

Record Management

Shan-Hung Wu & DataLab
CS, NTHU

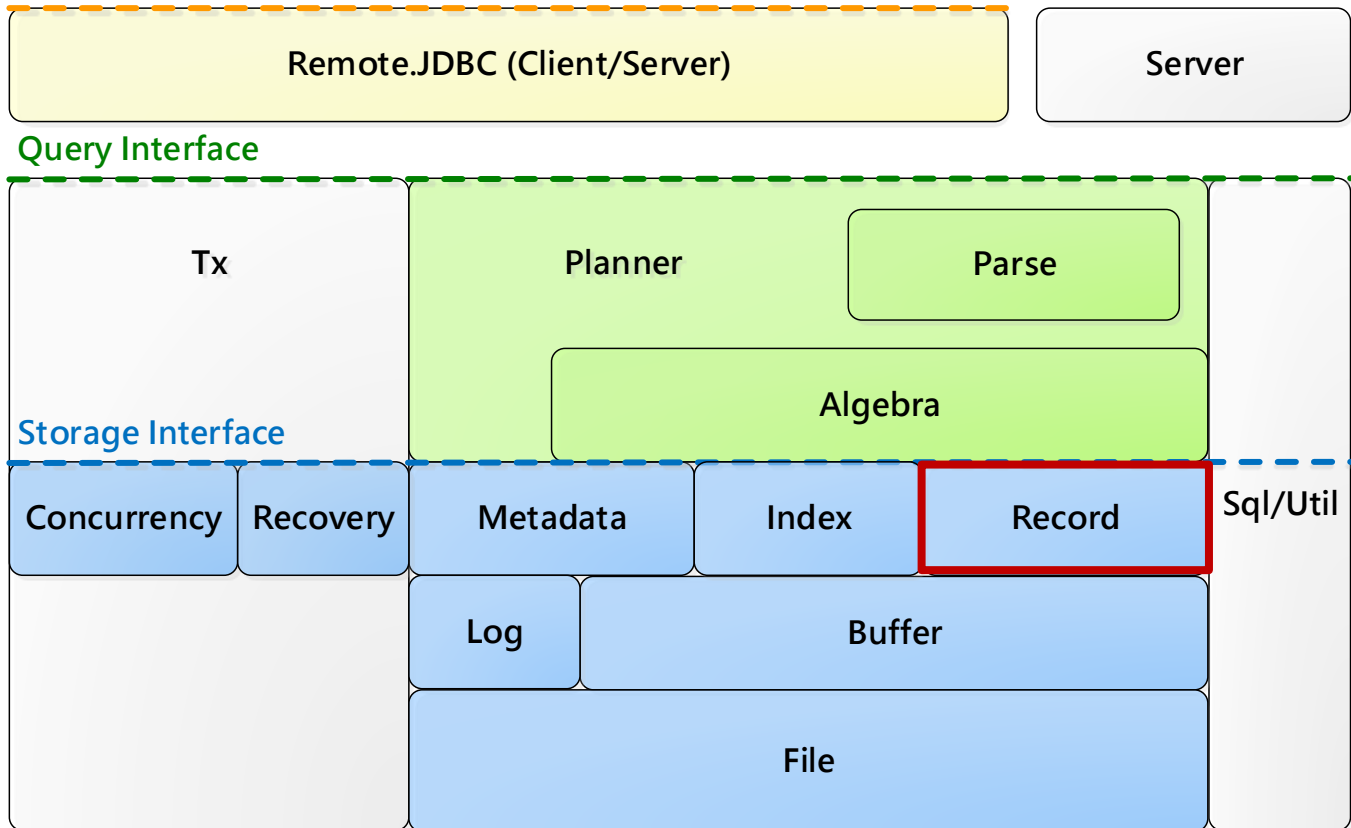
Outline

- Overview
- Design Considerations for Record Manager
- The VanillaCore Record Manager

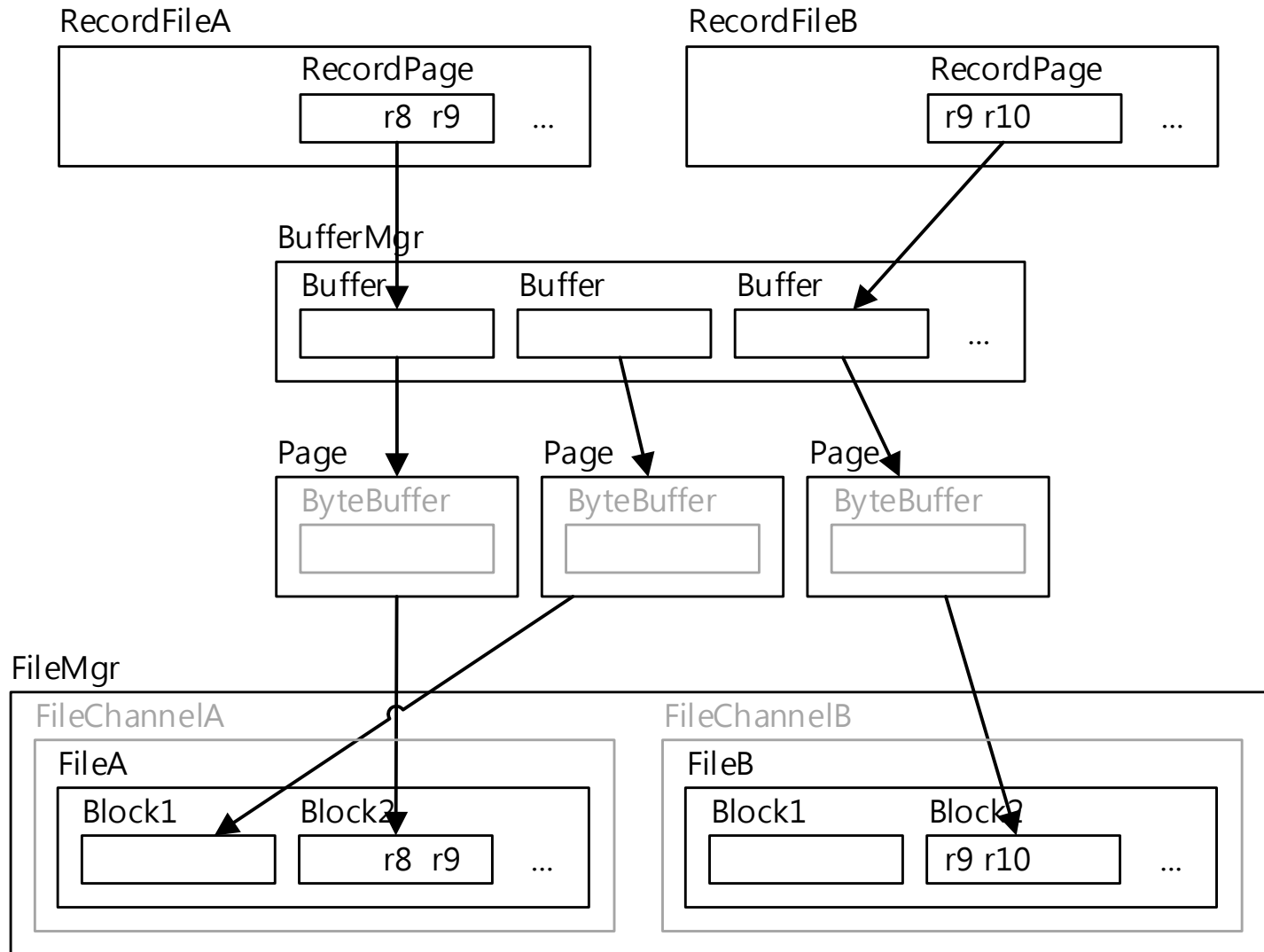
Where?

