CS346 Redbase Part 5 - EX Component Proposal
Distributed RedBase

Aditya Bhandari (adityasb)

1 Idea

The high-level idea is to make a **Distributed RedBase** by horizontal fragmentation of data across multiple nodes. The fragmentation of data and handling of queries will be done by a centralized control or *"master"* node, whereas all the data will reside on children or *"data"* nodes. All the user commands and queries will be given to the *master* node, who will then redirect the command or query to the appropriate *data* node.

This proposal document tries to list the various goals of the project extension, followed by a rough idea of the proposed system design and implementation details, finally followed by a set of milestones in order to track the progress of the project.

1.1 Goals

The primary goal of the extension is to create a distributed database system, consisting of data fragmented and stored across multiple nodes. The user should be able to perform the required queries as if the data is stored in a single node database. Underneath, the query will be distributed to the appropriate nodes and the results combined at the *master* node.

The aim is to be able to support all the queries from the `QL` component of the project in the distributed scenario as well - `INSERT, DELETE, UPDATE` and `SELECT`, in addition to the other RQL commands from the `SM` component - `create/destroy table, create/destroy index, load table` and so on. The user should be able to specify the distributed nature of the database through the `dbcreate` command, and specify the partitioning while creating a relation in the database.

1.2 Functionality

As mentioned in the previous subsection, the main functionality will be the option for the user to create a distributed database when using the `dbcreate` command - specifying the required number of *data* nodes. In the database, the RQL command to `create table` will enable the user to specify the partitioning for that relation across the *data* nodes. Similarly, the other RQL commands on the database - `help, print` should work as before even in the distributed scenario.

The functionality goal is to support all the basic RQL queries on a distributed database. The `INSERT` command should insert the required tuple in the corresponding *data* node, based on the partitioning vector. The `DELETE` command should delete the tuple from the corresponding *data* node, whereas the `UPDATE` command should perform the update at the corresponding *data* node. The `SELECT` command is the most challenging, which will perform a distributed join of the required relations by moving data temporarily across the *data* nodes.

(The movement of the data across the nodes, the communication of the *master* node with the *data* nodes and the *data* nodes among themselves will be handled by a newly added special communication layer underneath the RedBase nodes.)

2 System Design

The high-level design of the Distributed RedBase system is shown in the following figure and explained in the following two subsections.

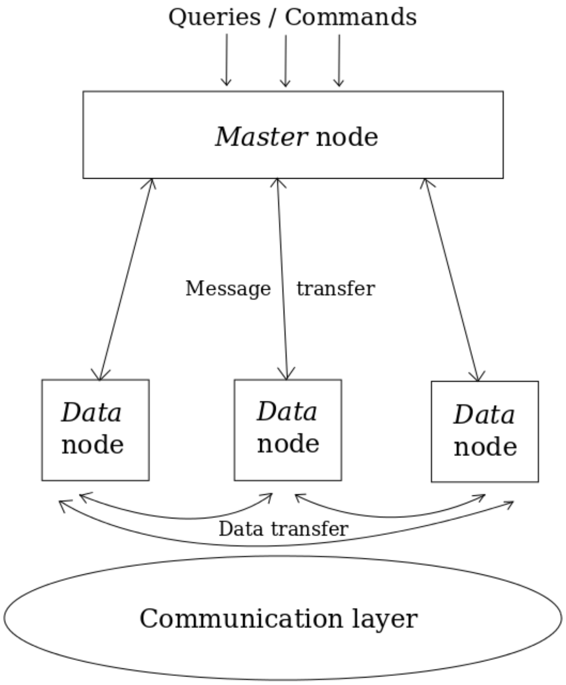

Figure 1: Distributed RedBase System

2.1 Distributed database - *Master* node and *Data* nodes

As explained in the previous sections, the RedBase system will consist of a *master* node to handle and control all the features and aspects of the distributed database. On the other hand, the actual data in the database will be stored in children *data* nodes, which are created by the *master* node when a distributed database is created, according to the requirements specified by the user.

When the *master* node creates the *data* nodes, it stores the partitioning information for each relation in the database in a **partitioning vector**. This vector basically represents the attribute value ranges for the fragmentation of the relation among the different nodes. We will only consider a range partitioning scheme over any one of the attributes in the relation (for simplicity). One such vector will need to be stored in the *master* node for every distributed relation in the database.

All the RQL queries and commands to the system are input to the *master* node. It then allocates the query to the appropriate node based on the partitioning vector for the corresponding relation. In the case of a local query - `INSERT, DELETE, UPDATE`, the *data* node performs the query and returns the result to the *master* node. But, in the case of a distributed query, that is, a `SELECT` requiring a distributed join, an entire relation will be transported to each of the *data* nodes, which will then perform a join with the local fragment of the other relation and return the result set to the *master* node. The *master* node will then evaluate the `UNION` of these result sets and get the final result of the query.

