Indexing

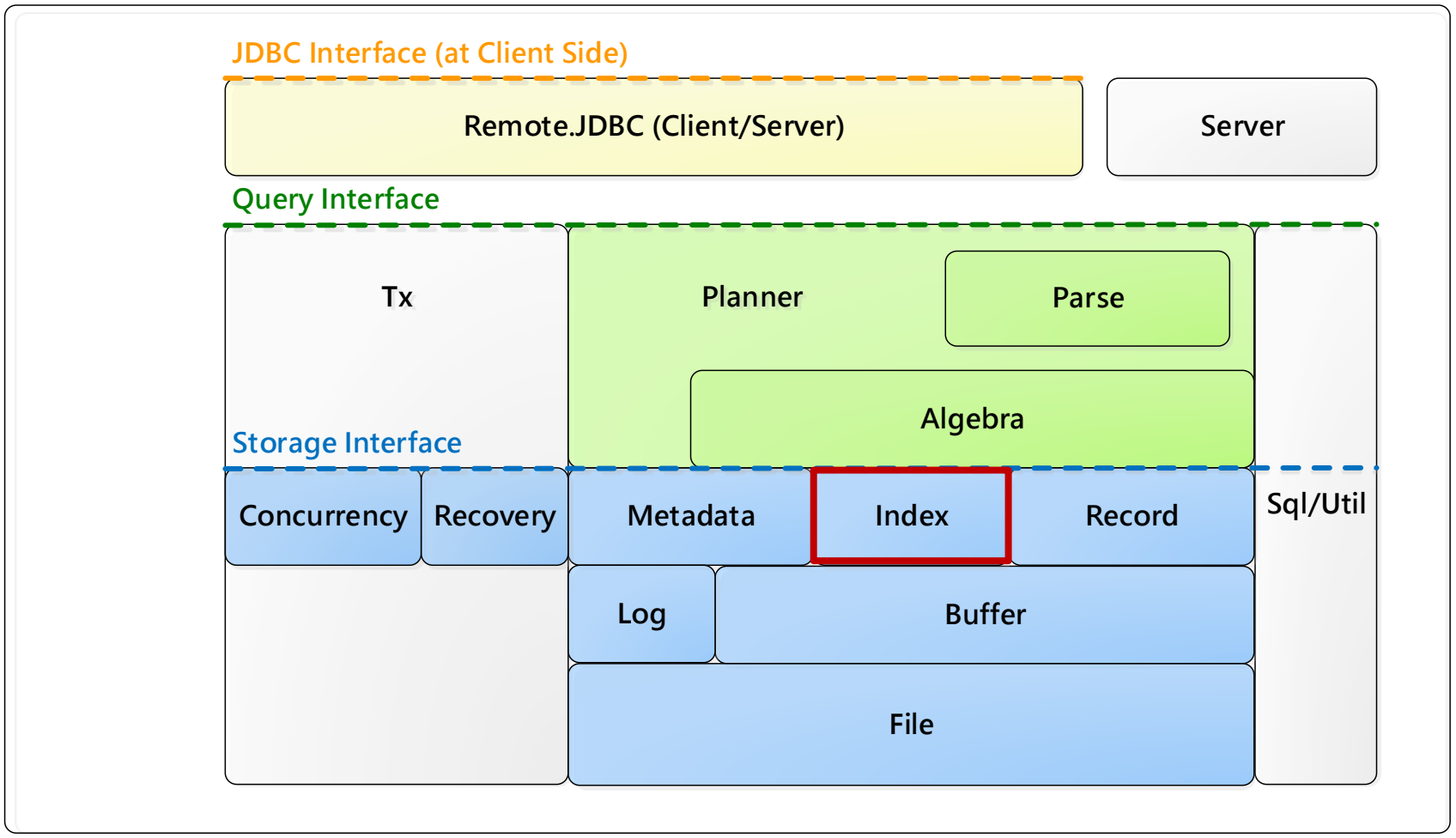
vanilladb.org

Outline

- Overview
- The API of Index in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Related Relational Algebra and Update Planner
- Transaction management revisited

Where are we?

VanillaCore

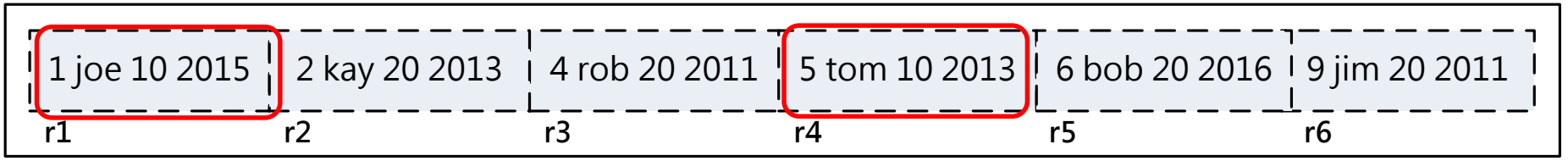


Outline

- Overview
- The API of Index in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Related Relational Algebra and Update Planner
- Transaction management revisited

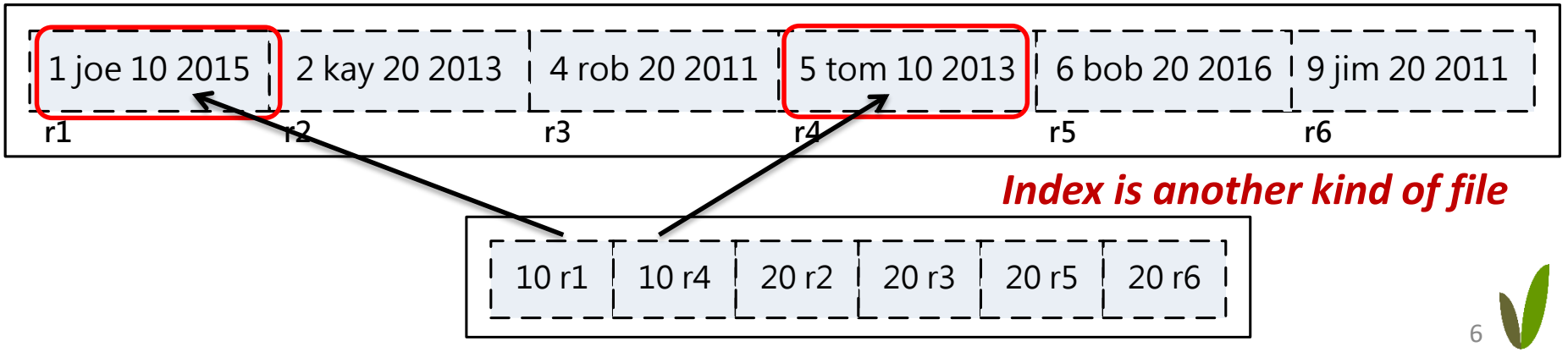
What is Index?