VanillaCore

JDBC Interface (at Client Side)



Data Access Layers



Record Management

- Main interface: `RecordFile`
 - An iterator of records in a file
 - One instance per `TableScan`
 - Via `VanillaDb.catalogMgr().getTableInfo(tblName, tx).open()`
 - *Thread local*

Responsibilities of RecordFile

- To decide how records are stored in a file
- To decide which block to pin
 - To save the cost of buffer access
- To work with the recovery and concurrency managers
 - To ensure tx ACID
 - Discussed later

Logical Schema vs. Physical Schema

- Record manager converts (logical) schema to *physical schema*

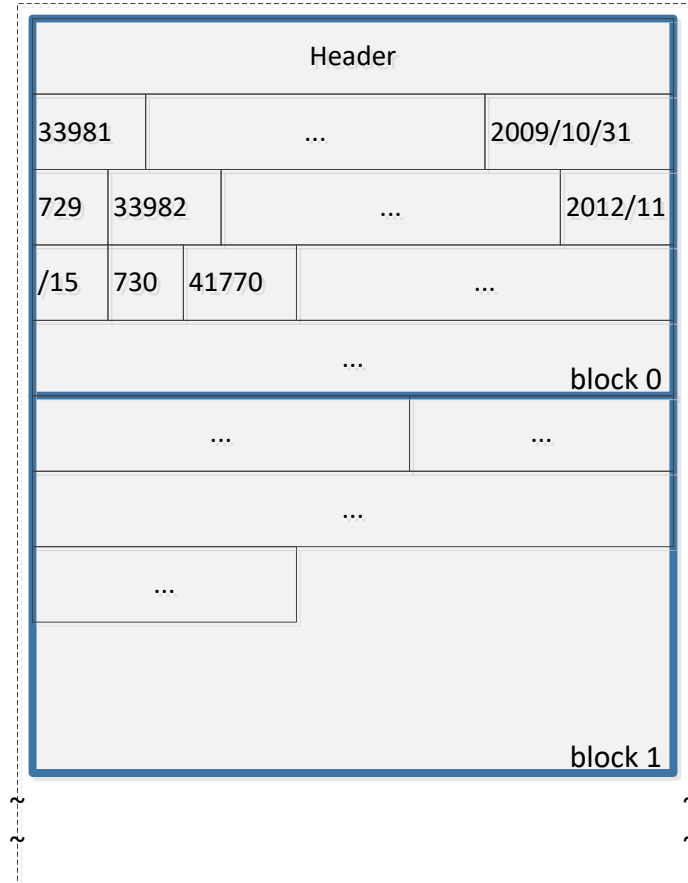
blog-posts

blog-id	url	created	author-id
33981	...	2009/10/31	729
33982	...	2012/11/15	730
41770	...	2012/10/20	736
45896	...	2012/10/31	729
50633	...	2013/01/15	25
55868	...	2013/8/21	199

record



file



Design Considerations for Physical Schema

- Should all records of a table be stored in the same file?
- Should a record be placed entirely within one block?
- Should all fields of a record to be stored next to each other?
- Should a field be represented as a fixed number of bytes?
- How to manage free space?

Outline

- Overview
- Design Considerations for Record Manager
- The VanillaCore Record Manager

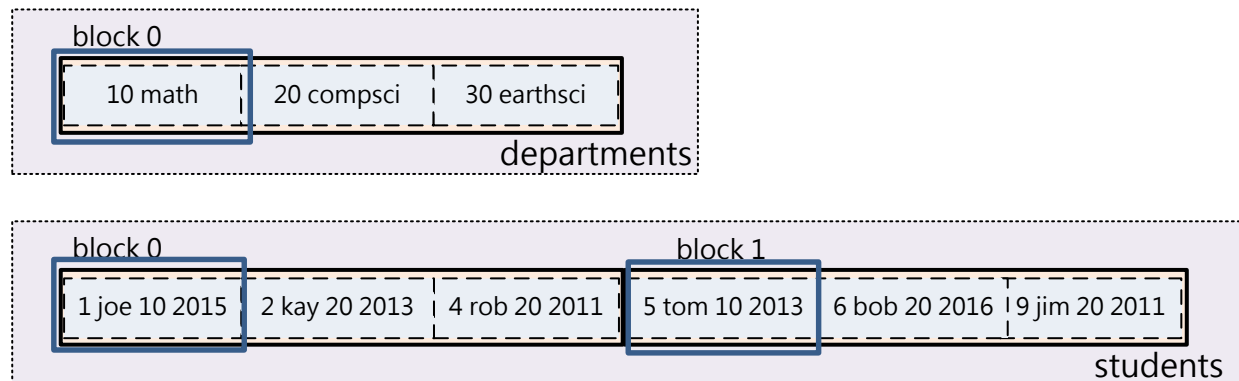
Should all records of a table be stored in the same file?

Homogeneous vs. Heterogeneous Files

- A file is *homogeneous* if all of its records come from the same table
 - Makes single-table queries easy to answer
- Allow *heterogeneous* files or not?

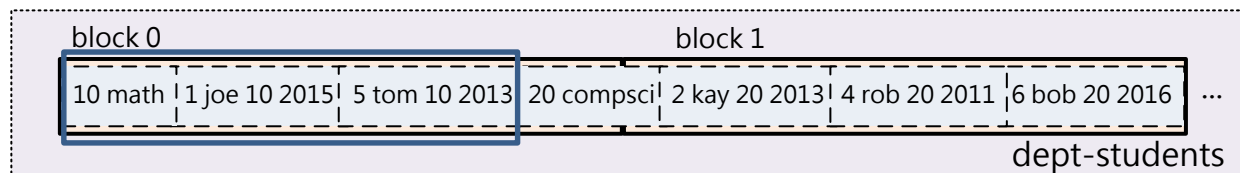
Tradeoff: Efficiency vs. Flexibility

- Query: SELECT s-name FROM students, departments WHERE d-id = major-id
- Homogeneous file
 - The disk drive has to seek back and forth between the blocks of two files



