CS 386D

Project #4
November 11, 2008

Cabrera, Guillermo
Pendyala, Swati

Mini-DB – Part 4

- Executive Summary

  For this project the main task involved adding functionality for the INDEX statement as well as making use of such indexes in order to optimize queries.

  Several changes have been made in the architecture of our application in order to provide a scalable framework that can be extended to include other optimizations. The major changes include:

  - **QueryPlan entity:** This new class encapsulates all of the information pertaining a query (ex. join predicates, relations, projected attributes, cost).
  - **In memory metaTable DB:** A hash map copy of the metaTable DB (schema information for databases) stored in BDB, this reduces I/O activity and improves speed while checking table schemas.
  - **Improved SELECT algorithm**: several modifications on the way information was saved from the AST, as well as computing temporary joins in memory as opposed to using BDB temporary databases (which we used in last project).
  - **Record structure on disk**: in order to save disk space while storing records we have removed redundant information. Previously we would store the type associated with each value, now, we only store only the values as we can easily check the types with out in-memory metaTable DB.

  Other minor changes have also been applied to code to account for performance; these are mentioned in the following sections. In order to test our application for performance we have used a combination of our scripts and the ones made available in the class website that include cases for one thousand and ten thousand records. Here is a summary of execution times for some queries, to view the details please look at “Sample Runs” section in this document.

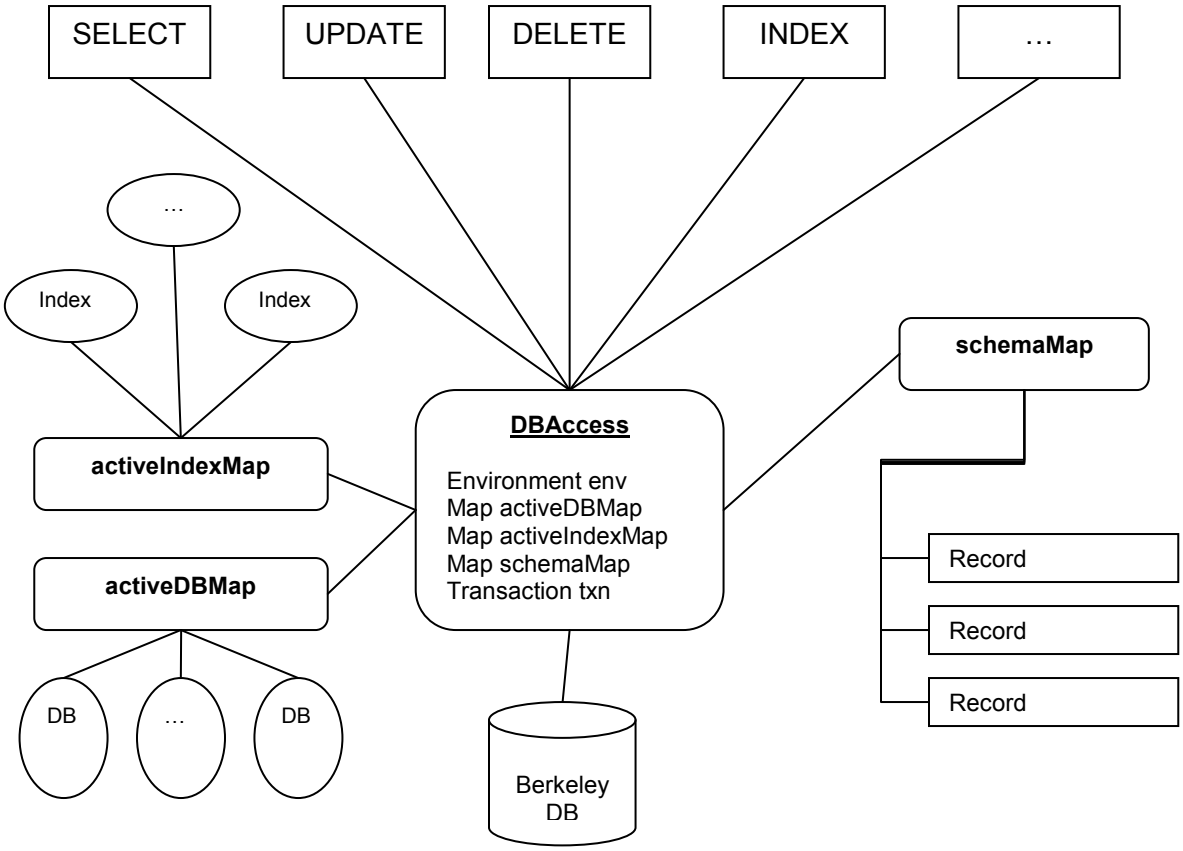
Avg Time*	**1000 Records**	**10 000 Records**
Join on 3 relations No index	0.1686 sec	1.25 sec
Join on 3 relations With index	0.0812 sec	0.7564 sec
Join on 3 relations With 2 indexes per relation	0.034 sec	0.3684 sec

*Average time is specified as the first time the query is run, BDB Databases need to be open, once open, our system keeps a reference to these databases in memory, thus subsequent queries will run faster. This difference time is heavily influenced by how Java creates objects.