- Query and its stratified records of a table
  - `SELECT * FROM students WHERE dept = 10`
- We are usually interested in only a few of its records
  - Full table scan results in poor performance



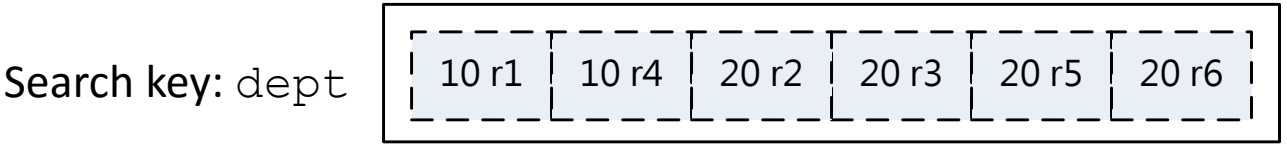
What is Index?

- Query and its stratified records of a table
  - `SELECT * FROM students WHERE dept = 10`
- Definition: ***Index***
  - An data structure that is intended to help us find rids of records that meet a selection condition

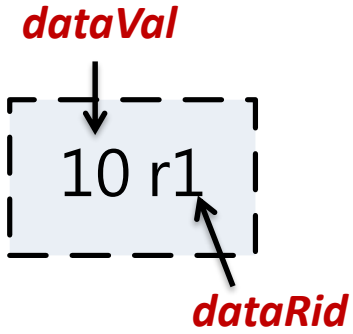


Related Terms

- Every index has an associated ***search key***
  - A collection of one or more fields of the table



- ***Primary index*** vs. ***secondary index***
  - If search key contains primary key, then called primary index
- Index entry (index record)
  - <data value, data rid>



Related Terms

- An index is designed to speed up ***equality*** or ***range selections*** on the search key
  - `dept = 10`
  - `dept > 30 and dept < 100`

SQL Statements to Create Indexes

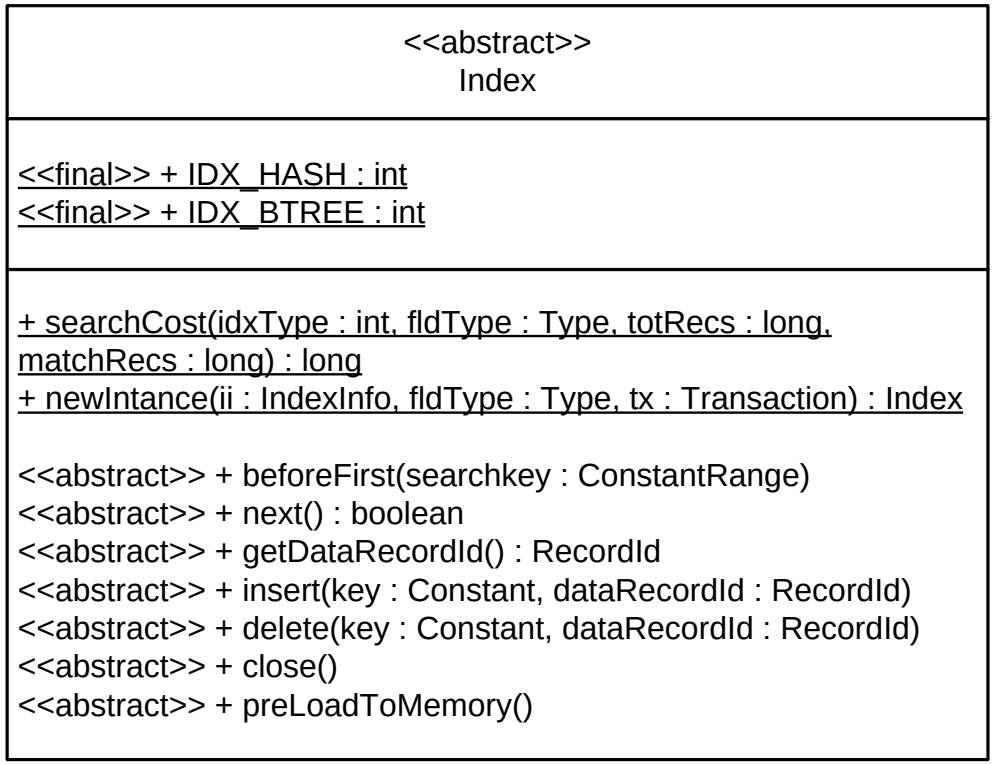
- The SQL:1999 standard does not include any statement for creating or dropping index structures
- Creating index in VanillaCore
  - An index only supports ***one*** indexed field
  - `CREATE INDEX index-name ON table-name(field-name)`
  - **e.g.,** `CREATE INDEX dept-of-stud ON students(dept)`

Outline

- Overview
- The API of Index in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Related Relational Algebra and Update Planner
- Transaction management revisited

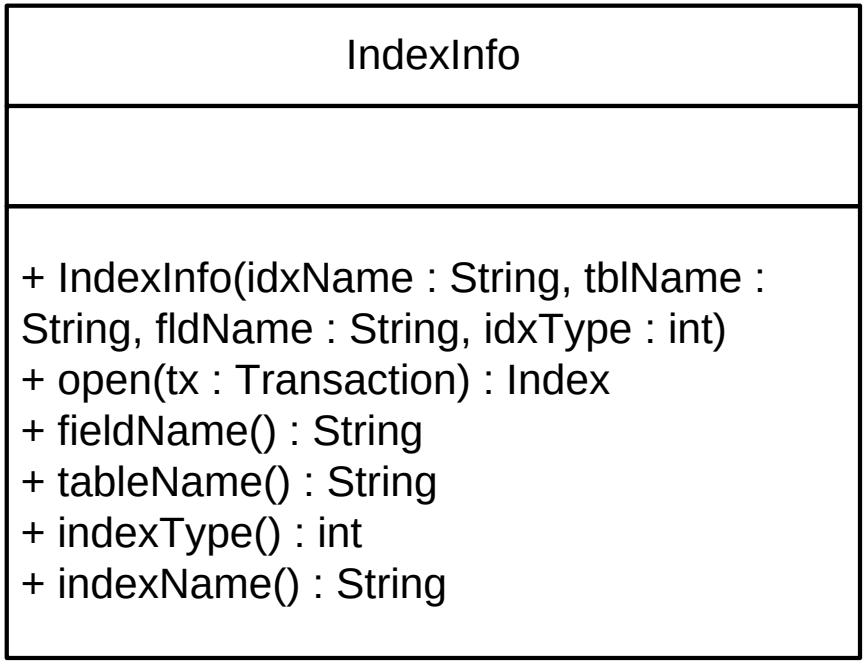
The Index in VanillaCore

- The abstract class `Index` in `storage.index`
  - Defines the API of the index in VanillaCore



IndexInfo

- The information about an index
- Similar to TableInfo



Using an Index in VanillaCore

- Example of using index
  - `SELECT sname FROM students WHERE dept = 10`

```
Transaction tx = VanillaDb.txMgr().newTransaction(
          Connection.TRANSACTION_SERIALIZABLE, false);

// Open a scan on the data table
Plan studentPlan = new TablePlan("students", tx);
TableScan studentScan = (TableScan) studentPlan.open();

// Open index on the field dept of students table
Map<String, IndexInfo> idxmap =
          VanillaDb.catalogMgr().getIndexInfo("students", tx);
Index deptIndex = idxmap.get("dept").open(tx);

// Retrieve all index records having dataval of 10
deptIndex.beforeFirst(ConstantRange
          .newInstance(new IntegerConstant(10)));
while (deptIndex.next()) {
     // Use the rid to move to a student record
     RecordId rid = deptIndex.getDataRecordId();
     studentScan.moveToRecordId(rid);
     System.out.println(studentScan.getVal("sname"));
}

deptIndex.close();
studentScan.close();
tx.commit();
```


