Advanced Databases Project 3 Report

Team 7

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

1. Design and Assumptions

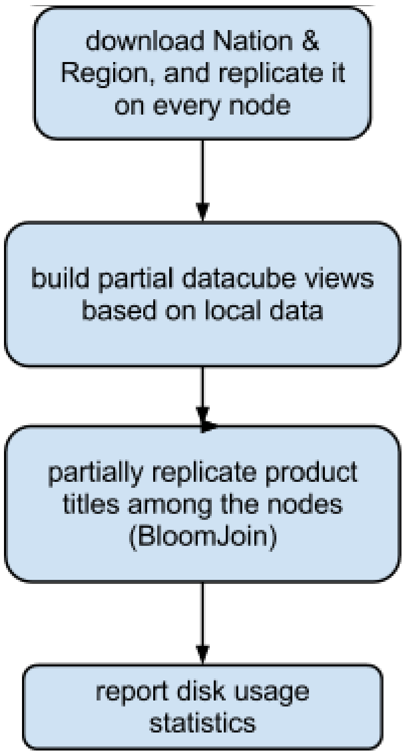
The main design goals were performance, scalability, low disk and network usage, and ease of use, which we present below.

Our cube is materialised for the queries which summarise large numbers of tuples, but not where the query requires accessing only a few tuples for each result line. This is primarily because of size constraints; secondarily, the high levels of detail are not particularly important since the amount of data output approaches the amount of data to be read, and limiting the output to a sensible amount to fit on the screen effectively limits the amount of data that needs to be read or processed.

1.1 Reusing data distribution from phase 2 of the project

Data is stored mostly as in phase 2 of the project. The differences are two-fold: firstly, we stored some the relations which are replicated on all nodes. In general they are small enough that the extra storage space used is negligible; the advantage of doing this over partitioning the relations or storing them exclusively on one server is that joins against these relations can be done on every node without waiting for inter-node data transmission.

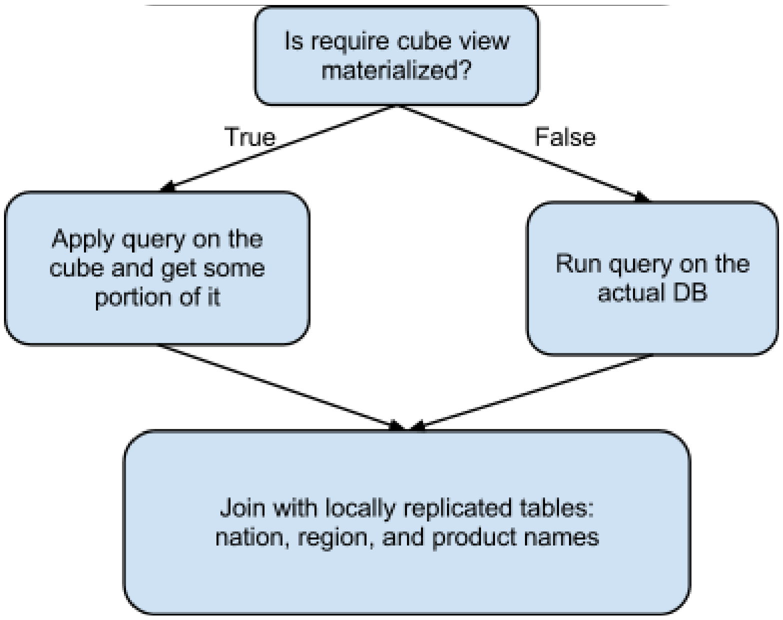
2. DataCube Builder


download Nation & Region, and replicate it on every node
build partial datacube views based on local data
partially replicate product titles among the nodes (BloomJoin)
report disk usage statistics

The first step in building the data cube is to obtain the data for the dimensions that we will use. We mention in design decisions that we would be storing ids in the data cube rather than materializing the strings for the dimensions in order to save storage costs. In this case, we store the ids and strings in a separate relations so that we can join them on the fly to perform cube operations.

Figure on the right presents the execution strategy for this step. We first join nation and region and download the results which we send to all nodes for joining with supplier. The supplier is then joined with nation and region to get the dimensions for supplier locations. Nation and region is then further used to obtain customer locations. We don't show this step in this plan as this is performed while building the data cube.For example, the supplier locations can be obtained by joining supplier with nation and region to obtain two zoom levels for supplier locations.
For building the final data cube, we run the queries for dimension combinations in batch, as some may be quick and others slow. We build separate materialized TABLES for each set of dimensions – having separate tables allows taking advantage of the fact that only one dimension zoom-levels combination is being accessed at the time: a) having all data in one place with specific clustered indexes is much better than filtering one huge table which may have millions of tuples, even if we needed just a couple of tuples for a small view, b) we can store only the required columns saving us some space. This is managed by the CubeSchemaMgr which handles the list of dimensions, zoom levels and builds queries for materializing the views.