Tradeoff: Efficiency vs. Flexibility

- Query: SELECT s-name FROM students, departments WHERE d-id = major-id
- Nonhomogeneous file
 - Stores the students and departments records in the same file
 - Records are *clustered* on department id
 - Requires fewer block accesses to answer this join query



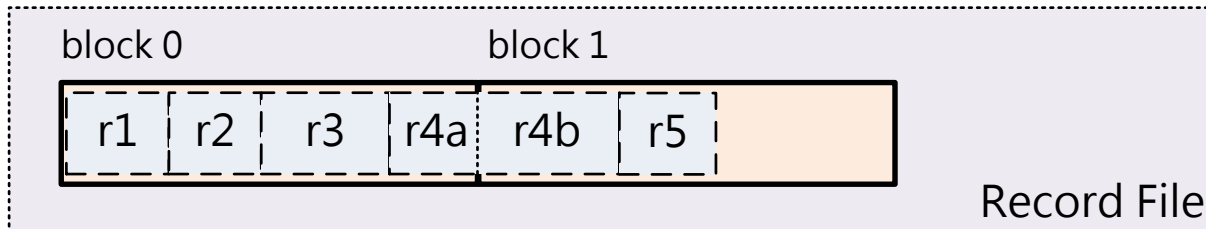
Homogeneous vs. Nonhomogeneous Files

- Nonhomogeneous file
 - Pros
 - Clustering improves the efficiency of queries that join the clustered tables
 - Cons
 - Single-table queries become less efficient
 - Join queries on non-clustered field will also be less efficient
 - Suits only for schemas with hierarchy

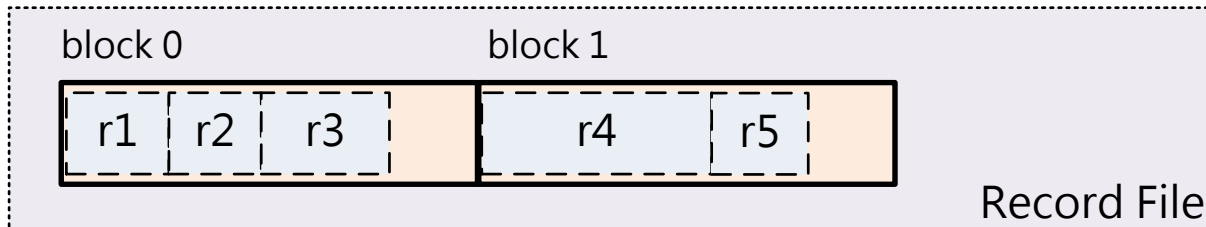
Should each record be placed entirely within one block?

Spanned vs. Unspanned Records

- A ***spanned record*** is a record whose values span two or more blocks



spanned



unspanned

Spanned vs. Unspanned Records

- Spanned record
 - Pros
 - No disk space is wasted
 - Record size is not limited by block size
 - Cons
 - Reading one record may require multiple blocks access and reconstruction

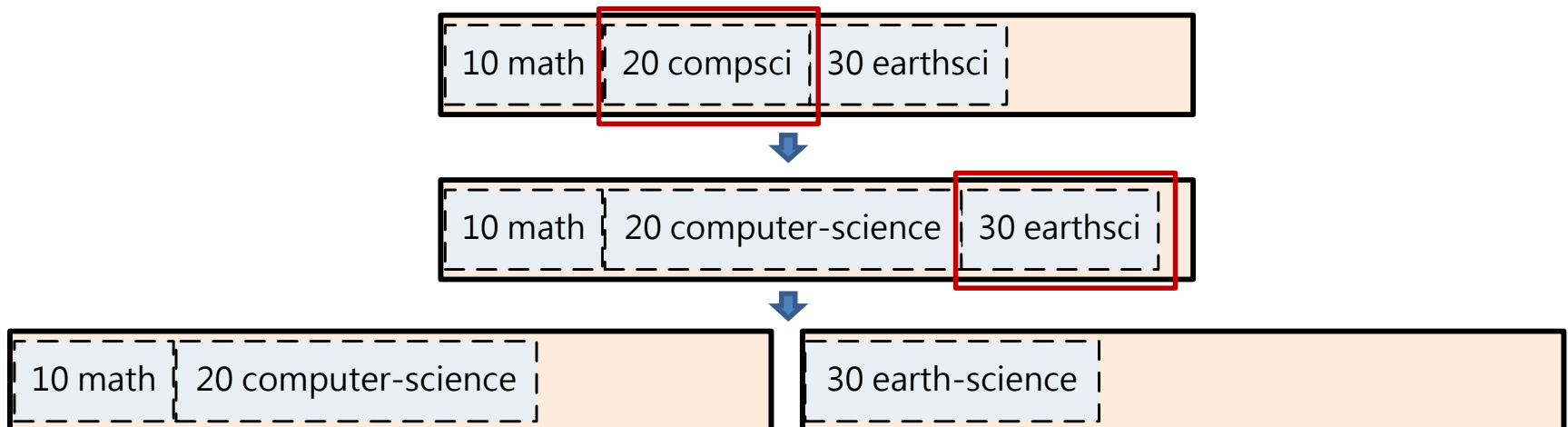
Is each field in a record represented as a fixed number of bytes?

Fixed-Length vs. Variable-Length Fields

- Field types supported by SQL
 - int, varchar(n), text, etc.
- Most of types are naturally fixed-length
 - All numeric and data/time types
- A ***fixed-length field representation*** uses the same number of bytes to hold each value of the field
 - Integer can be stored as 4-bytes binary value
- How about those fields with variable-length types?
 - varchar(n), clob(n), etc.

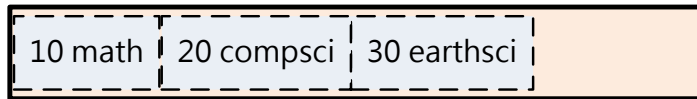
Fixed-Length vs. Variable-Length Fields

- Consider a field “d-name” defined as type varchar(20) using the variable-length representation
- Modifying this field may require rearrange other records

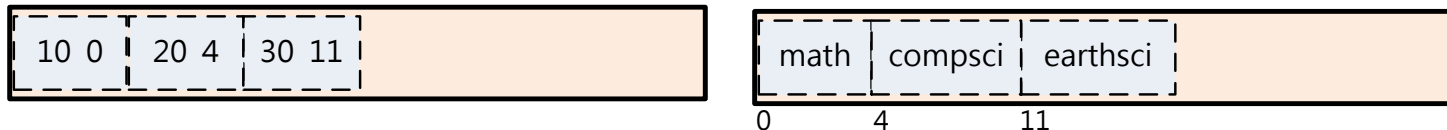


Storing Variable-Length Fields

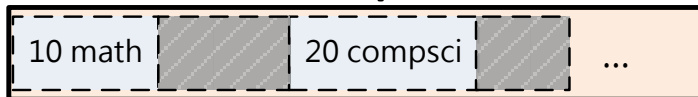
- Three different ways to store a varchar(n)
 - Variable-length representation



- Indexed representation, which stores the string value in a separate location



- Fixed-length representation, which allocates same amount of space for this field in each records



Pros & Cons

- Variable-length representation
 - Space-efficient
 - Record rearrangement is possible
- Indexed representation
 - Space-efficient (although with overhead of index)
 - Extra index access for each record read/write
 - Suits for text, clob(n)
- Fixed-length representation
 - Easy implementation of random access
 - Wastes space

Should all fields of a record to be stored next to each other?