Updating Indexes in VanillaCore

- `INSERT INTO student (sid,sname,dept,gradyear) VALUES (7,'sam',10,2014)`

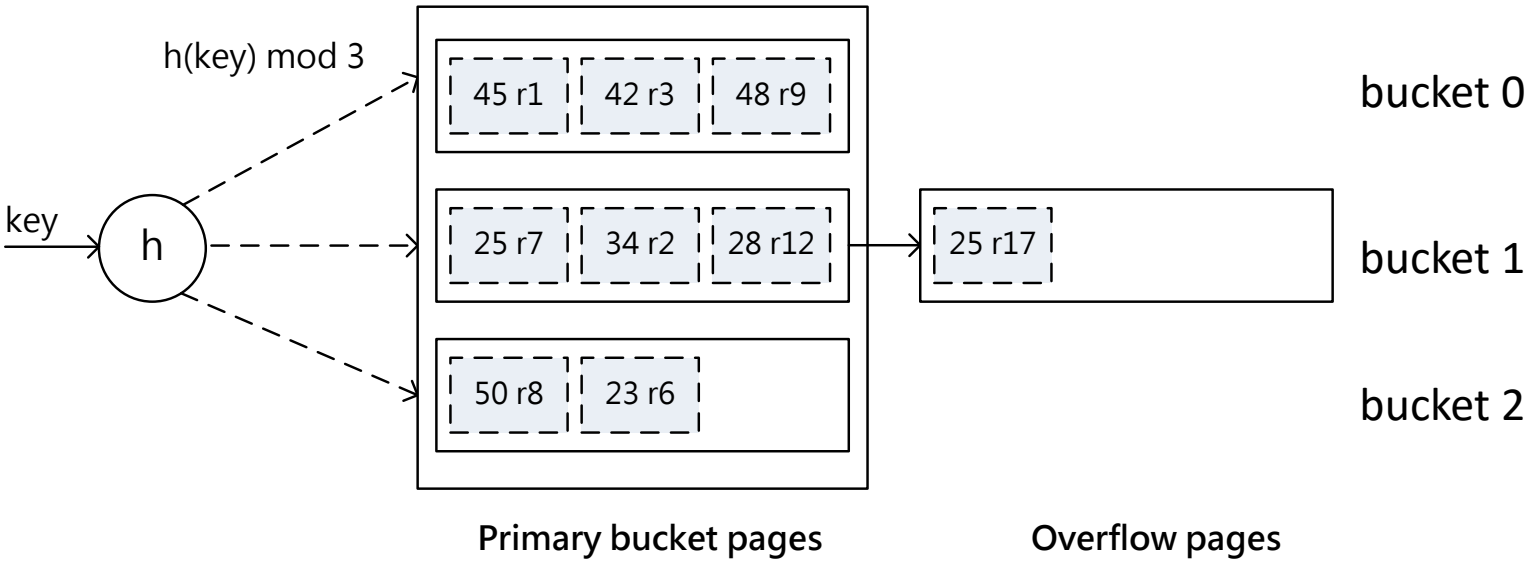
```
Transaction tx = VanillaDb.txMgr().newTransaction(
            Connection.TRANSACTION_SERIALIZABLE, false);
TableScan studentScan = (TableScan) new TablePlan("students", tx).open();

// Create a map containing all indexes of students table
Map<String, IndexInfo> idxMap = VanillaDb.catalogMgr().getIndexInfo(
            "students", tx);
Map<String, Index> indexes = new HashMap<String, Index>();
for (String fld : idxmap.keySet())
      indexes.put(fld, idxMap.get(fld).open(tx));

// Insert a new record into students table
studentScan.insert();
studentScan.setVal("sid", new IntegerConstant(7));
studentScan.setVal("sname", new VarcharConstant("sam"));
studentScan.setVal("dept", new IntegerConstant(10));
studentScan.setVal("grad", new IntegerConstant(2014));

// Insert a record into each of the indexes
RecordId rid = studentScan.getRecordId();
for (String fld : indexes.keySet()) {
      Constant val = studentScan.getVal(fld);
      Index idx = indexes.get(fld);
      idx.insert(val, rid);
}

for (Index idx : indexes.values())
      idx.close();
studentScan.close();
tx.commit();
```

Outline

- Overview
- The API of Index in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Related Relational Algebra and Update Planner
- Transaction management revisited

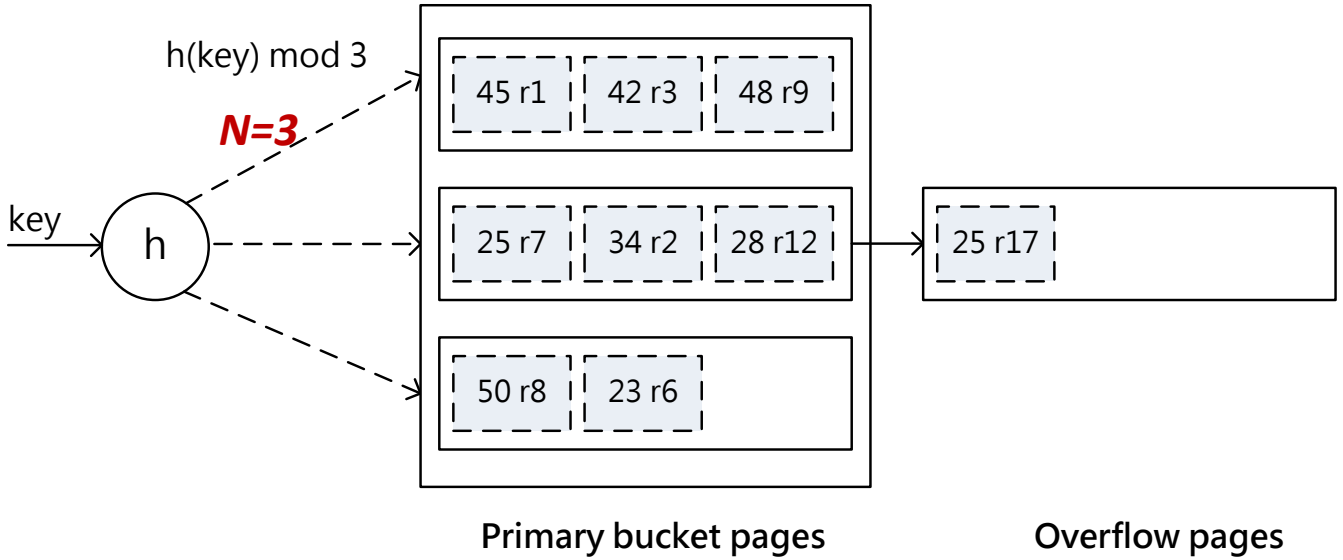
Hash-Based Indexes

- Good for equality selections
- Using a ***hashing function***, which maps values in a search key into a range of ***bucket*** numbers
- Bucket
  - Primary page plus zero or more overflow pages
- Static and dynamic hashing techniques