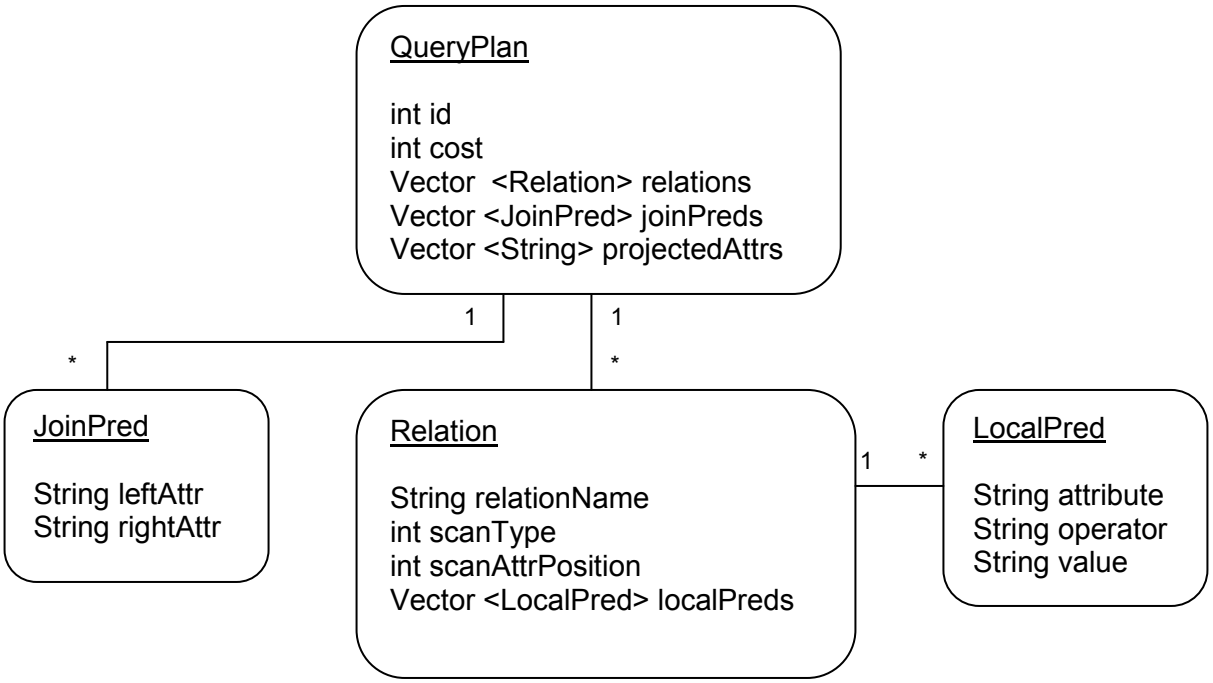
- Application architecture

  Similar to last part of project we have a main singleton class (DBAccess.java) which handles all interactions with Berkeley DB (BDB). This class provides a single BDB Environment for all transactions in a session. In addition, four key members of this class are used to ease transaction processing for DDL and DML commands:

  - **Map *activeDBMap***: A directory for all the opened databases in the environment. As per the recommendation from Berkeley DB documentation it is better to open databases and have multiple databases available for processing, and at a later time closing all of them.
  - **Map *activeIndexMap***: A directory of SecondaryDatabase objects, these are crated by the user using the INDEX command and this map, is used by our IndexScan class to retrieve an index when it is needed for a scan.
  - **Map *schemaMap***: A copy of the metaTable Database; it contains schema information (ex. attribute name, type, indexed) for each relation stored. Having this in memory greatly improves performance, and schema information is constantly used. By having an in memory structure we avoid I/O with the stored database. We only read once from disk when the environment is open, and then update whenever a new table or index is created.
  - **Transaction *txn***: One transaction is available at a time, different commands needing a transaction are given a reference to this single transaction, once it is committed or aborted a new transaction will be created.



The previous diagram constitutes a high level overview of the architecture in our application; it shows the main elements described earlier and their interactions with other classes in the application. Moreover, in order to accommodate the new algorithm for SELECT and improve on performance from last project, we have designed the following framework to process queries:



Having a single entity (QueryPlan) to work with eases processing, and given that we have logically divided it into its corresponding parts (ex. relations, local predicates, join predicates) we are able to have a map of the required parts needed in the query. Key ideas from the above diagram include:

- **QueryPlan *cost***: This member variable is individual to each QueryPlan, it is modified by rules that give penalty points relations involved in a query. For a join involving three relations, six plans will be generated, each with a different combination of **join order**. The penalty points for a relation will vary based on the position it has in the join order for each QueryPlan. For instance, in the case of a three relation join, a relation involving a cross product will have a high penalty value; joining this relation at the beginning will be expensive as we will produce a great number of tuples, whereas, if it is computed at the end, we might have less tuples to join it with if the previous joins were filtered.
- **Relation *scanType***: Flag used in SELECT command to determine whether a relation will be scanned using an Index or if we will be doing a sequential scan.

The logical model presented above permits the extension of functionality, given that we now have a cost associated with each plan, this cost can be affected by as many computations desired, in our application we have used a rule-based system that gives “penalty points” to non optimal cases (ex. not having filtering predicates), but we have contemplated further additions such as selectivity calculation that could also be taken into account.