Column-Store vs. Row-Store

- Row-oriented store

- Row-by-row sequentially on disk

- (s-id, s-name, major-id, grad-year)

1 joe 10 2015	2 kay 20 2013	4 rob 20 2011	5 tom 10 2013	6 bob 20 2016	9 jim 20 2011
---------------	---------------	---------------	---------------	---------------	---------------

- How about storing the values of a single column contiguously on disk?

- Sorted by s-id

1 2 4 5 6 9	joe kay rob tom bob jim	10 20 20 10 20 20	2015 2013 2011 2013 2016 2011
-------------	-------------------------	-------------------	-------------------------------

Pros & Cons

- Row-oriented store
 - Accessing a single row is more efficiently
 - Write-optimized
 - For OLTP workloads
- Column-oriented store
 - Efficient column read
 - Efficient column calculation (e.g., group by and aggregation)
 - Better comparison
 - For OLAP workloads

Design Considerations for Record Manager

- How to choose a proper record file structure?
- Several factors that should be taken into account
 - Workload
 - Supported SQL types
 - Schema

Implementing a File of Records

- A straightforward implementation
 - Homogeneous files
 - Unspanned records
 - Fixed-length records
 - Row-oriented store
- Treats each file as a sequence of blocks and treats each block as an array of records
 - We call such a block a *record page*

Record Page

- Divides a block into **slots**, where each slot is large enough to hold a record plus one additional integer
 - This integer is a flag that denotes the slot usage
 - 0 means “empty” and 1 means “in use”

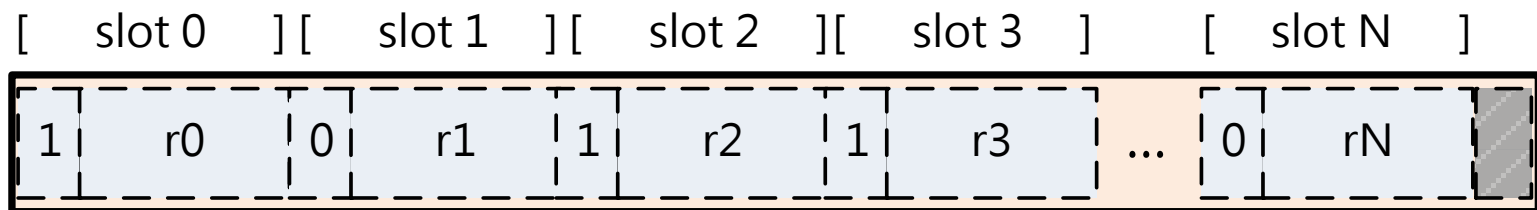


Table Information

- The table information stores
 - The record length
 - The name, type, length, and offset of each field of a record
- The table information allows the record manager to determine where values are located within the block

Table Information

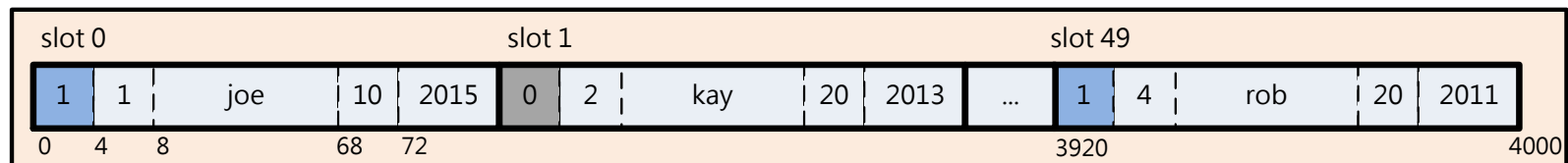
- Table information of `students` table

- Record length: 76 bytes

```
students(s-id:int,  
         s-name:varchar(20),  
         major-id:int,  
         grad-year:long)
```

- Fields information:

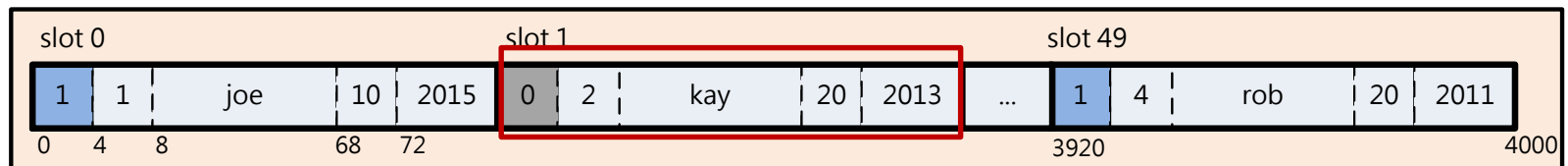
Field Name	Type	Max Size (in byte)	Offset
s-id	int	4	0
s-name	varchar(20)	60	4
major-id	int	4	64
grad-year	long	8	68



The position s-id field of record in slot n is $n * (76 + 4) + 4$

Accessing The Record Page

- To insert a new record
 - The record manager finds a slot with empty flag
 - Updates the flag as in use
 - Returns the slot number
 - If all flag values are “1”, then the block is full



Accessing The Record Page

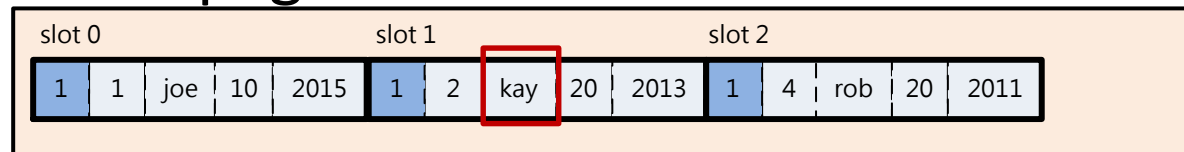
- To delete the value of the record in slot k
 - The record manager simply sets the flat at that slot to 0 as empty
- To modify a field value of the record in slot k
 - The record manager determines the location of that field, and writes the value to that location
- Each record in a page has an ID. When the records are fixed-length, the ID can be its slot number