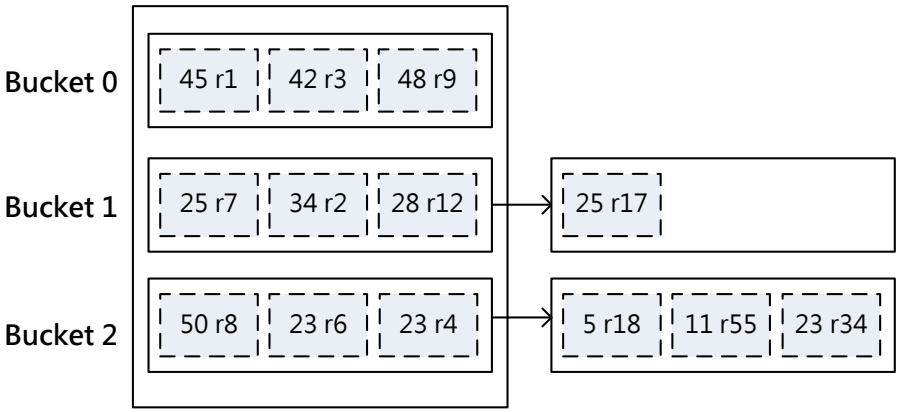
Static Hashing

- The number of bucket N is fixed
- Overflow pages if needed
- h(k) mod N = bucket to which data entry with key k belongs
- Records having the same hash value are stored in the same bucket



The Search Cost of Static Hashing

- How to compute the # of block access?
- If an index contains B blocks and has N buckets, then each bucket is about B/N blocks long



#rec = 13
rpb = 3
B = $\lceil 13/3 \rceil$=5
N = 3
#blockAccess = 2

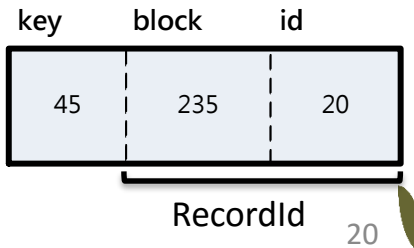
Hash Index in VanillaCore

- Related Package
  - `storage.index.hash.HashIndex`

HashIndex
<<final>> + NUM_BUCKETS : int
+ searchCost(ifldType : Type, totRecs : long, matchRecs : long) : long + HashIndex(ii : IndexInfo, fldtype : Type, tx : Transaction) + beforeFirst(searchRange : ConstantRange) + next() : boolean + getDataRecordId() : RecordId + insert(key : Constant, dataRecorId : RecordId) + delete(key : Constant, dataRecorId : RecordId) + close() + preLoadToMemory()

HashIndex

- This class stores each bucket in a separate table, whose name is the {index-name}{bucket-num}
  - e.g., indexdeptonstu25
- The method `beforeFirst` hashes the search key and opens a record file for the resulting bucket
- The index record [key, blknum, id]

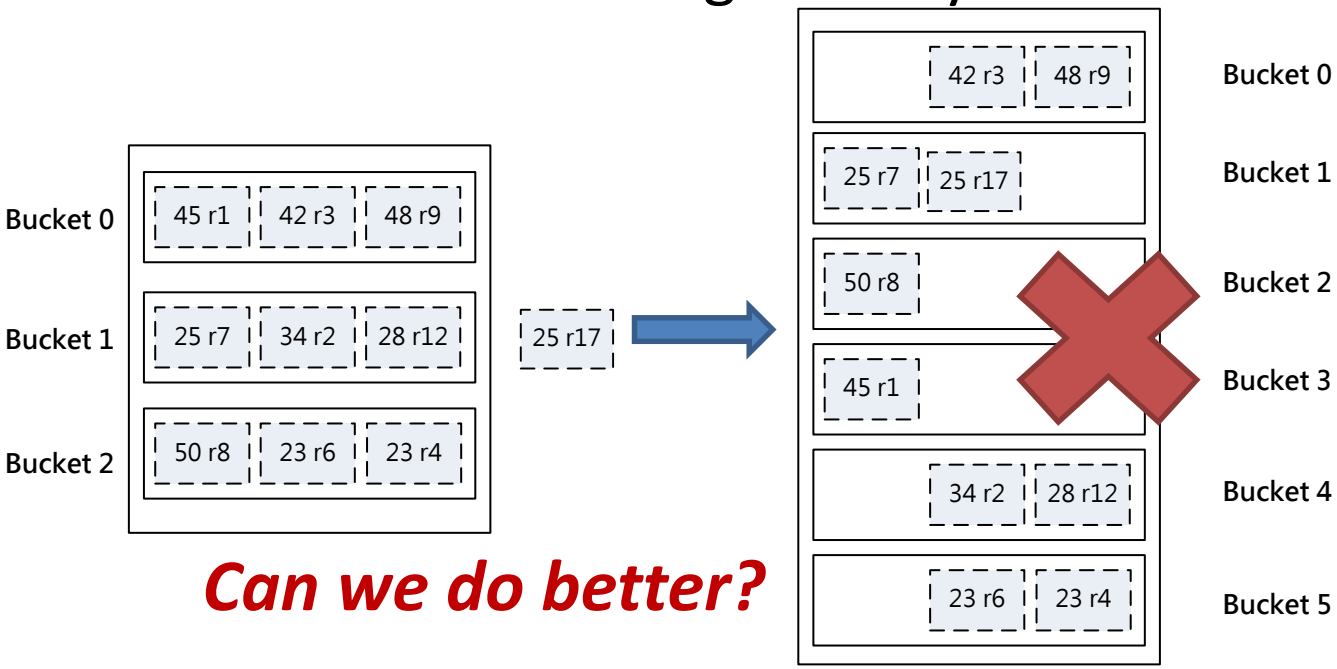


Number of Bucket and Hash Indexes

- If we can maintain each bucket with only one page, it result in efficient index access
- The search cost of static hashing index is inversely proportional to # of bucket
  - B/N
- The large # of bucket will create a lot of wasted space until the index grows into it

Number of Bucket and Hash Indexes

- Hard to choose # of bucket and maintain 1 page/bucket
- How about double the # of bucket when bucket becomes full?
  - Redistribute static hashing is costly



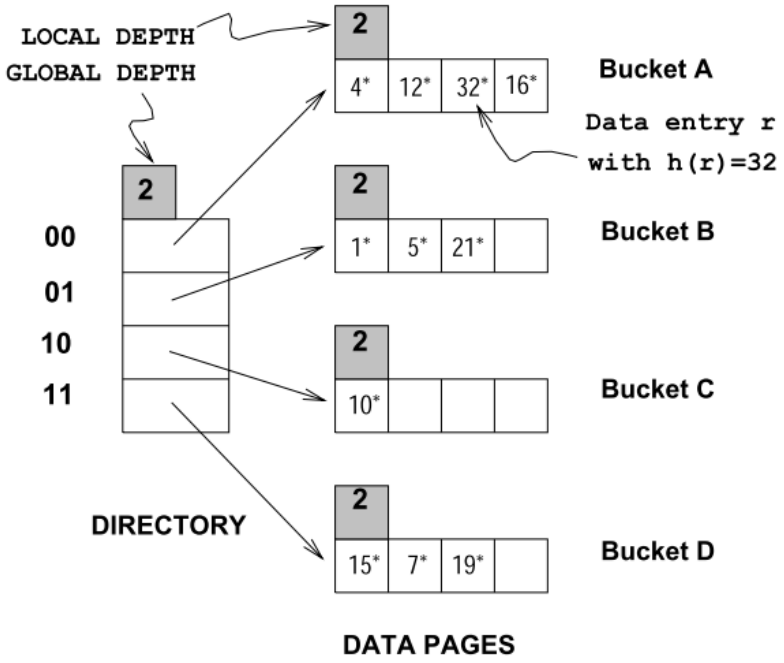
Can we do better?