- Indexing module

  User inputs the attribute to be indexed as Relation.Attribute. The main algorithm for creating an index in BDB is:

  1. Parse input from query and check if relation/attribute exist
  2. Find attribute in metaTable DB (schema information) and check if item is already indexed.
     2.1. If true, then issue error and exit
  3. Create index by creating a secondaryDatabase in BDB
  4. Update values in metaTable DB (in memory and in disk)

  Given that indexes are represented as SecondaryDatabases in BDB, we have taken the same approach as with PrimaryDatabases, thus, indexes are handled in the same way. There is an **activeIndexMap** (Hash map) containing objects of type SecondaryDatabase of all the indexes in the environment. Whenever a class such as IndexScan needs to retrieve an index, it can get a reference to the in memory index via methods in or main DBAccess class.

  The record structure for our metaTable DB has now changed in order to accommodate for information regarding indexes. A record now has the following format:

Value	Type	Index Flag	Value	Type	Index Flag	…

  Classes added/modified:

- Select module

  The implementation in the last project did not focus on performance of joins; also, initial error checking and the ability to perform cross products were not available. These items were main focus for this final part, as performance was the focus in this last stage, thus, several changes were made, including the earlier mentioned QueryPlan framework, algorithm and implementation details specific to java.

  - Algorithm

    1. Parse query and error check
    2. Enumerate access plans
    3. Compute cost for each plan and choose the one with the minimum cost
    4. Set the scan method for relations involved in the plan.
    5. Execute plan
       5.1. For each relation in plan
            5.1.1. Perform an Index or Sequential scan
       5.2. For each relation in plan
            5.2.1. Join with the next relation in plan (following specified order)
       5.3. Project the final join result based on query specifications

    1 – Parse and check

One of the major improvements from last time was collecting all the information up front, this made it easier to do error checking and avoid further processing if we detected an error. We are retrieving all the information from the AST starting from the SELECT, then FROM and finally the WHERE clause if one exists. Given that our application stores attribute names **in the form "Relation.Attribute"** to uniquely identity each attribute when joining, we are making sure we retrieve elements taking this into consideration, thus, when we populate the initial QueryPlan with information, this is taken into account. For this first step in the algorithm we are checking for the following errors:

- Check if relations in FROM clause exist
- For SELECTs involving joins, check if attributes are in the form "Relation.Name" to avoid ambiguity.
- If join predicates exist, check if they are in the form "Relation.Name" to avoid ambiguity.
- Check if all attributes in WHERE clause exist.

2 – Enumerate

As for plan enumeration we have focused on doing so only for queries involving **up to three relations**, queries involving more relations will not be optimized and their joins will be processed in the order the are specified in the FROM clause. To enumerate plans we have parted from the assumption that a query execution plan determines the order on how relations will be accessed. Thus, for joins of two or three relations, given the order the user has specified in the query, we are rearranging (swapping) the order of relations to all possible combinations. For instance:

SELECT * from A, B, C WHERE …

For this query involving 3 relations, we will list the 6
combinations in which we can access A, B and C:
ABC, BAC, BCA, CBA, CAB, ACB, ABC.
There will be a total of 6 plans for which we will later calculate cost

3 – Compute cost

The cost for every plan is computed based on a **rule-based penalty point system**, for this step we first check every relation involved in a query and based on the following rules, a penalty point is added to a relation does not satisfy the rule:

- Are there local predicates associated with a relation?
- If predicates exist for a relation, are any of the predicate attributes indexed?
- Is the relation specified in any of the join predicates? (cross product case)
- For joins involving 3 relations, is the first relation to be joined included in any of the join predicates? (Leave cross product of this relation at the end).

Additional rules can be added that will give a relation a higher penalty value. Once penalties have been computed, the cost for the plan is computed in the following way:

1. For each relation in plan
   1.1. Multiply the penalty value for the relation with the value of its position in the join order.
   1.2. Add this result to the overall cost.'

This **positional method** of assigning penalties helps us select a QueryPlan that includes a relation with a less penalty value as the first relation to scan and join. For example, for a query specifying relations AB and C, if B has a local predicate and one of its attributes is indexed, it is likely our system will chose a QueryPlan containing order BAC or BCA. After the plan with the minimum cost has been selected from the list of available plans, all further operations will refer to this "master plan".

4 – Set scan type

For each relation in the QueryPlan, we check if there are any attributes in the local predicates that have an index and that the operator for that predicate is an "=" operator. If so, we set the flag in the Relation object containing such index, so that we can do an **index scan to retrive the relation** instead of a sequential scan.

5.1 - Scan

For every relation in the master QueryPlan we check if have to do an index or sequential scan. Two new classes have been created (query.IndexScan and query.NormalScan), whose purpose is to traverse the relation on disk, apply any local predicates to that relation and then create a Vector of items remaining. This was another major improvement from last project, as we are now **filtering before joining**, instead of joining all relations and then filtering.

The output of either an index or normal scan is a Vector containing the tuples to be joined from every relation. In our SelectCmd class we will have a Collection (scannedRels) of these vectors that will be used to join. The two algorithms involved for each one are:

- Normal Scan Algorithm

  1. For each record in the database.
     1.1. For each local predicate
        1.1.1. Check if the attribute in the predicate matches to that of the existing predicates in the record
        1.1.2. Apply the local predicate to the record and qualify he record
     1.2. Check if more local predicates exist
        1.2.1. Go to Step 1.1 until all local predicates are applied to the record
     1.3. If the record is a qualified record, insert it into a temporary vector of records which holds all qualified records.