Implementing Variable-Length Fields

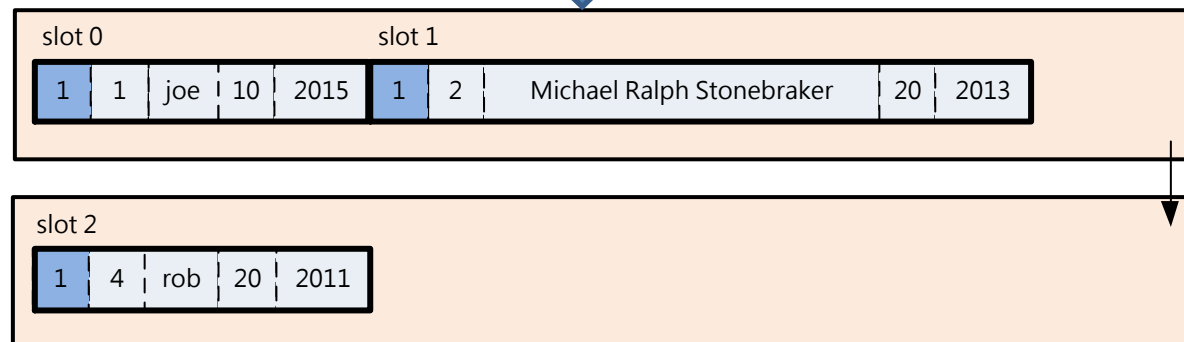
- What are the implementation changes when we want to support variable-length fields?
 - The field offsets in a record are no longer fixed
 - The records of same table can have different lengths
 - The record position cannot be calculated by multiplying its slot number by slot size
 - Modifying a field value can cause a record's length to change

Implementing Variable-Length Fields

- If the record's length changes
 - We need to shift the records after modified record
 - The shifted records may spill out of the block
 - Move to **overflow block**
- The original block and overflow block form a single large record page

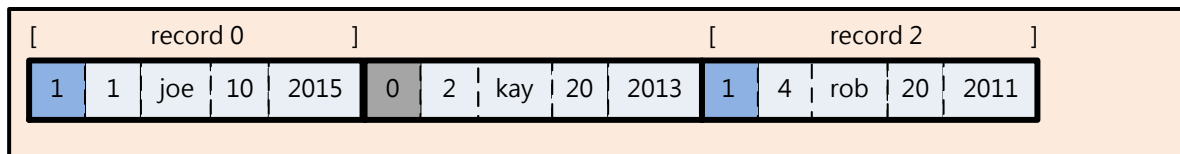
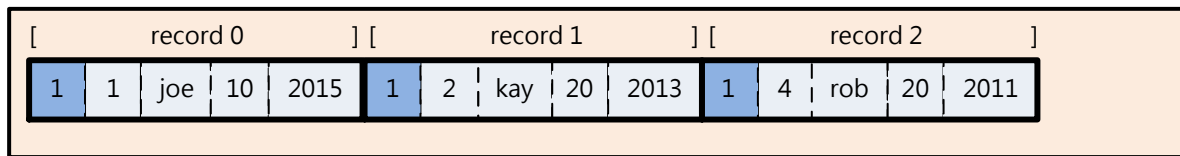


Modify the s-name of second record in original block

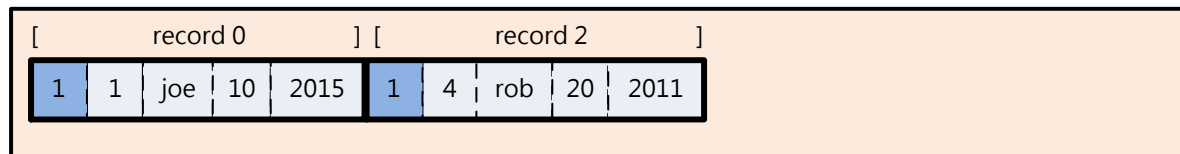


Implementing Variable-Length Fields

- How to delete a record?
 - Only set the flag to empty
 - Record size is variable, this empty space may not be re-use



- Reclaim the empty space
 - Dissociate the record's ID from slot

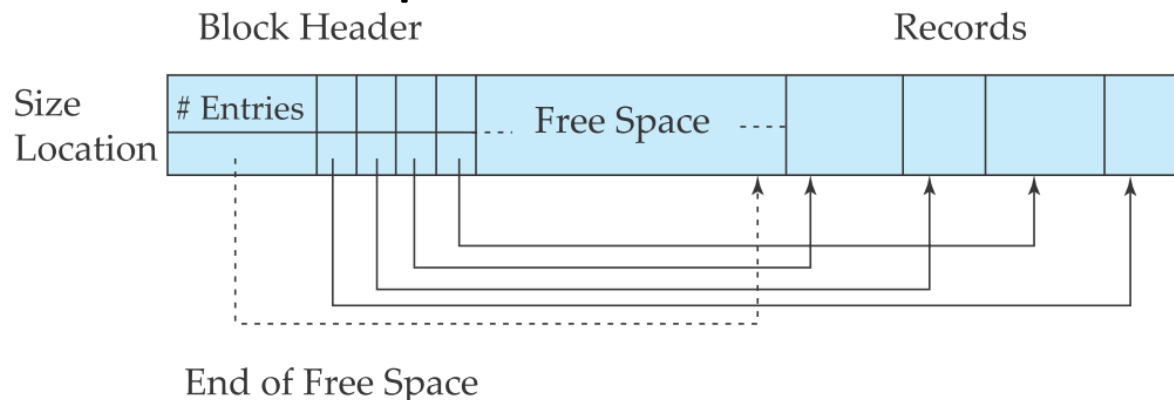


Implementing Variable-Length Fields

- The record manager cannot random access a record in a page, because it has no position information
 - We need a different *page layout*

Implementing Variable-Length Fields

- There is a header at the beginning of each record page containing following information
 - Number of records
 - The end of free space in that page
 - IDs and pointers to each record and size of each record
- The records are placed at the other end of page

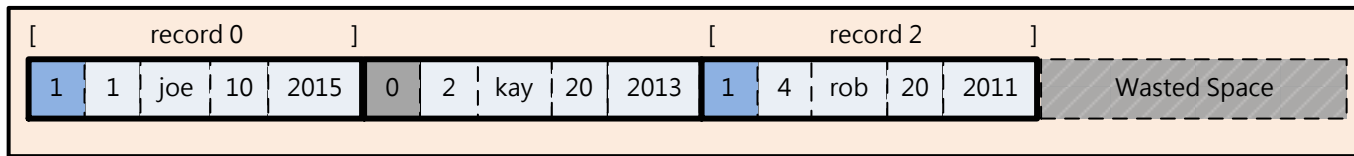


Implementing Variable-Length Fields

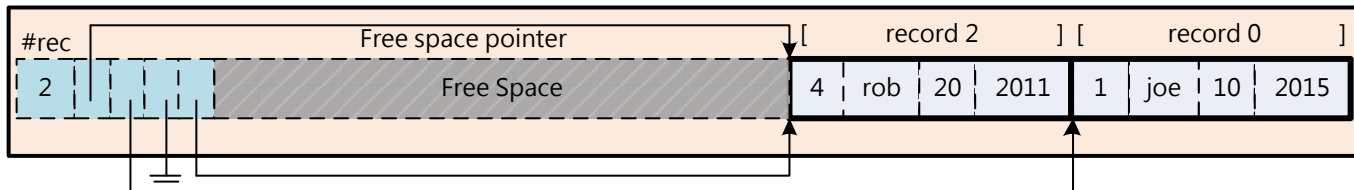
- When a modification on a record requires more spaces, the record manager will find a continuous free space within that page
- Rearranging the record page when record's length changes can eliminate the fragmentation
 - VACUUM command