3. Online Query Processing


Is require cube view materialized?
True
False
Apply query on the cube and get some portion of it
Run query on the actual DB
Join with locally replicated tables: nation, region, and product names

Given a query, there are two ways our system might process it.

If the query can be applied against a materialised dimension of the data cube, we start by doing that. We then join the results against the local copy of the nation/region table, and return them.

The other case requires running the query against the database. Since results for more broader queries have been stored in the data cube but not any containing the level of detail needed, we need to make the query against the original data relations. As mentioned in section 1.1, this query is not slow to evaluate since we restrict it to the first N lines of output. The same code used to generate parts of the data-cube can be used to generate this output, making the query simple to specify.

3.1 Operations

We have implemented roll-up, drill-down, slice, dice, limit, dump and undo operations. CubeOperations class keeps a current view of the cube and a stack including all previous views to enable the undo operation. After executing rollup, drilldown, slice, dice, limit or undo, the corresponding query is run to display updated results.

The roll-up operation takes a dimension defined by the CubeSchemaMgr class and uses the CubeDimension class to decrease zoom level. For example, with the time dimension it finds the current zoom level by querying the current view, then steps from, for example, the by-day zoom level, to a monthly year level. Drill-down does exactly the same thing but in the opposite direction.

The slice and dice operations use the ParameterSelection class. This contains upper and lower bounds for some dimension. When used, a WHERE clause is appended to the appropriate query.

The final two operations are limit and dump. Limit simply sets the maximum number of results to display (“LIMIT x” clause at the end of a MYSQL statement) and reruns the query. Note that this limit is not stored on the view stack; hence issuing an undo operation will not reset the limit; this seemed reasonable from a usage point-of-view. The dump operation executes a query like normal, but redirects the output to a file. It also uses an independent limit since it is likely that more data will be wanted in a file; however it still uses some limit since copying all available rows into a local file would in some cases be slow and possibly also undesirable. The default limit is 1000 rows which allows fast execution; a different limit can be specified as a parameter to the dump command.

3.2 Generator

Defined operations always return a boolean flag to denote if they are successful to update the current view. Generator exploit this feature so it randomly generates a dimension and chooses an operation to execute on the dimension. Then, according to operation, it further generates some values. For example, for dice on year zoom level of Time dimension, two year values are generated. After parameter randomization, operation is tried, if it is a valid operation, it is done otherwise, new operation and parameters are generated.

4. Experiment Results[1]

4.1. DataCube Builder

To build the data-cube, we use the command "./run.sh Main build".

Scale factor	Initial Size	Cube Size	Building time, s (datacube building + distribute product)
0.01	11.49 MB	12.3 MB	2s + 13s=**15s**
0.05	56.83 MB 432K	34.55 MB 1.8M rows	8.5s +51s = **59.5s**
0.1	113.84 MB 866K rows	55.11 MB 3,3M rows	17s +119s =**136s**
0.5	568.53 MB 4,3M rows	272.94 MB 15M rows	101s + 751s= **852s (14min)**
1	1137.02 MB 8.6M rows	534.53 MB 29M rows	256s + 1545s = **1801s (30min)**
5	5,68 GB 43M rows	2GB for cube + >500MB product replicas[2] >140M rows	1200s[3] + ~7,417s **> 2.3h**

[1] Experiments were run on only 7 mysql nodes as 1 server became unreachable through ssh (though mysql was still working). This was achieved with a small rewrite to our partitioning scripts and the dbgen source code.

[2] Product distribution (distributed partial replication) has partially failed with OutOfMemory exception on Java heap space. This is not too surprising knowing the fact that at the moment instead of complete distribution with N workers, we use one node running N^2 or N Threads (as an option) doing the shuffling. The system could be quite easily adapted to work in a completely distributed fashion through ssh. We tested executing remote jobs via SSH in Java: see the SSHJTest class for details. This required manually specifying host fingerprints since the version of SSH on the servers does not appear to maintain a known_hosts file.

[3] DataCube building completed without issues in 1,200s, but subsequent product distribution partially failed.

