14TH NOVEMBER 2018
20171401 - JAYITHA .C

DATABASE SYSTEMS

THEORY ASSIGNMENT 2

According to Codd, discuss the problem of access path dependencies in the existing systems and how relational model could overcome

Access Path Independence is a property that describes relatively loose coupling between how data is organized or managed and how data is accessed and processed. Access path dependence is when data is stored using pointers to locate data such as in a tree or map. In order to get to each entry a path of pointers must be followed. Application programs are built on top of the structure of the file system. If a given mini-world has to be represented by either network or hierarchical model, it could be represented in multiple different structures and hence our application programs cannot be built independent of these things.

In order to combat the issue of dependent and ad hoc databases Codd uses his mathematical knowledge of set theory to define a more abstract system. Edgar's system would be a relational system that stores data in tables. Each row in the table is a new entry and the column defines the type. This allows engineers to treat tables as a set.

In mathematics a set can be any collection of definite and distinguishable data. Each element in this set is called a member. A member can have, triple, quadruple, sextuple, septuple, n-tuple the number of values. In relational databases every table is a set, every row is a tuple, and columns define the order of data in that tuple. One of the key principles that carries over from set theory is that each member has to be unique. In order to comply with set theory RDBMS will require that each row has its own unique primary key. Every entry in a table/set is given a unique primary key. This allows each member/row/tuple to be unique.

By using the principles of set theory Codd was able to define a whole new type of database. This method is much more abstract, which means goodbye ad hoc method. The ad hoc databases of the past were programmed uniquely for a single purpose or application. If a programmer needed to change the structure of the database, even if that meant changing a floating number field to an integer field, a programmer would have update the source code to make the change. But with the set like representation of data we have eliminated the dependence on the structure. Concepts like views also help in dealing with ownership problems.

Why existing character oriented file system provided by operating system is not suitable for DBMS.

The file system provided by UNIX supports objects (files) which are character arrays of dynamically varying size. Below are 2 reasons as to why building a DBMS on top of this system doesn't work.

• *Physical contiguity*: A file is usually stored in blocks that are spread across the disk. So to access the whole files these blocks have to be read sequentially. This leads to a considerable read overheard due to disk arm movement.

• *Tree Structured File System*: The OS uses 2 tree structures to manage this distributed block structure of files, and a DBMS built on this uses another tree leading to a 3 tree structure and maintaining such a structure incurs some severe overhead

In conclusion, Character based file systems aren't a suitable base to build DBMS systems on instead it is possible to do the converse, i.e. build character arrays on top of records, the only issue is efficiency. So it is concluded that OS designers should provide DBMS facilities as lower level objects and character arrays as higher level ones.

As compared to typical DBMS engine, discuss the tradeoff in C-store approach

Assumption: Typical DBMS engine implies we are talking about a Row Store DBMS.

Comparisons between row-oriented and column-oriented databases are typically concerned with the efficiency of hard-disk access for a given workload, as seek time is incredibly long compared to the other bottlenecks in computers.

Record Inserts: In row store DBMS, it is easy to add records and there need not be any preprocessing or splitting up of the record, this application is especially useful for OLTP applications where updates and insertions are frequent whereas to insert into a column store DBMS the record has to be split up column wise and inserted accordingly, this overhead is incurred for every new record. Hence they are not suitable for OLTP applications.

Record Reads: When we want to read a full record, in row store we can easily get a record as the column values are stored together whereas in a column store DBMS we have to traverse to the particular record's column value for each attribute of the record. But at the same time, when we want the values of only a few attributes we tend to read unnecessary data in the case of row store which we don't do in column store and hence the kind of record read query is important.

Range queries on a particular attribute: When trying to perform comparisons on one attribute value, the row store case, we'll have to read or scan through every record in a sequential manner and then identify the value of the attribute and then perform the comparison whereas in column store systems we can directly access these values and hence make performing these range queries faster. To help row store systems we can implement indexes over these attributes but index accesses.

The above 2 points indicate that C-Store DBMS are more suited for OLAP kind of queries.

Compression: Column data is of uniform type; therefore, there are some opportunities for storage size optimizations available in column-oriented data that are not available in row-oriented data. For example, many popular modern compression schemes, such as LZW or run-length encoding, make use of the similarity of adjacent data to compress. Columnar compression achieves a reduction in disk space at the expense of efficiency of retrieval.