Managing the Free Space Within a Record File

- Each record page in a file has different amount of free spaces
 - The fixed-length field implementation

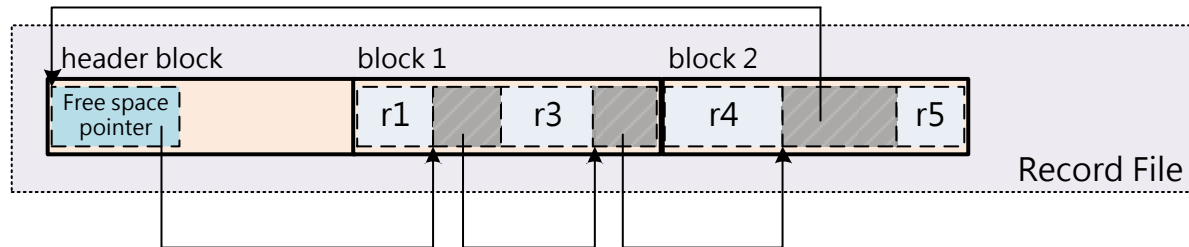


- The variable-length field implementation with id table



M1: Chaining

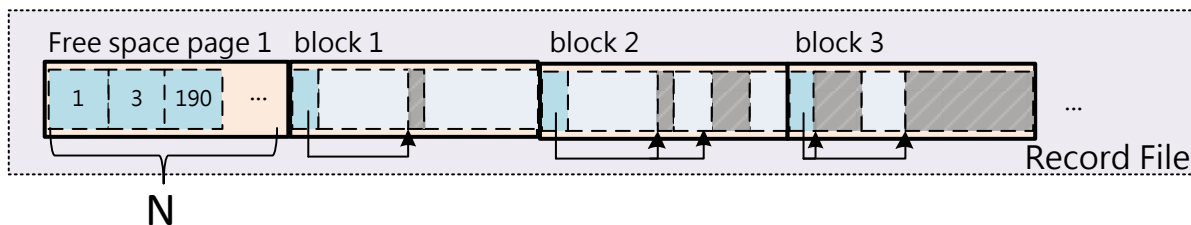
- When the client wants to insert a new record, the record manager needs to find continuous unused bytes for it
- How to manage the free space within a file?
- Chaining the free spaces



- For variable-length records, it may access many blocks to find out a large enough free space

M2: Meta-Pages

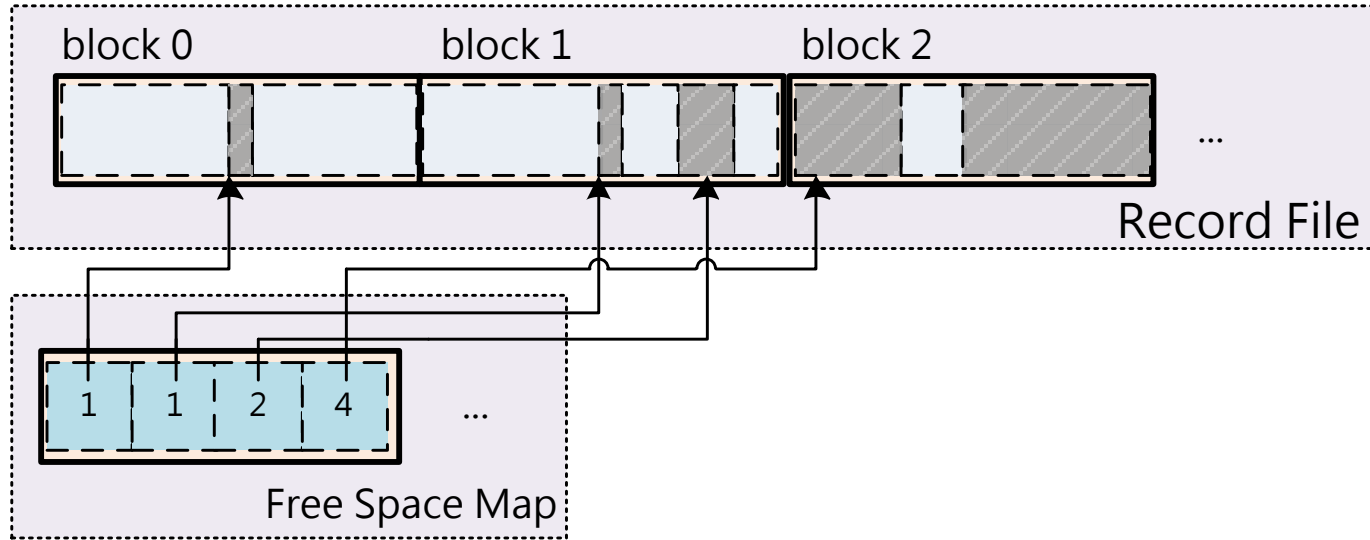
- Using special pages to track the usage of record pages
 - Allocates one free space page for N record pages
 - Free space page uses one byte to track the size of unused space size for each following page



- Microsoft SQL Server approach

M3: Meta-File

- Using additional file to track the location and size all free spaces
 - PostgreSQL approach



Outline

- Overview
- Design Considerations for Record Manager
- The VanillaCore Record Manager
 - How records are stored?
 - Which blocks to pin
 - Working with the recovery and concurrency manager to ensure tx ACID

Responsibilities of RecordFile

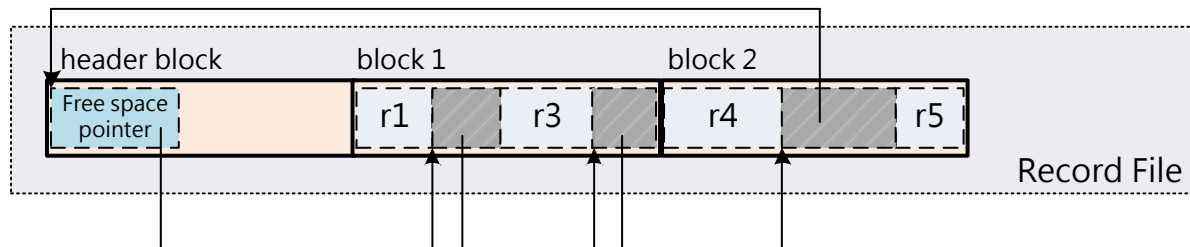
- To decide how records are stored in a file
- To decide which block to pin (to save the cost of buffer access)
- To work with the recovery and concurrency manager to ensure tx ACID

Outline

- Overview
- Design Considerations for Record Manager
- The VanillaCore Record Manager
 - How records are stored?
 - Which blocks to pin?
 - Working with the recovery and concurrency manager to ensure tx ACID

How Records are Stored?