4.2. Online Query Processing

We evaluate our OLAP system using a simple walk as follows: Data cube on customer location, time and product => total volume; starting view: sales group by nothing (total sales);
sample walk: drill down into product, drill down into time, dice 1992 to 1996, drill down time to months, slice Jan 1993, drill down into location, roll up product.
Here are commands to execute the above walk:

1. drilldown product
2. drilldown time
3. dice time 1992 1996
4. drilldown time (this view is not materialized in datacube!)
5. slice time 199301 (this view is not materialized in datacube!)
6. drilldown customer_location (this view is not materialized in datacube!)
7. rollup product

(The test was executed by "./run.sh olap_datacube.CubeOperations" command)

Scale factor	Running time
0.05	1.89s
0.1	3.5s
0.5	30.7s
1	72.6s
5	1,181s (19 minutes)[4]

[4] A minority (~5%) of product titles were missing due to an out-of-memory exception in the Product replication phase (see the comments in earlier in this section). We are happy however that the queries ran successfully even with incomplete data.

5. Remarks

To minimise the size of the data cube, we initially tried storing only product IDs in the cube, and doing a join against the product table to find names when returning results. This turned out not to be a good idea. The space saved by only storing IDs was not very significant. In contrast, the extra time taken to process the query was in some cases enormous: 200s when doing the extra join verse 2s otherwise. The main reason for this difference is that since most tuples (particularly those in LINEITEM) are distributed by customer key, a large amount of data (matching items from the PART relation) must be copied between machines.

Our bloom-join operation has also been optimised since the second stage of the project. In that stage, we generated a hash filter on each node within MYSQL as a sequence of zeros and ones — in text, not binary, hence each bit took up eight bits of space. We then took the binary OR of all these filters before sending all tuples matching any of these bits to all nodes. In our current implementation we have optimised this in two ways: firstly, we generate the bloom-filter on each node and apply the these boom-filters on each pair of nodes individually, which allows the filter to be returned as a compressed binary sequence, saving a lot of space. Secondly, only the tuples matching each node's filter are sent to that node, further reducing data transmission.

Using only one Java process with N threads, our system has demonstrated an ability to deal with at least 5 GB of data (with minor memory usage issues only), or at least 1 GB without any issues.

As a further improvement one could implement the last data-cube building stage of replicating the data with bloom-joins (product titles) in a completely distributed way with multiple worker nodes, however this is not so important then this is now done during cube building phase.

A. Usage Guide

Compiling

`./compile_and_publish.sh`: This will compile the code, generate a jar and scp it to the server (team7@icdatasrv2).

Alternatively the code can be compiled as follows:

```
cd ./dpch_team7/src/
javac -classpath "../../lib/mysql-connector-java-5.1.19-bin.jar:../../lib/bcprov-
jdk15on-147.jar:../../lib/slf4j-api-1.6.4.jar:../../lib/bcpkix-jdk15on-147.jar:../../
lib/slf4j/slf4j-simple-1.6.4.jar:../../lib/jzlib.jar:../../lib/sshj.jar:." -d
buildcube_classes */*/*.java */*.java Main.java
jar -cvmf META-INF/MANIFEST.MF buildcube.jar -C buildcube_classes/ .
scp ./buildcube.jar <user>@<server>:/path/to/jar/buildcube_test.jar
cd -
cp ./dpch_team7/src/buildcube.jar ./buildcube_test.jar
```

Usage

```
./run.sh Main <arg>
(or java -cp "lib/...:...:buildcube_test.jar" Main <arg>)
```

Where <arg> can be:
build: Build the data cube.
browser: Browse the data cube.
operations: Browse the data cube with a demo walk.

In the *browser* mode, the user has the following available commands:
drilldown [dimension]: Drill-down on the given dimension.
rollup [dimension]: Rollup on the given dimension
slice [dimension] [arg1]: Apply slice on the given dimension with the filter arg1
dice [dimension] [arg1] [arg2]: Apply dice on the given dimension with the limits arg1 <= value <= arg2
limit [num]: Set maximum number of result to display
dump [file] [num]: Dump results of the last query to disk, with some upper limit (default 1000)
undo: Undo and switch back to the previous view.
exit: Exit

The available dimensions are:
customer_location: Location of the customer (two levels: region and nation)
time: Time at which the order was placed (three levels: year, yearmonth, date)
product: Name of the product (part) (one level: product_id)

If an invalid operation is performed (e.g. drilldown on time when the current view of time is date), the program will respond with a *Unable to perform operation* message.

Data Distribution

./populate.sh <scale-factor>

*Notes: Press Enter after eight “success” are displayed

Server Health Monitoring (MySQL, SSH)

./monitor_mysql_health.sh - shows status of all mysql servers + if ssh is not dead