Extendable Hash Indexes

- Main idea
  - Use ***directory*** of pointers to buckets, double # of buckets by doubling the directory, splitting just the bucket that overflowed
- Directory much smaller than file, so doubling it is much cheaper
- Only one page of data entries is split

Extendable Hash Indexes

- Directory is array of size 4
- To find bucket for r, take last `global depth' # bits of h(r); we denote r by h(r)

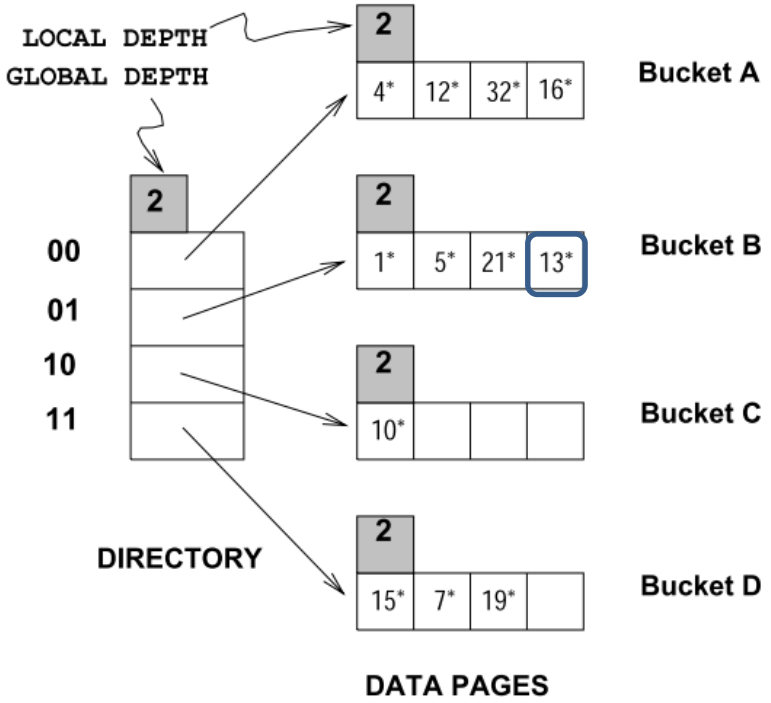


Global depth of directory:
Max # of bits needed to tell which bucket an entry belongs to

Local depth of a bucket:
# of bits used to determine if an entry belongs to this bucket