- Choices:
 - Un-spanned record
 - Homogeneous file
 - Row-oriented store
 - Fixed-length field
 - Chained free space: $O(1)$ search time
- RecordPage: lays out records in a page
- FileHeaderPage: header of free-space chain



Using the Table Information

- The VanillaCore record manager needs to know the table information
- The classes `storage.metadata.TableInfo` and `sql.Schema` manage the table information
- The record manager can get this information from metadata manager

TableInfo
+ TableInfo(tblname : String, schema : Schema) + fileName() : String + tableName() : String + schema() : Schema + open(tx : Transaction) : RecordFile

Schema : Serializable
+ Schema() + addField(fldName : String, type : Type) + add(fldName : String, sch : Schema) + addAll(sch : Schema) + fields() : SortedSet<String> + hasField(fldName : String) : boolean + type(fldname : String) : Type + toString() : String + equals(obj : Object) : boolean + hashCode() : int

Using the Table Information

- Sample code of constructing table information

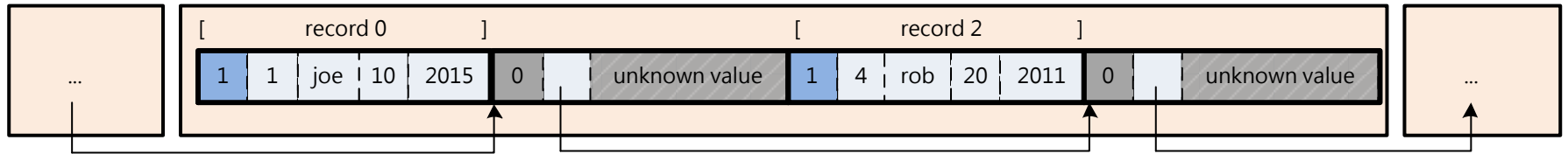
```
Schema sch = new Schema();
```

```
sch.addField("s-id", Type.INTEGER);  
sch.addField("s-name", Type.VARCHAR(20));  
sch.addField("major-id", Type.INTEGER);  
sch.addField("grad-year", Type.BIGINT);
```

```
TableInfo ti = new TableInfo("students", sch);
```


Managing the Records in a Page

- Implements the record page as following layout
 - Minimal slot size: 4+4+8 bytes (flag, pointer to next deleted slot)



- The `RecordPage` manages the records within a page
- The `RecordId` denotes the identifier of each record

RecordId

- Identifier of a record
 - `id` is equal to **slot number** because of fixed-length implementation

RecordId
+ RecordId(blk : BlockId, id : int) + block() : BlockId + id() : int + equals(obj : Object) : boolean + toString() : String + hashCode() : int

RecordPage

- Extends the interface `Record`
- Manages a buffer for the currently opened data block
- Calls the concurrency control manager to ensure the isolation property

RecordPage

RecordPage : Record
<u>+ offsetMap(sch: Schema) : Map<String, Integer></u> <u>+ recordSize(sch: Schema) : int</u> <u>+ slotSize(sch: Schema) : int</u> + RecordPage(blk : BlockId, ti : TableInfo , tx : Transaction, doLog : boolean) + close() + next() : boolean + getVal(fldName : String) : Constant + setVal(fldName : String, val : Constant) + delete(nextDeletedSlot : RecordId) + insertIntoNextEmptySlot() : boolean + insertIntoDeletedSlot(): RecordId + moveTold(id : int) + currentId() : int + currentBlk() : BlockId

Accessing Records in a Record Page

- Sample code of using a record page

```
Transaction tx = VanillaDb.txMgr().transaction(
    Connection.TRANSACTION_SERIALIZABLE, false);
TableInfo ti = VanillaDb.catalogMgr().getTableInfo(tableName, tx);
String fileName = ti.fileName();
RecordId lastDeletedRid = ...;
BlockId blk = new BlockId(fileName, 235);
RecordPage rp = new RecordPage(blk, ti, tx, true); // pin the buffer

// Part1: read and delete
while (rp.next()) {
    Constant sid = rp.getVal("s-id");
    if (sid.equals(new IntegerConstant(50))) {
        rp.delete(lastDeletedRid);
        lastDeletedRid = new RecordId(rp.currentBlk(), rp.currentId());
    }
}

// Part 2: insert into empty slot if exist
rp.moveToId(-1); // point before the first record
boolean hasFreeSlot = rp.insertIntoNextEmptySlot();
if (hasFreeSlot) {
    rp.setVal("s-id", new IntegerConstant(65));
    ...
}
rp.close(); // unpin the buffer
tx.commit();
```

Formatting Record Page

- A record page has a specific structure
 - Partitioned into slot, with the value of the first integer in each slot as usage flag
- Formatting the record page before it can be used
- The class `RecordFormatter` performs this service, via its method `format`

RecordFormatter : PageFormatter
+ RecordFormatter(ti : TableInfo) + format(page : Page)

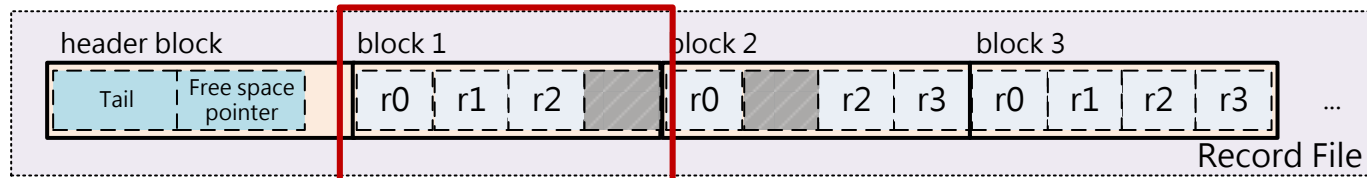
File Header

- The class `FileHeaderPage` manages the header
 - The pointer to the deleted slot chain
 - The tail slot

FileHeaderPage
<ul style="list-style-type: none">+ <code>FileHeaderPage(fileName : String, tx : Transaction)</code>+ <code>close()</code>+ <code>hasDataRecords() : boolean</code>+ <code>hasDeletedSlots() : boolean</code>+ <code>getLastDeletedSlot() : RecordId</code>+ <code>getTailSlot() : RecordId</code>+ <code>setLastDeletedSlot(rid : RecordId)</code>+ <code>setTailSlot(rid : RecordId)</code>

Managing the Records in a File

- A record file consists of several record pages
 - Data access API is similar to record pages
- Record file manages the file properties
 - File header, file size
 - Appends new block at the end of file
 - Maintains the current position in a file and uses the data manipulation methods of the record page



RecordFile

- Manages a file of records and calls the concurrency manager to ensure isolation property
- Provides methods for iterating through the records and accessing their contents

