Project 2 - Design and Implementation Summary

CS222: Principles of Data Management

Christopher A. Wood (woodc1 - 54168394)
Tamir Husain (thusain - 40206791)
February 10, 2014

Task

Write a report to briefly describe the design and implementation of your relation module.

Summary

In this document we outline the design and implementation of our record-based file system and relation manager modules. In particular, we describe the changes made to the RBFM layer to support enhanced features and performance efficiency for the RM layer. We also detail the design rationale and implementation strategy for the system catalog, as this same design is duplicated to handle all relations and their corresponding operations in the RM layer.

Modified RBFM Layer

After implementing the updateRecord and deleteRecordmethods in the RBFM layer, we then implemented the reorganizePage method to complete the remaining functionality in this layer. However, in an attempt to reduce the page-level work done in the relation module, we also designed a primitive page reorganization policy in the RBFM layer. The guiding principle for this policy is that after a certain size of “holes,” or contiguous unutilized space between adjacent records, accumulates due to invocations of the deleteRecord and updateRecord methods, we trigger a page reorganization. The gap size threshold, which we have fixed at 0.25*PAGE_SIZE = 1024 bytes, was determined empirically by analysis of the amount of disk space consumed during stress tests of the system. In order to support this page reorganization policy, the header of each page was amended to include the total gap size which is maintained internally within the RBFM record operations.

In addition to this page reorganization policy, we also amended the record format to include pointers to the offset of the end of the record so as to make size calculations much easier. Furthermore, in an effort to support future features such as adding attributes to fields, we included another new field that records the number of attributes persisted on disk.

Finally, to support the updateRecord and deleteRecord functionality, we modified the format of record index slots to include space for forward RID pointers (page and slot tuples) that can be used to find a record if it has been moved due to an update. With the new record index format, which altogether includes the record size, page offset, and (potentially unused) forwarding RID pointer, deleted records are indicated by null RID pointers and empty sizes. Similarly, moved

records (tombstones) are indicated by empty sizes and non-null RID pointers. The readRecord method was updated accordingly to follow an RID pointer when encountered to retrieve a record at its new location on disk.

The new record, page header, and page formats are shown below in Figure 1 for illustration purposes.