1.4. Check if more records exist in database.
1.4.1. Go to 1 until all records are evaluated.
2. Return the vector of records.

- Index Scan Algorithm

1. Get the attribute and value from the local predicate.
1.1. If the attribute in the local predicate is the first attribute of the database
1.1.1. Get record from the Primary Database with the matching value of the attribute in the local predicate.
1.1.2. Insert this record into the temporary vector of records which hold qualified records.
1.2. Else
1.2.1. For each record in the secondary DB of the attribute in the local predicates
1.2.1.1. Get record from the Primary Database with the matching value of the attribute in the local predicate.
1.2.1.2. Insert this record into the temporary vector of records which hold qualified records.
1.3. Check if more duplicate records exist in the secondary DB
1.3.1. Go to 1.2.1 until all duplicate records are evaluated in the secondary DB.
2. Return the vector of records.

5.2 – Join

The join algorithm we have used is the nested for loops. We are still only processing left-deep trees and not bushy trees. We are doing this in the following way:

1. For every relation in the master QueryPlan
1.1. Get name of first two relations to join
1.2. Select the join predicate to use
1.3. For each record in outer relation
1.3.1. For each record in inner relation
1.3.1.1. If qualified tuple based on join predicate
1.3.1.1.1. Create new record in newJoinRelation
1.4. Create entry of newJoinRelation in metaTable DB

In our join algorithm we are also checking if this join will be a **cross product** before step b; we know that if we have two relations in the FROM clause and there are no join predicates, this involves a cross product. We also have the other case where we might have join predicates involved but the relation is not part of the join predicate.

In order to handle **joins of multiple relations**, our algorithm joins the first two mentioned tables and creates a temporary relation; this relation is then joined with the third and then produces another temporary relation, and so on. The metaTable DB will have the schema information for the final result table, and it will let us project all the attributes in the FROM clause with ease.

5.3 – Project

The first thing we check in this step is if we have a query involving one or more than one relations. This is important as for two or more relations **our application requires all attributes to be specified in the form "Relation.attribute".** This is enforced to avoid ambiguity as to which attribute was specified. If the user specified "*" we simply project all items in the result relation, otherwise, we map the attributes specified in the FROM clause to those in the schema of the result relation and print them.

Finally, we display the number of records that were printed from the query, the time it took to process the query and if an index scan was used to retrieve a relation from disk.

- Highlights and issues

Query execution time reduced: Whenever we open an environment, the first query involving a SELECT statement will normally take a greater amount of time than subsequent queries given on the same relations. This is the main reason why we have included the average time in computing performance; we have come to the conclusion that this event can be seen because of Java mechanisms in creating objects. For further execution of commands certain objects with still be referenced thus will not need to be created again. The following image shows a snapshot of this behavior, where a query has been executed three times, showing different execution times:

```
mdb> select item_master.item_name,customers.c_name,item_sale.item_sold from customers,item_sale,ite
 > .

|item_master.item_name              |customers.c_name                  |item_sale.item_sold
|"fusoegjatxpittsmduejjmmec"        |"elcku"                           |1
|"fbyhkarw"                         |"elcku"                           |11
|"yhwbyxmzubhyvgzcuosihaoutu"       |"elcku"                           |9
|"exjgjgmqbucbwvkmwrdonbtx"         |"elcku"                           |20
|"eggjuajqxnhilpdac"                |"elcku"                           |8
(5) Record(s) found in: 0.485 seconds.
 Using index on:item_sale

mdb> select item_master.item_name,customers.c_name,item_sale.item_sold from customers,item_sale,ite
 > .

|item_master.item_name              |customers.c_name                  |item_sale.item_sold
|"fusoegjatxpittsmduejjmmec"        |"elcku"                           |1
|"fbyhkarw"                         |"elcku"                           |11
|"yhwbyxmzubhyvgzcuosihaoutu"       |"elcku"                           |9
|"exjgjgmqbucbwvkmwrdonbtx"         |"elcku"                           |20
|"eggjuajqxnhilpdac"                |"elcku"                           |8
(5) Record(s) found in: 0.094 seconds.
 Using index on:item_sale

mdb> select item_master.item_name,customers.c_name,item_sale.item_sold from customers,item_sale,ite
 > .

|item_master.item_name              |customers.c_name                  |item_sale.item_sold
|"fusoegjatxpittsmduejjmmec"        |"elcku"                           |1
|"fbyhkarw"                         |"elcku"                           |11
|"yhwbyxmzubhyvgzcuosihaoutu"       |"elcku"                           |9
|"exjgjgmqbucbwvkmwrdonbtx"         |"elcku"                           |20
|"eggjuajqxnhilpdac"                |"elcku"                           |8
(5) Record(s) found in: 0.094 seconds.
```

- Improvements from last project submission

  As previously mentioned, several items were addressed since last project that had to be fixed in order to obtain better performance. The following is a list of some of the main items that were added or changed which have had the greatest impact in performance for this last project:

  - **Removal of temporary databases**: Even though BDB temporary database resided in memory, there was still a great time spent in doing TupleBinding our class so that it could be stored as a record in a database. We have replaced this with just Vector structures for joins. Thus, a record is not represented as a Vector and this Vector is stored in another Vector which holds a set of records. This, combined with the other changes proved to be critical in drastically reducing time to compute joins. A join involving 3 relations on 1000 records would take close to two minutes in the last project, that same query takes close to two seconds to complete.

