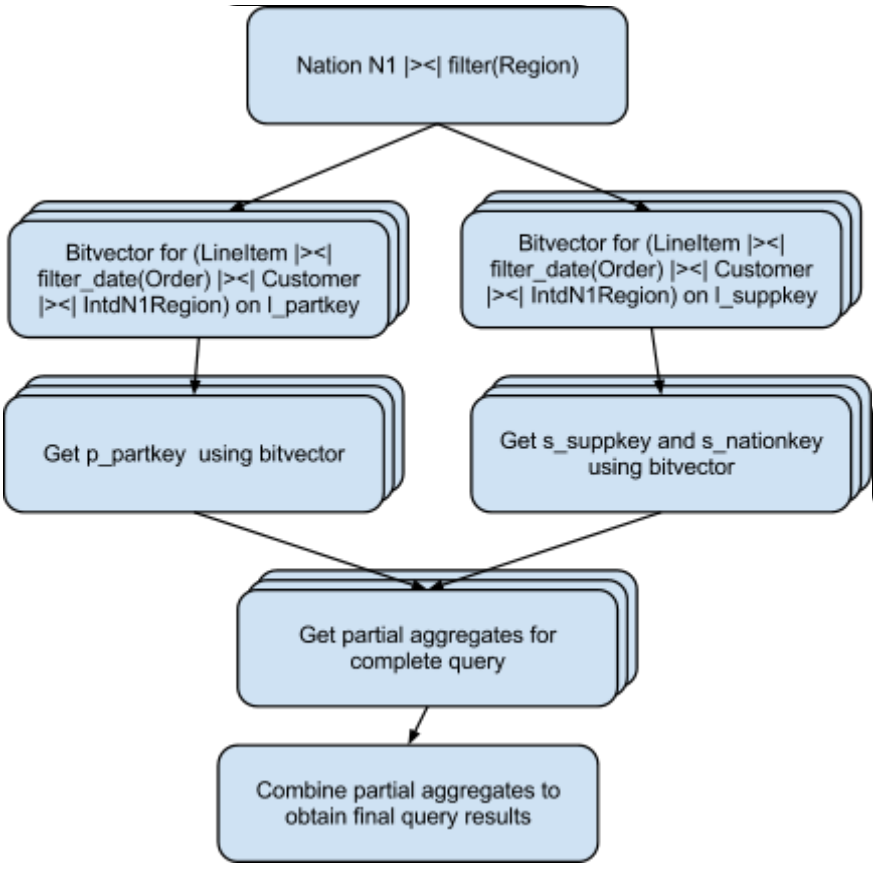

Figure 2: Query plan for Q8

4.2 Experiment Results

Scaling Factor	Running time (milliseconds)
0.01	3 207
0.1	4 227
1	22 996
10	179 060
20	385 770
29	572 700

5. Discussion

From the experimental results, we see that the results from Hadoop are good for large data (scale factor 10 and more) whereas our distributed queries outperform Hadoop for smaller scale factors. This is because Hadoop is optimized for large scale distributed processing and distributes the tasks to several mappers and reducers. Our distributed query processing is however limited to 8 servers only which makes the results slower for larger scale factors.

Another disadvantage of using our distributed query processor is that we have a central server which manages all the data. This means that the central server has to receive the data from all the joins and then send it again to other nodes which increases the time taken while passing the data through the network which is the main bottleneck for our tasks. On the other hand, Hadoop pipelines the data efficiently between the mappers reducing the cost incurred with a central server.

Could completely distributed execution with p processors improve the situation?

A possible improvement for very large scale ratios could be to execute query in completely distributed fashion, where using bloom join each machine fetches only the remote data needed to do the join subset of tuples stored on the locally, and then we would be centrally combining only the partial aggregates from each of p processors.

For all the following reasons below and the fact that the given infrastructure is not able to run at large scale because of data-size limitations we have chosen the simpler option (according to the specification document we don't have to implement this).

First with this implementation we would need more complex dataflow, we have to send partial results between different processors, and we have to execute O(p^2) queries instead of O(p)) in total, which may also cause considerable overhead for smaller scaling ratios.

Second it is much more more complex and not clear how to implement such distribute system given the current system infrastructure:

- One could use MPI, but a) is it installed b) it is quite nasty to use
- Hadoop is not quite designed for such workflows
- as a hack one could start workers by sending ssh commands and piping data to every worker with pubkey authentication or storing materialized data temporary at the database serves

6. Manual

6.1 Usage

java -jar dpch-team.jar <Query> <Query Parameters>

Examples

java -jar dpch-team.jar q7 FRANCE GERMANY
java -jar dpch-team.jar q8 BRAZIL AMERICA "ECONOMY ANODIZED STEEL"

6.2 Data Distribution

./populate.sh <scale-factor>

*Notes: Press Enter after eight "success" are displayed