RecordFile

RecordFile: Record
<u>+ formatFileHeader(fileName : String, tx : Transaction)</u> + RecordFile(ti : TableInfo , tx : Transaction, doLog : boolean) + close() + beforeFirst() + next() : boolean + getVal(fldName : String) : Constant + setVal(fldName : String, val : Constant) + delete() + insert() + moveToRecordId(rid : RecordId) + currentRecordId() : RecordId + fileSize() : long

Accessing Records in a Record File

- Sample code of using a record file

```
Transaction tx = VanillaDb.txMgr().transaction(
    Connection.TRANSACTION_SERIALIZABLE, false);
TableInfo ti = ...;
RecordFile rf = ti.open(tx, true);
rf.beforeFirst();

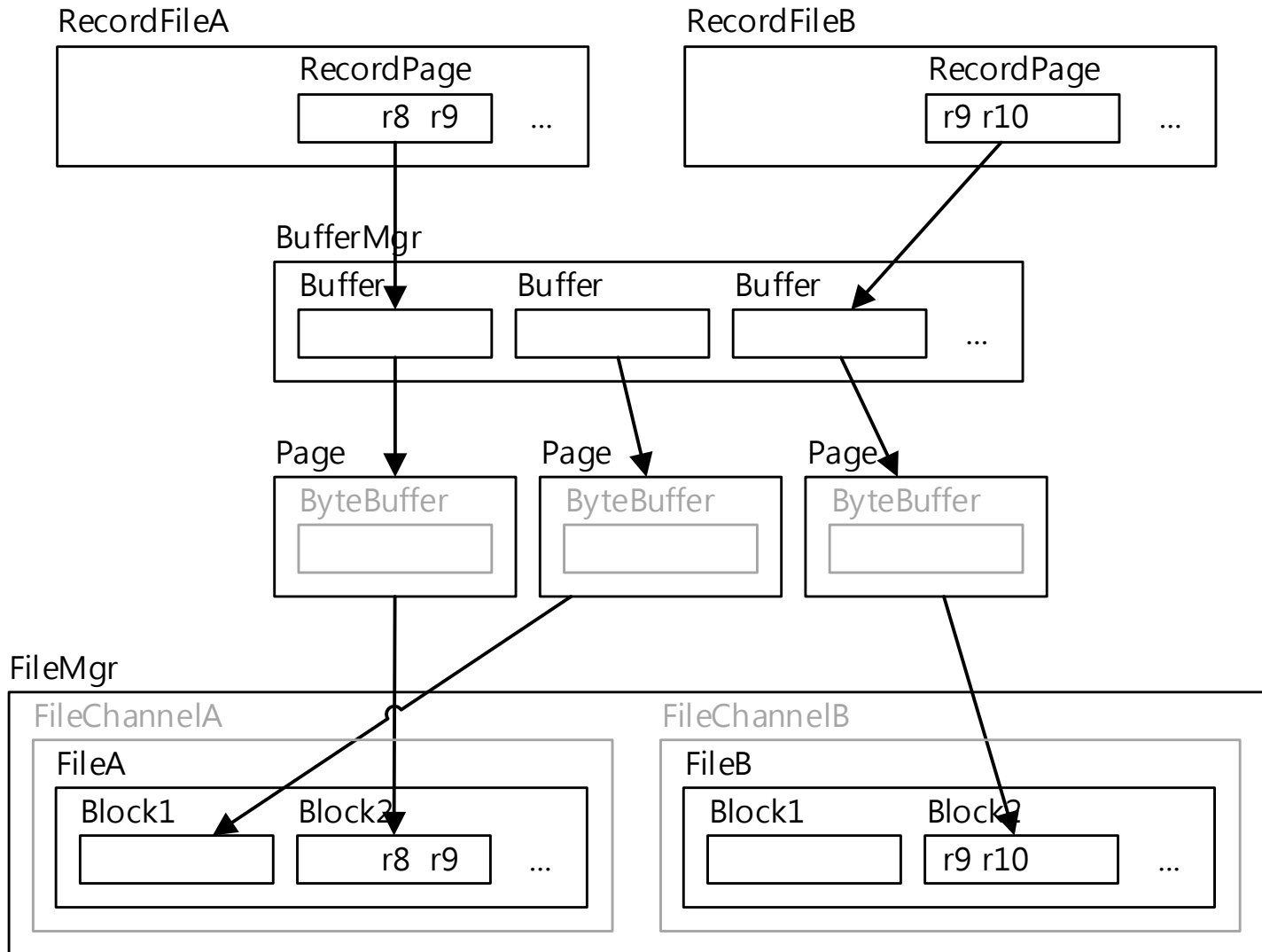
// Part 1: reads records and delete records
while (rf.next())
    if (rf.getVal("s-id").equals(new IntegerConstant(50)))
        rf.delete();
rf.close();

// Part 2: insert new record
rf = ti.open(tx, true);
for (int id = 0; id < 100; id++) {
    rf.insert();
    rf.setVal("s-id", new IntegerConstant(id));
    rf.setVal("s-name", new VarcharConstant("student" + id));
    rf.setVal("major-id", new IntegerConstant((id % 3 + 1) * 10));
    rf.setVal("grad-year", new BigIntConstant(2016));
}
rf.close();
```

Caution:

When inserting a new record, all the fields should have inserted values. Otherwise, the user might read some unpredictable value

Recap of Data Access Layers



Outline

- Overview
- Design Considerations for Record Manager
- The VanillaCore Record Manager
 - How records are stored?
 - **Which blocks to pin?**
 - Working with the recovery and concurrency manager to ensure tx ACID

Which Block to Pin?

- Each `RecordFile` instance pins only two pages:
 - `RecordPage` corresponding to the current position
 - `FileHeaderPage`
- Unpin upon `close()`
 - This is why a JDBC user should close a `ResultSet` as soon as possible

Outline

- Overview
- Design Considerations for Record Manager
- The VanillaCore Record Manager
 - How records are stored?
 - Which blocks to pin?
 - Working with the recovery and concurrency manager to ensure tx ACID

Tx Support

- C and I by working with **ConcurrencyManager**
 - All read/write from/to files and blocks must obtain appropriate locks first via
`concurrencyMgr.read/modifyXxx()`
- A and D by working with **RecoveryManager**
 - All set values are logged via
`recoveryMgr.logXxx()`
 - By virtue of WAL implementation in memory-management layer

References

- Database page layout of PostgreSQL.
<http://www.postgresql.org/docs/8.0/static/storage-page-layout.html>
- Microsoft SQL Server page structure.
[http://msdn.microsoft.com/en-us/library/ms190969\(v=sql.105\).aspx](http://msdn.microsoft.com/en-us/library/ms190969(v=sql.105).aspx)
- Database Design and Implementation, chapter 15.
Edward Sciore.
- Database system concepts 6/e, chapter 10.
Silberschatz.