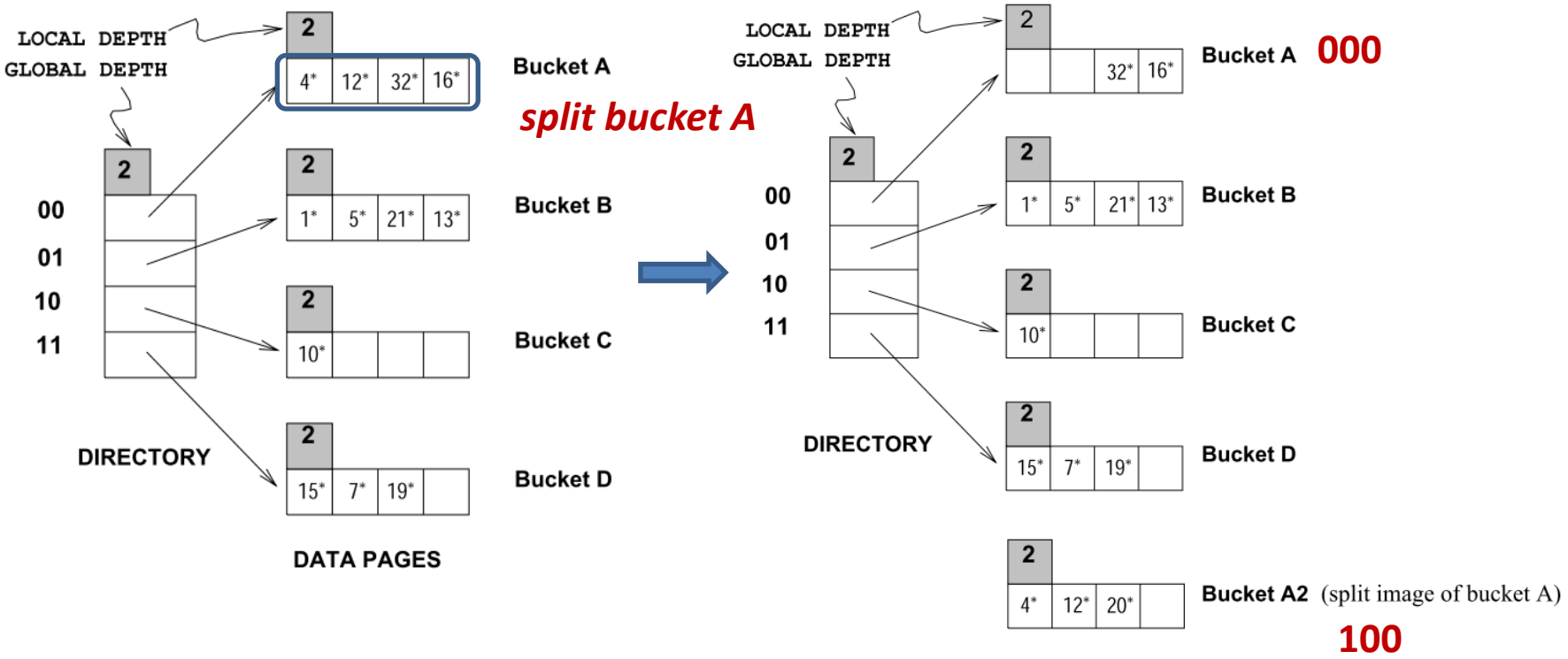
Example of an Extendible Hashed File

- After Inserting Entry r with h(r)=13
  - Binary number: 11**01**



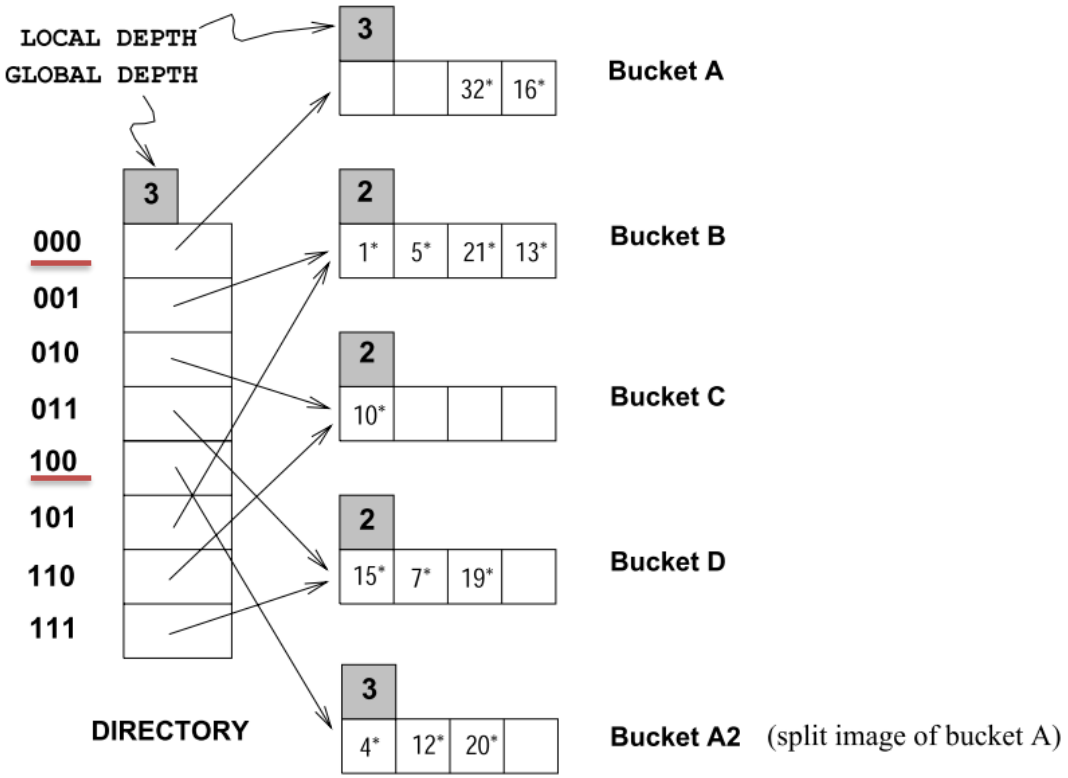
Example of an Extendible Hashed File

- While Inserting Entry r with h(r)=20
  - Binary number: 10100



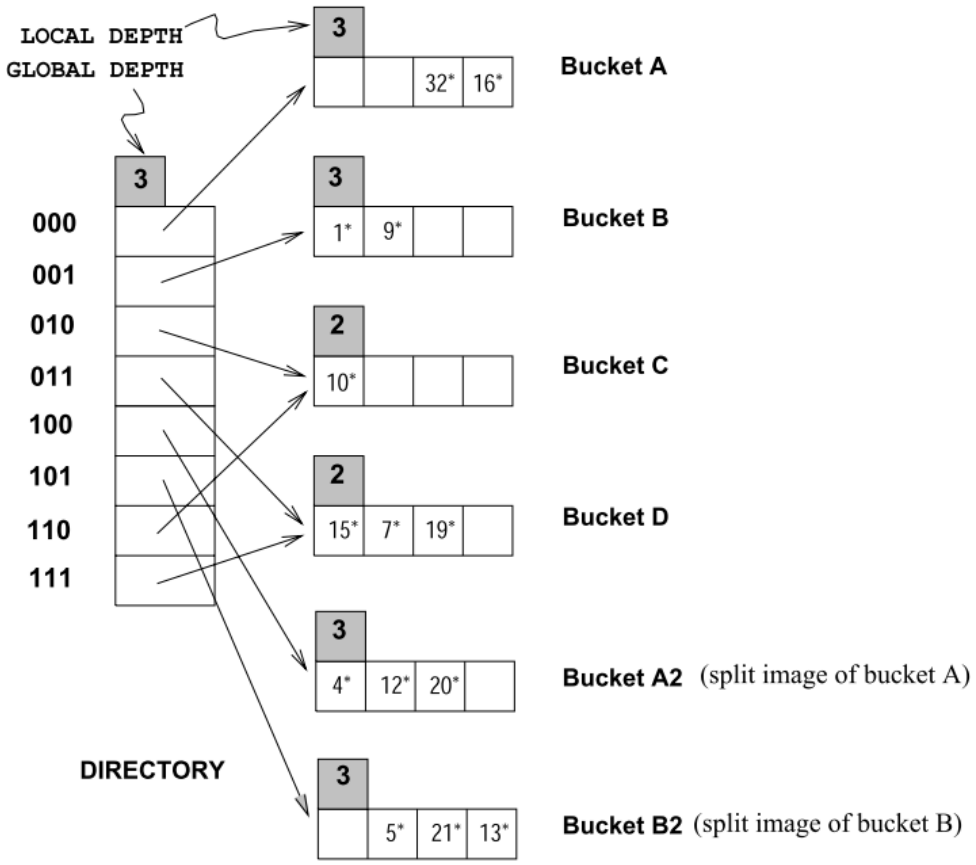
Example of an Extendible Hashed File

- After Inserting Entry r with h(r)=20
- Update the global depth
  - Some bucket will has local depth less than global depth



Example of an Extendible Hashed File

- After Inserting Entry r with h(r)=9



Remarks

- When does bucket split cause directory doubling?
  - Before insert, local depth of bucket = global depth. Insert causes local depth to become > global depth
- Directory is doubled by copying it over and `fixing' pointer to split image page
- No overflow page?
  - A lot of records with same key value

Outline

- Overview
- The API of Index in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Related Relational Algebra and Update Planner
- Transaction management revisited

Is Hash-Based Index Good Enough?

- Hash-based indexes are best for equality selections
  - Cannot support ***range searches***
  - e.g., `dept>100`
- We now consider an index structured as a ***search tree***
  - Speed up search by ***sorting*** leaf node values
- These structures provide efficient support for range and equality searches

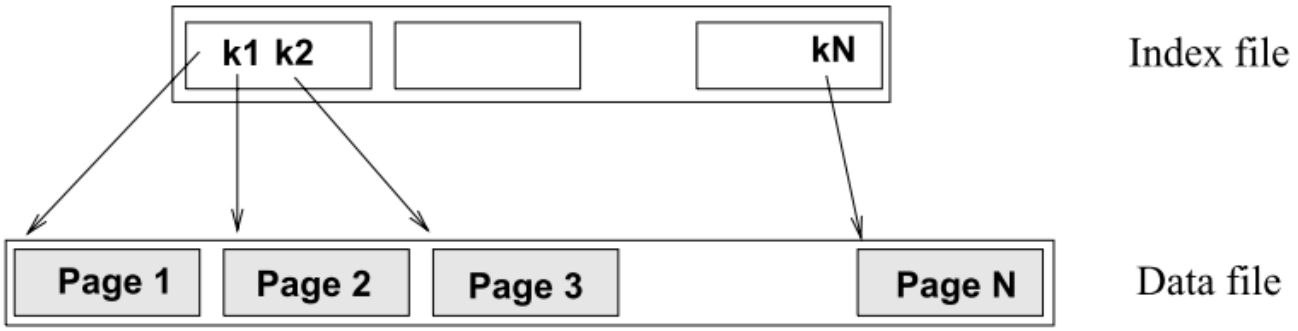
Power of Sorting

- "Find all students with dept > 100"
  - If data file is sorted on 'dept', do ***binary search*** to find first such student, then scan to find others
- Cost of binary search can be quite high if the data file is large

- Can we improve upon this method?