In conclusion, Columnar databases boost performance by reducing the amount of data that needs to be read from disk, both by efficiently compressing the similar columnar data and by reading only the data necessary to answer the query. There are other differences also in relation to query execution and data recovery.

Justify the following text "...Also, an I/O measure would have revealed a serious drawback of XRM: Storing the domains separately from the tuples causes many extra I/Os to be done in retrieving data values...."

XRM stores relations in the form of "tuples," each of which has a unique 32-bit "tuple identifier" (TID). Since a TID contains a page number, it is possible, given a TID, to fetch the associated tuple in one page reference. However, rather than actual data values, the tuple contains pointers to the "domains" where the actual data is stored.

Storing the domains separately from the tuples causes many extra I/Os to be done in retrieving data values because when we want the value for a tuple given the TID, the TID actually gives us access to the domains and not the values corresponding to the tuple itself. So not only do we have to access the TID we also have to access the domain mapping from the TID to the correct domain value. This I/O access could increase further if inversions are implemented, i.e. if instead of the domain containing domain values, they contained TIDs again of tuples that had the same domain value as the value of the domain of the current tuple we want, we would have to perform more I/Os to find out the domain value.

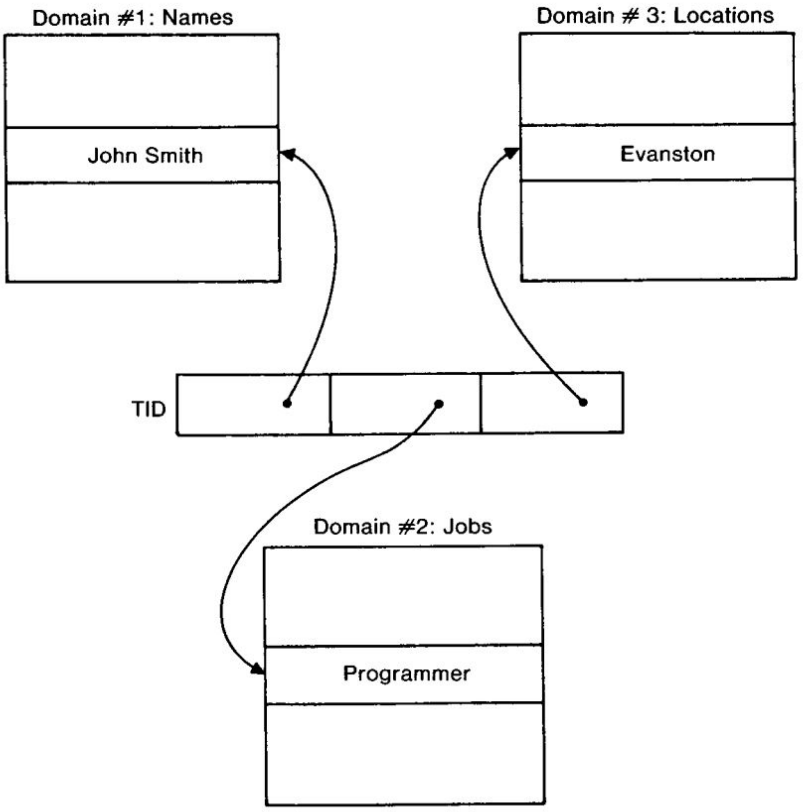

Fig. 2. XRM Storage Structure.

Justify the following text: "...In general, our experience showed that indexes could be used very efficiently in queries and transactions which access many records, but that hashing and links would have enhanced the performance of "canned transactions" which access only a few records...."

If several records are to be fetched using the index scan, the three start-up I/Os are relatively insignificant. However, if only one record is to be fetched, other access techniques might have provided a quicker path to the stored data.

Hashing was not used when retrieval of many tuples was needed because it does not have the convenient ordering property of a B-tree index. And direct links were not implemented because essential links were inconsistent with the UI of the Relational database and it was has to maintain the consistency of nonessential links upon updates.

But for canned transactions i.e. standard types of queries and updates which are frequently used by naive end users to constantly query and update database, when the kind of query is well defined, rather than traversing through the index structure of every needed attribute to retrieve values, it would incur less I/Os if we used hashing or direct links. i.e. suppose getting the employee name given the employee ID is a very common query, then hashing on the Emp ID would yield faster results than going through an index structure. But again if the number of records to be queried are large then the index structure overhead is less in comparison and indexing is preferred