  - **QueryPlan encapsulation**: this set of encapsulation classes helped give a unified logical view on operations to process on a query (ex. parsing, computing cost, scanning). It also provided flexibility in coding, as we were able to closely relate to other commercial DMBSs which also have a Query Plan or graph as their central unit of processing, and process any optimizations based on these plans.

  - **SELECT algorithm**: The algorithm mentioned above also had a great impact, by error checking and discarding any erroneous queries at start we could avoid further errors. Also, filtering relations after scan and then joining these filtered relations greatly drew a comparison from last project, as in last project we were scanning and joining all relations and later filtering a massive relation.

  - **File structure on disk**: Records in BDB are stored using a GenericTuple class, which is serialized and then stored in disk. This class is a logical representation of a record in a database; it contains a vector in which we store each attribute. Earlier we would store the value and also the type of the attribute, thus, 2 elements were needed per attribute. This redundant use of data was removed so that it is only the values we store, as we can now map things trivially to the relation schema via the in-memory metaTable DB.

  - **Indexes populated at start**: We check all the attributes available in tables that are not the primary key (first column) and we then open these indexes for them to be used in the session. Once opened they are added to our activeIndexMap which acts as a directory for other transactions that require the use of an index (ex. IndexScan class).

  - **Memory metaTable DB**: Another important change, since the schemas for tables are constantly checked, in previous projects we would have to retrieve a record from out metaTable database and then use that information. As of this project we only read once at start to load all schema information to a Hash map, and we later only apply updates whenever we issue an INDEX or CREATE command.

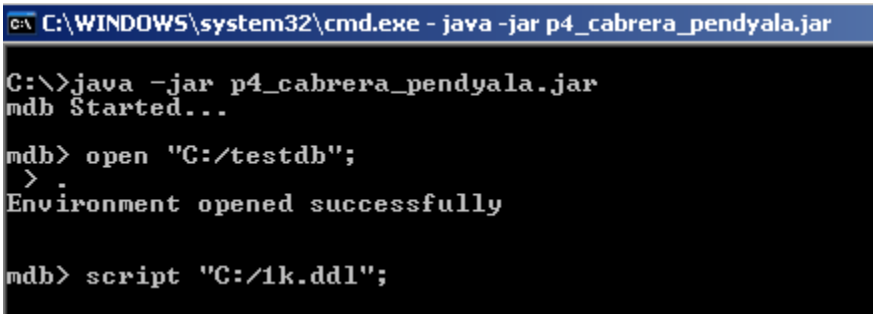
Sample run

The following screenshots are sample runs of the testcases made available on the class website, the same steps have been executed for each testcase:

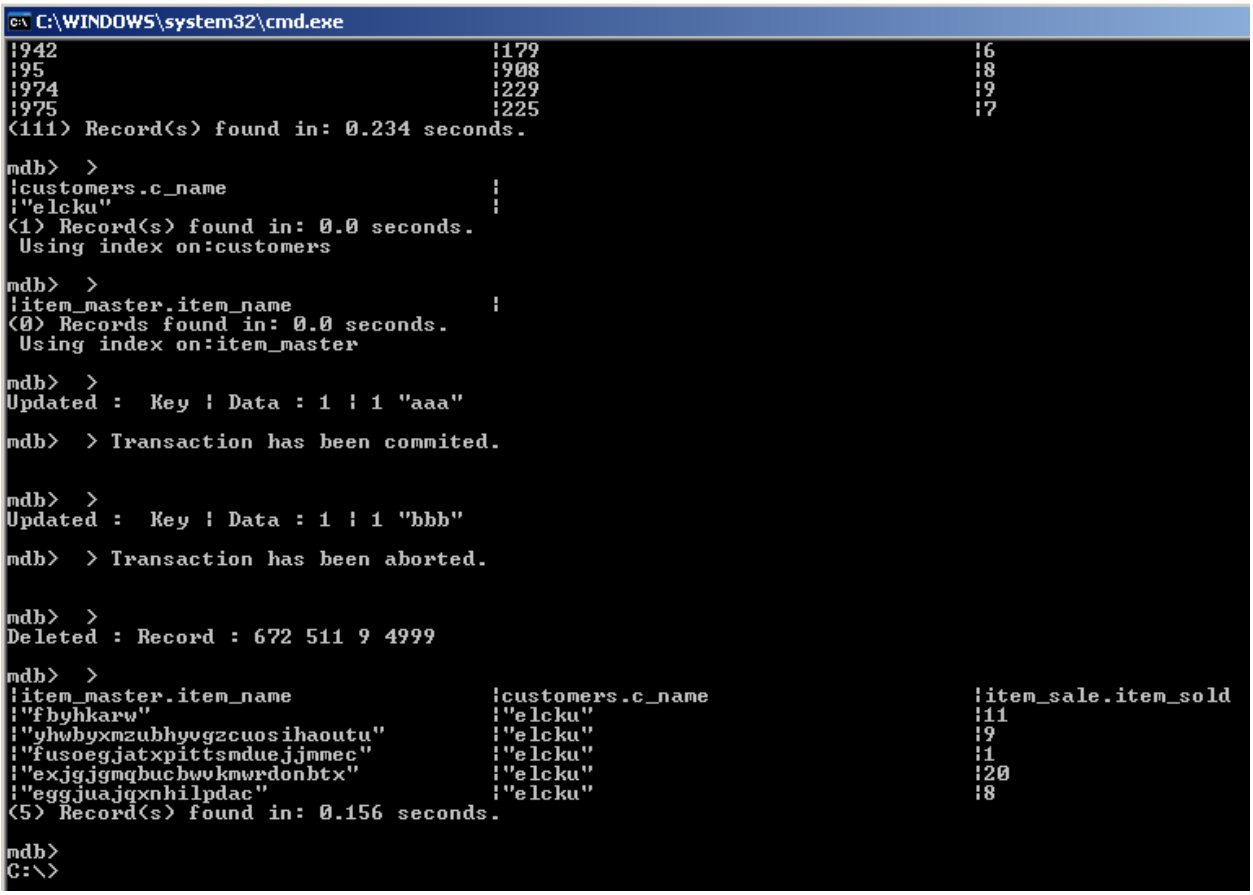
Script 2: 1k tuples: http://www.cs.utexas.edu/users/dsb/cs386d/Projects/test1.sql
Script 3: 10k tuples: http://www.cs.utexas.edu/users/dsb/cs386d/Projects/test2.sql