Intuition for Tree Indexes

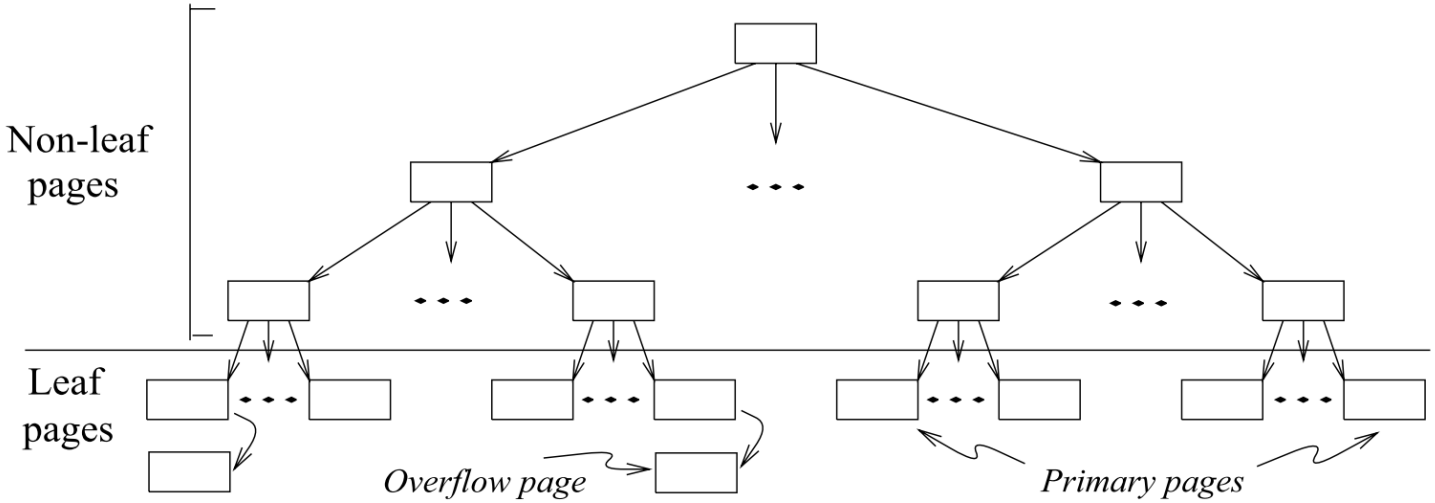
- Create an “index” file
  - Do the binary search on (smaller) index file
- What if there are too many key values in index file?
  - The index file is still large enough to make inserts and deletes expensive



One-Level Index Structure

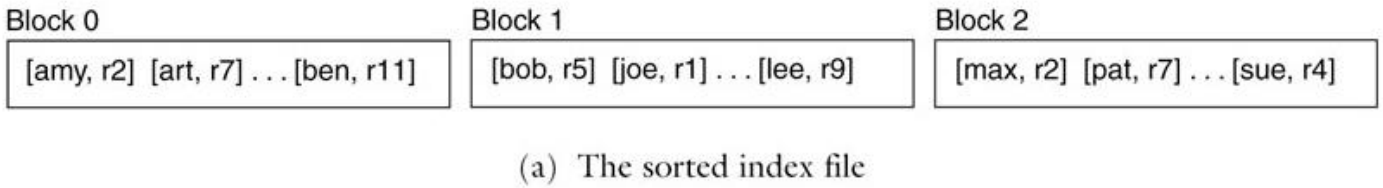
Intuition for Tree Indexes

- Why not apply the previous step of building an auxiliary file on the index file and so on ***recursively*** until the final auxiliary file fits on one page?



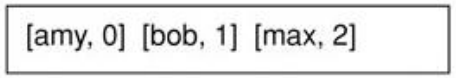
B-tree Index

- The most widely used index
- Balanced tree---all paths from root to leaf are of the same length
- An index for '`sname`' of `students` table



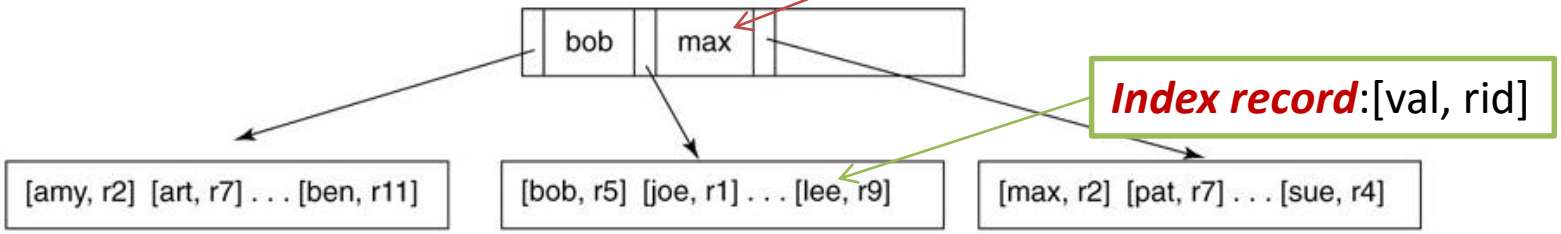
(a) The sorted index file

The records are sorted on dataval



(b) The sorted level-0 directory

Directory record:[val, blkNum]

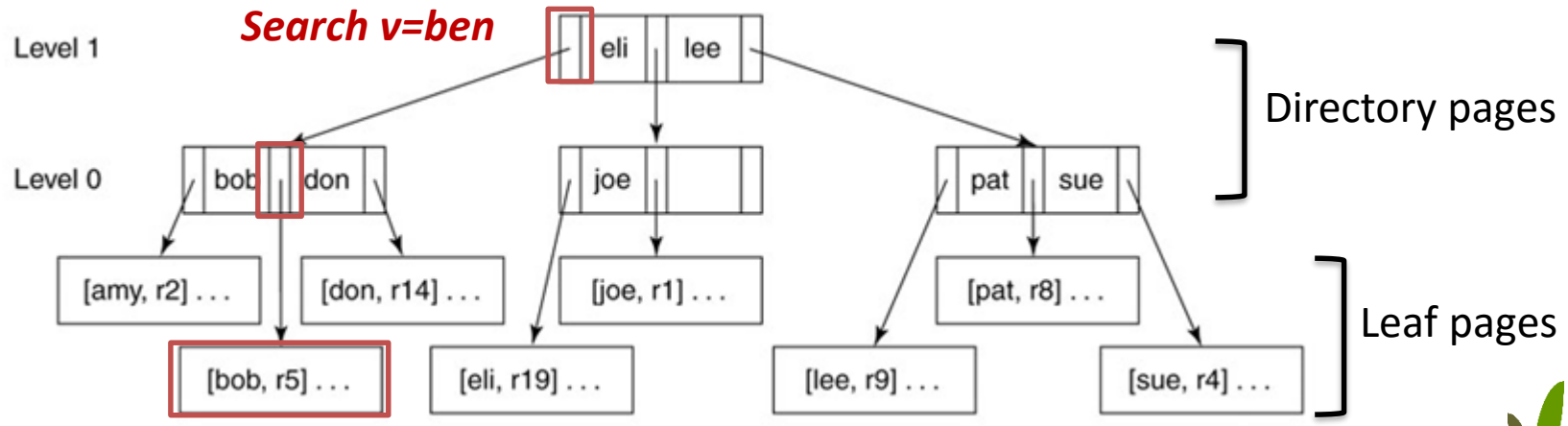


Index record:[val, rid]