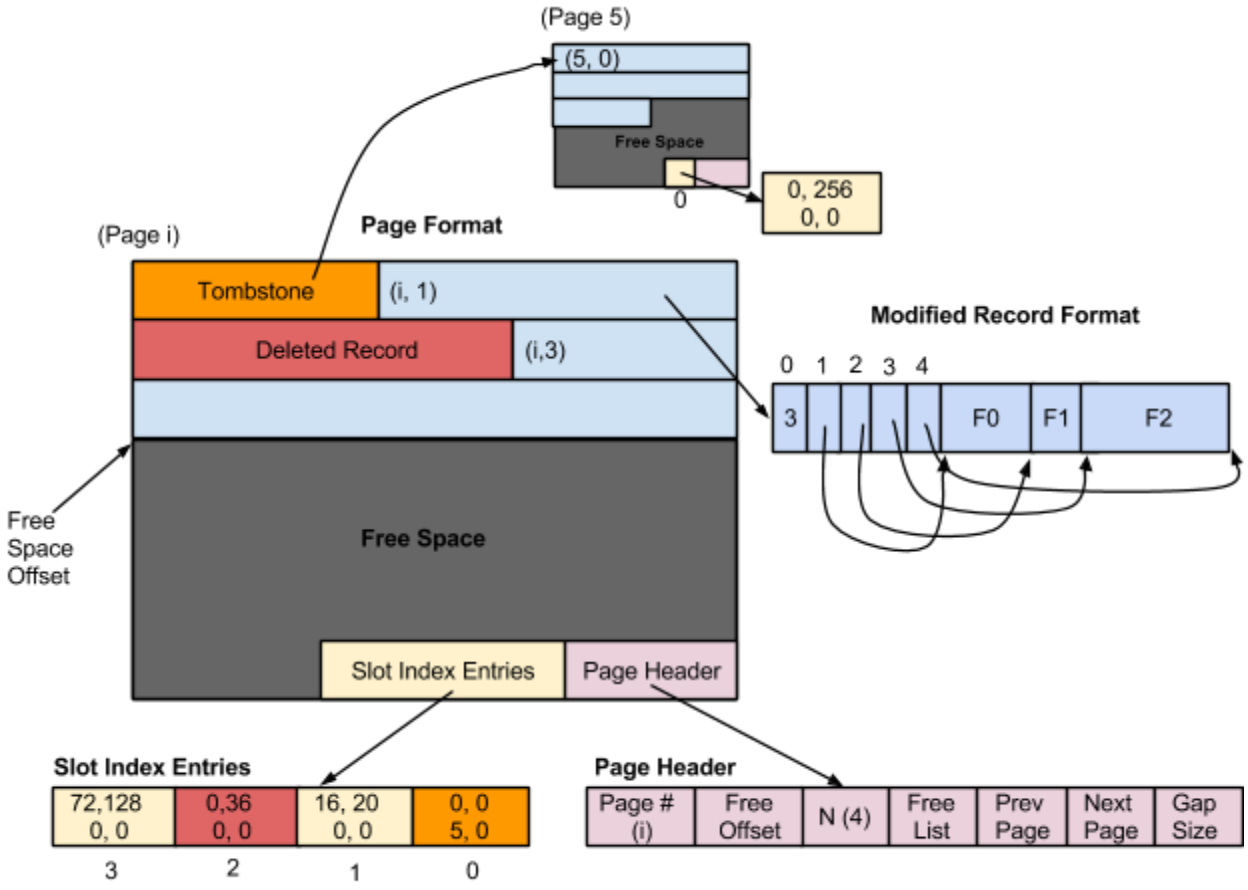

Figure 1. Updated page and record format in the RBFM layer.

System Catalog Design

A principal challenge in this portion of the project was settling how we wanted to store table and attribute information, or metadata, in our relation module. Ultimately, in order to support potential features in later iterations of the project, we settled on representing such information by distinguishing between attribute and table metadata and separating such data into separate records. The information about a single relation, which is encompassed by a single piece of table metadata record and one or more attribute metadata records, are tied together using a linked-list design similar to our page format in the RBFM layer. More specifically, the table metadata record stores an RID pointer to the first attribute metadata record in the system attribute table, which is a physically separate file where the table metadata is stored, and each of

these attribute metadata records store RID pointers to the next attribute in the table alongside other important information such as the attribute type, owning table name, length, and name, among other information.

Since the system catalog is itself a relation in the system, this is always stored in the first slot on page 1 (the first usable page in the relation file). In order to store information about other relations in the system, each table metadata record stores RID pointers to the next table metadata record, which in turn have the same format as the catalog table metadata record (i.e., they have all the necessary table metadata and a RID pointer to the first attribute metadata record entry for that relation in the system attribute table).

This design and on-disk memory layout is illustrated in Figure 2 below.