Script 2:

Open environment and execute script.

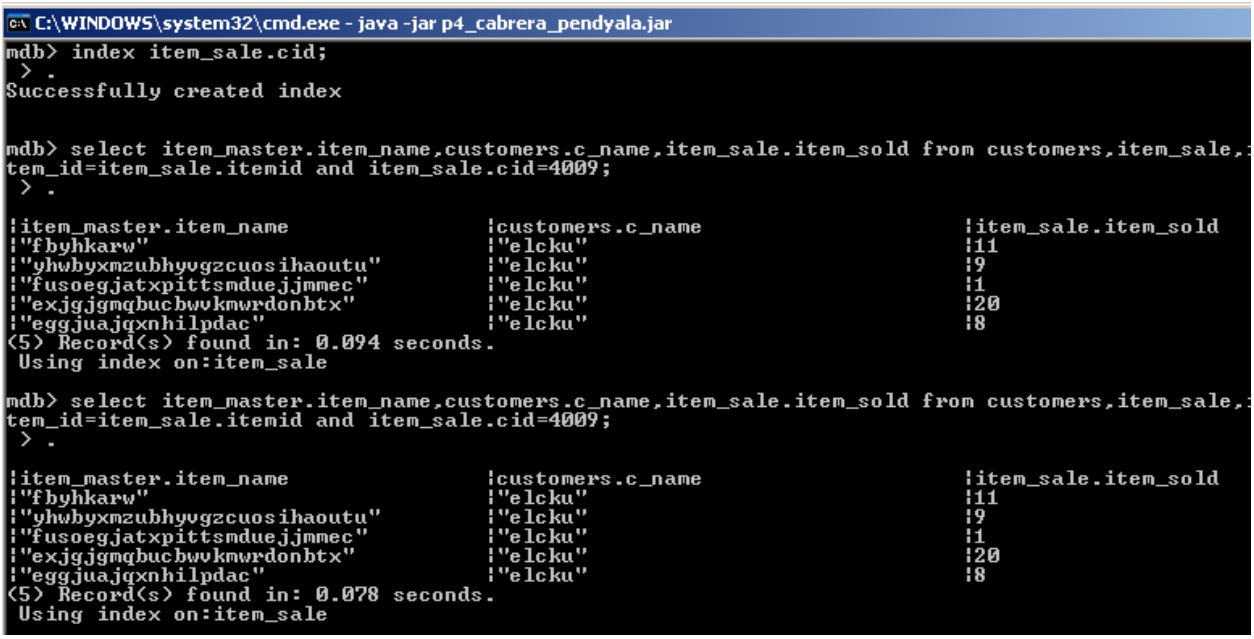


The output from the final UPDATE, DELETE, COMMIT and SELECT statements is the following:



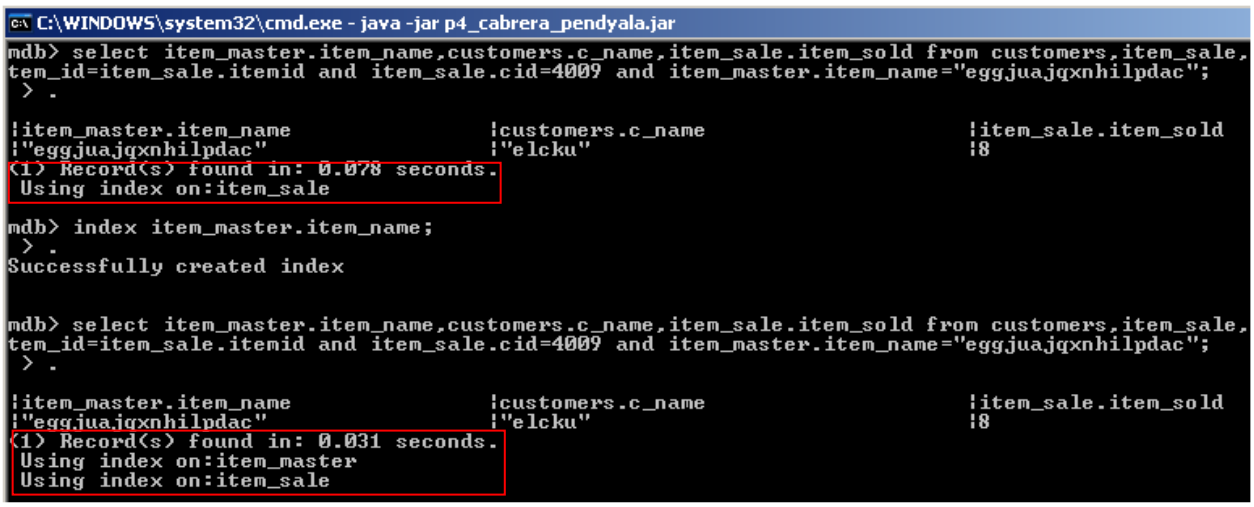
Using the same environment we have now added an INDEX to the column specified in the last query from the testcase:

```
SELECT item_master.item_name,customers.c_name,item_sale.item_sold
FROM customers,item_sale,item_master
WHERE customers.c_id=item_sale.cid and
item_master.item_id=item_sale.itemid and item_sale.cid=4009;
```

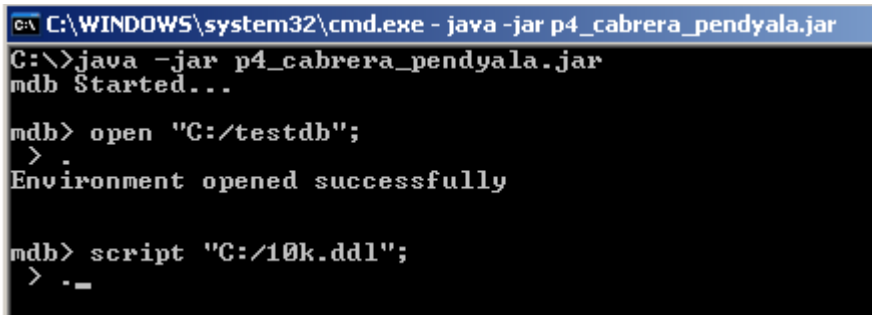


Let us now use two INDEXES on different relations to select an individual record from the set produced by the above query, using the following query:

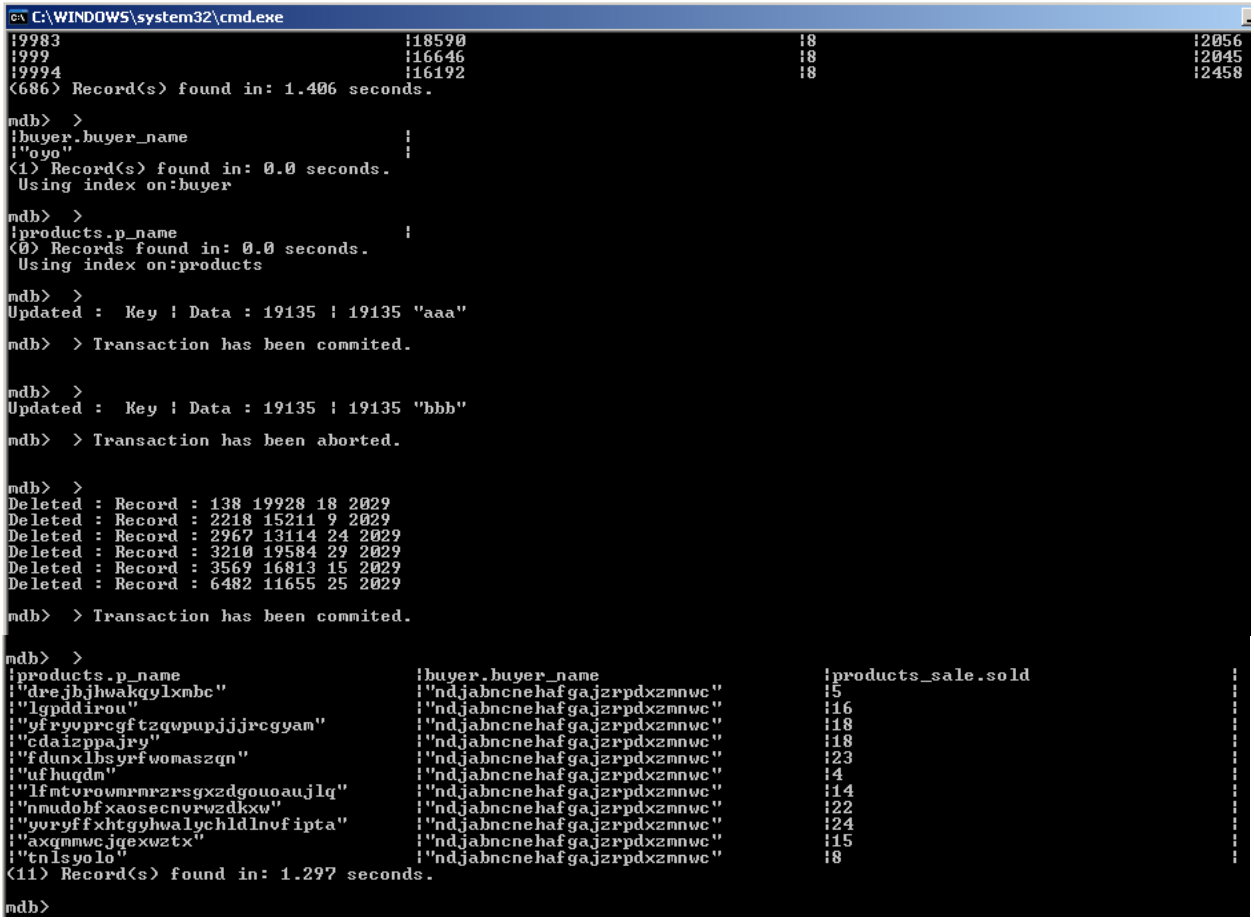
```
SELECT item_master.item_name,customers.c_name,item_sale.item_sold
FROM customers,item_sale,item_master
WHERE customers.c_id=item_sale.cid
AND item_master.item_id=item_sale.itemid
AND item_sale.cid=4009 AND item_master.item_name="eggjuajqxnhilpdac";
```