2.2 Network simulation - Communication layer

The function of the communication layer is to enable two things - message transfer between the *master* node and the *data* nodes and data transfer among the *data* nodes. This network communication between nodes will be simulated in a very näive way, since it is not the main part of the extension, but more of a supplementary feature for the distributed database.

The data transfer between the *data* nodes can be done through simple files. If node A needs to transfer a relation to node B, node A can write a file containing the relation data into the subdirectory for node B. Node B can then read the data from the file, perform the necessary query and then just delete the temporary data file that node A had written.

At first thought, the message transfer between the nodes can be done by just using `SM` and `QL` component methods. The details on this are still unclear and might become more clear according to the needs faced while implementing the required features of Distributed RedBase.

3 Implementation Overview

The implementation of the functionality and design of the proposed distributed extension as described in the previous sections will mainly involve extending the `SM` and `QL` components of RedBase. Also, it might require tweaking some features of the lower components (`PF, RM, IX`) as well. In addition to these, the communication layer will need to be implemented for the network simulation and data transfer between the nodes of the distributed database.

3.1 `SM` Component extension

The following features (commands) from the `SM` component will need to be extended:

- Create a new database: `dbcreate dbName` <*optional* `-distributed n`>

  The `dbcreate` command will have additional optional parameters to specify that the database should be distributed among `n` nodes, with the value of `n` specified by the user. This will create a *master* node which will store the `relcat` and `attrcat` relations, and `n` *data* nodes for storing the distributed relations.

- Create a new relation: `create table relName(attrName1 Type1, ..., attrNameN TypeN)` <*optional* `distribute attrName (value1, ..., value(n-1))`>

  The `create table` command will have additional optional parameters to specify whether the relation should be distributed. If yes, the attribute to be used for range partitioning should be specified along with the values for the range partitioning vector.

- Print an entire relation: `print relName`

  The `print` command will print out the entire relation. If the relation is distributed, the `print` command will be passed on to each of the *data* nodes from the *master* node, each of which will then perform the query locally.

- Metadata management: `relcat` relation

  The `relcat` relation will need to store whether a relation is distributed and if yes, it will store the attribute used for range partitioning in addition to the range partitioning vector and node mapping for these values.

3.2 QL Component extension

Essentially, the entire implementation of the QL component will need to be extended to support the queries on a distributed database. A rough idea of the extension is as follows:

- INSERT query:

  The INSERT query will be passed by the *master* node to the appropriate *data* node, based on the partitioning vector for the relation. The *data* node will then execute the query locally in its database.

- DELETE query:

  The DELETE query, similar to the INSERT query, will be passed to the corresponding *data* node by the *master* node, based on the partitioning vector for the relation. The query is then executed locally at the *data* node.

- UPDATE query:

  The UPDATE query will be similar to the DELETE query above.

- SELECT query:

  The SELECT query, if it requires the join of two distributed relations will need to transfer one of the relations to each *data* node using the communication layer described below. The join of a relation with the local part of the other relation can be computed using the join previously implemented in the QL component. The overall distributed join will be computed by combining (UNION of) the local result sets at the *data* nodes.

3.3 Network / Communication layer

The implementation of the communication layer might be embedded in the SM and QL component extensions mentioned above. For example, the transfer of data between the *data* nodes during a distributed join can be added using additional operators in the query plan generated for the SELECT query using the QL component. The message transfer between the nodes can be done by appropriate SM method calls whenever they are needed. (It is difficult to visualize and hence unclear right now about what additional implementation this will need to correctly supplement the required features.)

4 Milestones

For the purpose of tracking progress, it makes sense to divide the EX component of the project into three milestones, each progressively more challenging than the previous one. The goals of each of these milestones are listed and explained in brief below.

4.1 Basic - Distributed database with multiple nodes and whole relation scans

- dbcreate command to create a distributed database with multiple *data* nodes
- Fragment data horizontally across the nodes (simple user-specified range partitioning)
- Communication layer for the interaction among the *data* nodes and with the *master* node
- Scan the whole relation for the print command (by UNION of the fragmented result sets)

4.2 Intermediate - `INSERT, UPDATE` and `DELETE`

- Issue the query from the *master* node to the appropriate *data* node (based on the partitioning vector for the corresponding relation)

- Execute the query locally at the *data* node and result returned to the *master* node

4.3 Ambitious - `SELECT` (Distributed join)

- Copy an entire relation temporarily at all nodes using the communication layer (Add new operators in the query plan for this step)

- Perform join locally at each *data* node and return the result set to the *master* node

- Evaluate the final result at the *master* by evaluating the `UNION` of the result sets