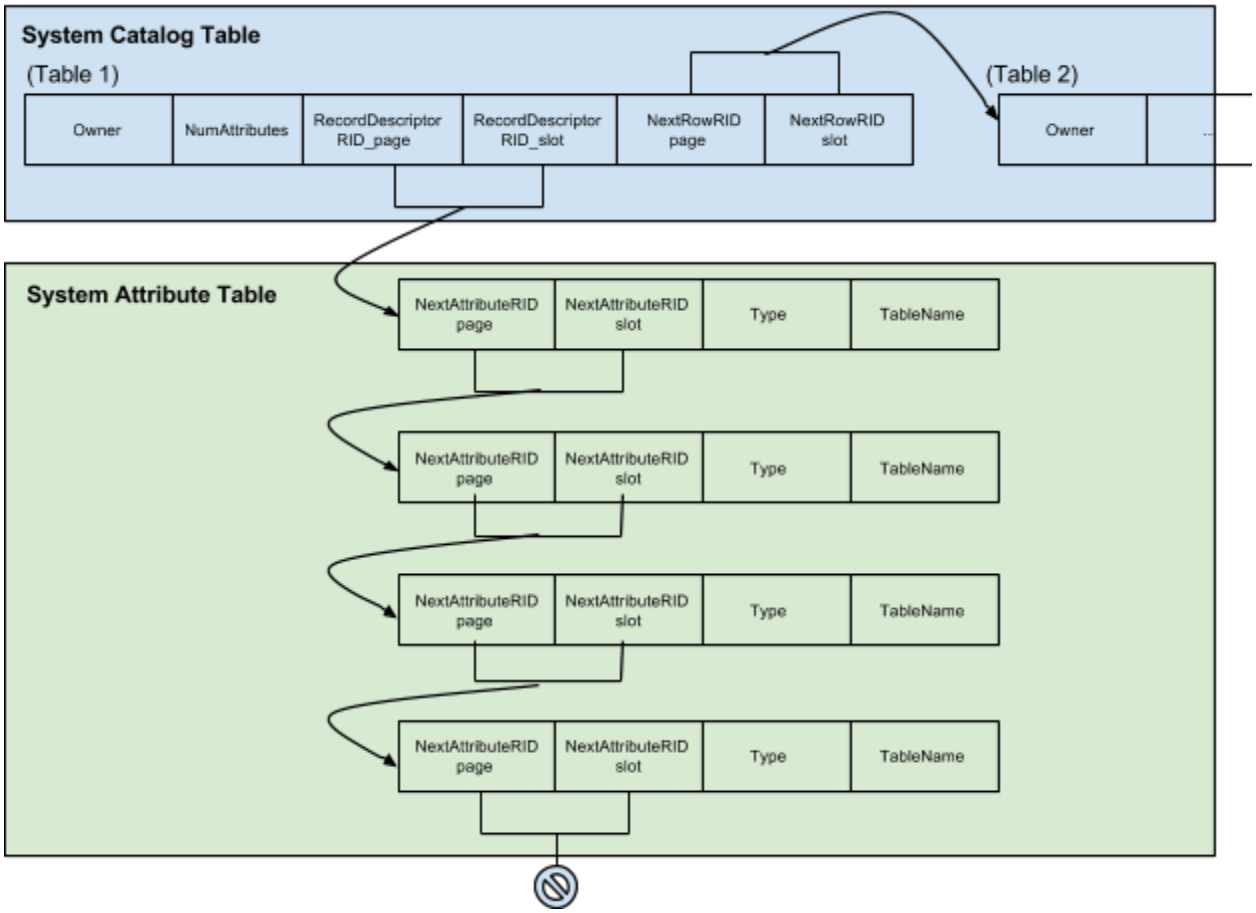

Figure 2. Relation metadata storage scheme.

Also, since the metadata for relations is not updated very frequently, we keep a shadow copy of everything stored on disk in memory so as to facilitate more efficient catalog usage through less I/Os. Of course, whenever relation or any piece of corresponding attribute information is updated, this shadow copy is persisted to disk for permanent storage.

RM Layer

After settling on a design for the system catalog, which served as a blueprint to store all relation information, implementing such relation information became a matter of constructing correct record descriptors and raw data buffers which are then passed to the lower RBFM layer methods. For example, to implement the createTable method, we created the corresponding relation and attribute metadata records using pre-defined record descriptors and invoked the insertRecord methods on the underlying RBFM instance (after creating and opening a handle to the new relation file). Also, “adjacent” metadata records are updated on disk to reflect the addition of the new table. That is, the forward RID pointers in the most recently added relation metadata record of the system catalog table is updated to point to the RID of the newly created relation, and the attribute metadata records of the new relation are inserted (in linked-list form) to the system attribute table. All of these updates are persisted to disk and the in-memory shadow copies are updated for future use by the remaining RM layer methods.

With the logic for managing relations encapsulated in the relation-oriented methods (i.e., createTable, deleteTable, etc.), the remaining tuple-oriented methods were implemented as simple wrappers around the corresponding RBFM layer methods. For example, insertTuple was implemented by direct invocation of insertRecord in the RBFM layer.