Script 3:



The output from the final UPDATE, DELETE, COMMIT and SELECT statements is the following:



Using the same environment we have now added an INDEX to the column specified in the last query from the testcase:

```
SELECT products.p_name,buyer.buyer_name,products_sale.sold
FROM products,products_sale,buyer
WHERE products.p_id=products_sale.pid and
buyer.buyer_id=products_sale.buyerid and products_sale.buyerid=2001;
```

```
C:\WINDOWS\system32\cmd.exe - java -jar p4_cabrera_pendyala.jar
mdb> index products_sale.buyerid;
 > .
Successfully created index


mdb> select products.p_name,buyer.buyer_name,products_sale.sold from products,products_sale,buyer where
cts_sale.buyerid and products_sale.buyerid=2001;
 > .

|products.p_name                    |buyer.buyer_name                   |products_sale.sold
|"drejbjhwakqylxmbc"                |"ndjabncnehafgajzrpdxzmnwc"        |5
|"lgpddirou"                        |"ndjabncnehafgajzrpdxzmnwc"        |16
|"yfryvprcgftzqwpupjjjrcgyam"       |"ndjabncnehafgajzrpdxzmnwc"        |18
|"cdaizppajry"                      |"ndjabncnehafgajzrpdxzmnwc"        |18
|"fdunxlbsyrfwomaszqn"              |"ndjabncnehafgajzrpdxzmnwc"        |23
|"ufhuqdm"                          |"ndjabncnehafgajzrpdxzmnwc"        |4
|"lfmtvrowmrmrzrsgxzdgouoaujlq"     |"ndjabncnehafgajzrpdxzmnwc"        |14
|"nmudobfxaosecnvrwzdkxw"           |"ndjabncnehafgajzrpdxzmnwc"        |22
|"yvryffxhtgyhwalychldlnvfipta"     |"ndjabncnehafgajzrpdxzmnwc"        |24
|"axqmmwcjqexwztx"                  |"ndjabncnehafgajzrpdxzmnwc"        |15
|"tnlsyolo"                         |"ndjabncnehafgajzrpdxzmnwc"        |8
(11) Record(s) found in: 0.75 seconds.
 Using index on:products_sale
```

Let us now use two INDEXES on different relations to select an individual record from the set produced by the above query, using the following query:

```
SELECT products.p_name,buyer.buyer_name,products_sale.sold
FROM products,products_sale,buyer
WHERE products.p_id=products_sale.pid and
buyer.buyer_id=products_sale.buyerid and
products_sale.buyerid=2001 and products.p_name="ufhuqdm";
```

```
mdb> select products.p_name,buyer.buyer_name,products_sale.sold from products,products_sale,buyer where
cts_sale.buyerid and products_sale.buyerid=2001 and products.p_name="ufhuqdm";
 > .

|products.p_name                    |buyer.buyer_name                   |products_sale.sold
|"ufhuqdm"                          |"ndjabncnehafgajzrpdxzmnwc"        |4
(1) Record(s) found in: 0.688 seconds.
 Using index on:products_sale

mdb> index products.p_name;
 > .
Successfully created index


mdb> select products.p_name,buyer.buyer_name,products_sale.sold from products,products_sale,buyer where
cts_sale.buyerid and products_sale.buyerid=2001 and products.p_name="ufhuqdm";
 > .

|products.p_name                    |buyer.buyer_name                   |products_sale.sold
|"ufhuqdm"                          |"ndjabncnehafgajzrpdxzmnwc"        |4
(1) Record(s) found in: 0.359 seconds.
 Using index on:products
 Using index on:products_sale
```

- Final Remarks

In this last phase of the project we have seen the importance of performance specially when dealing with large volumes of data, we were able to identify problem areas in our design and code. This enabled us to fix and improve the overall speed to process joins. For future work, we had thought of calculating selectivity for indexed attributes, we stopped at the planning phase with this idea, yet, we had all the required items to implement it, as our application enabled this change given the cost calculation framework.

For this project: Swati Pendayla worked on the indexing functionality and implemented the IndexScan and NormalScan classes. Guillermo Cabrera worked on the new algorithm and framework needed for the SELECT command.