(c) The tree representation of the index and its directory

Searching the B-tree Index

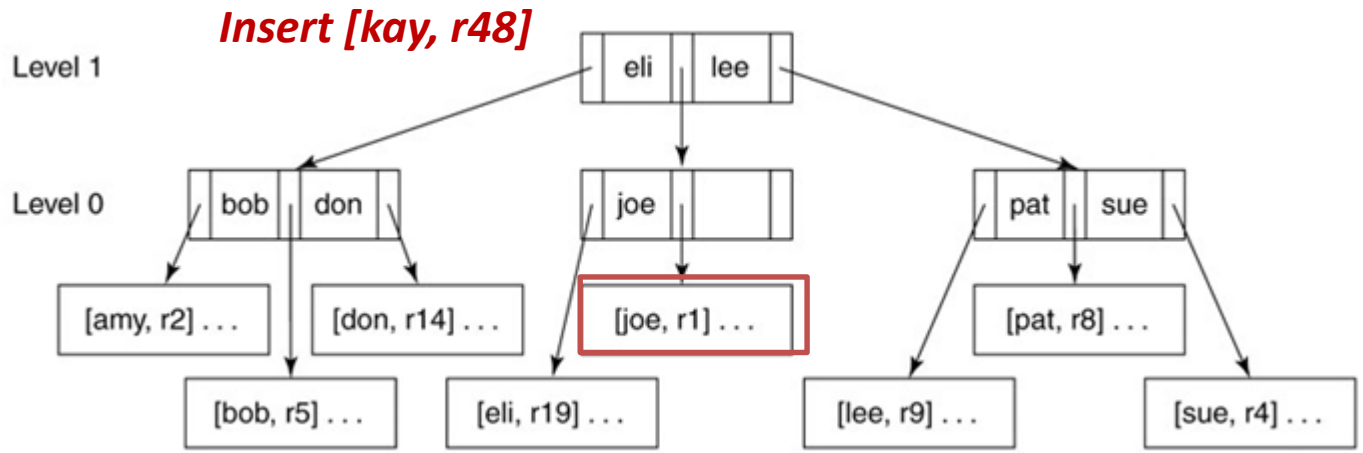
- Finding the index records having a specified dataval v
- Search begins at root, and key comparisons direct it to a leaf
- Search cost: the height of the tree



Inserting Record

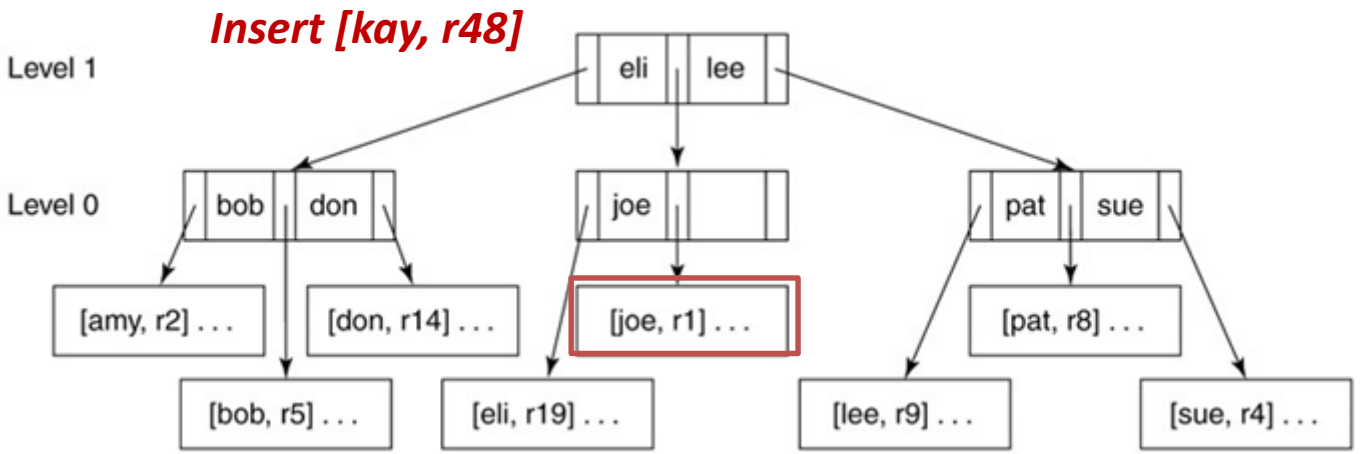
1. Search the index with the inserted dataval
2. Insert the new index record into the target leaf block

- What if the block has no more room?
  - Think about the extendable hashing. ***Spilt it***!



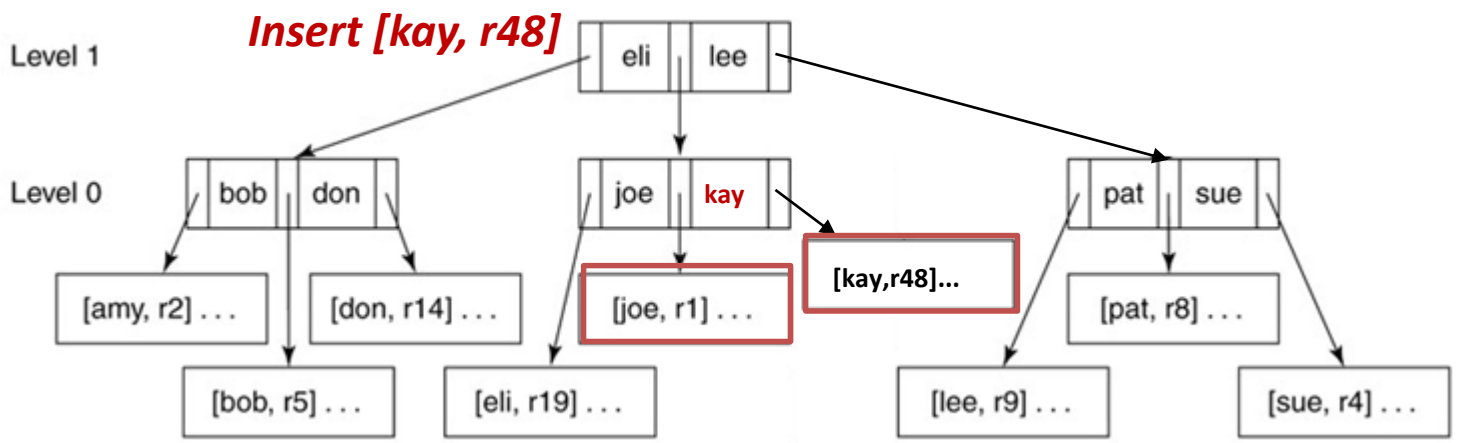
Splitting an Index Block

1. Allocate a new block in the index file
2. Move the high-valued half of the index record into this new block
3. Create a directory record for the new block
4. Insert the new directory record into the same level-0 directory block



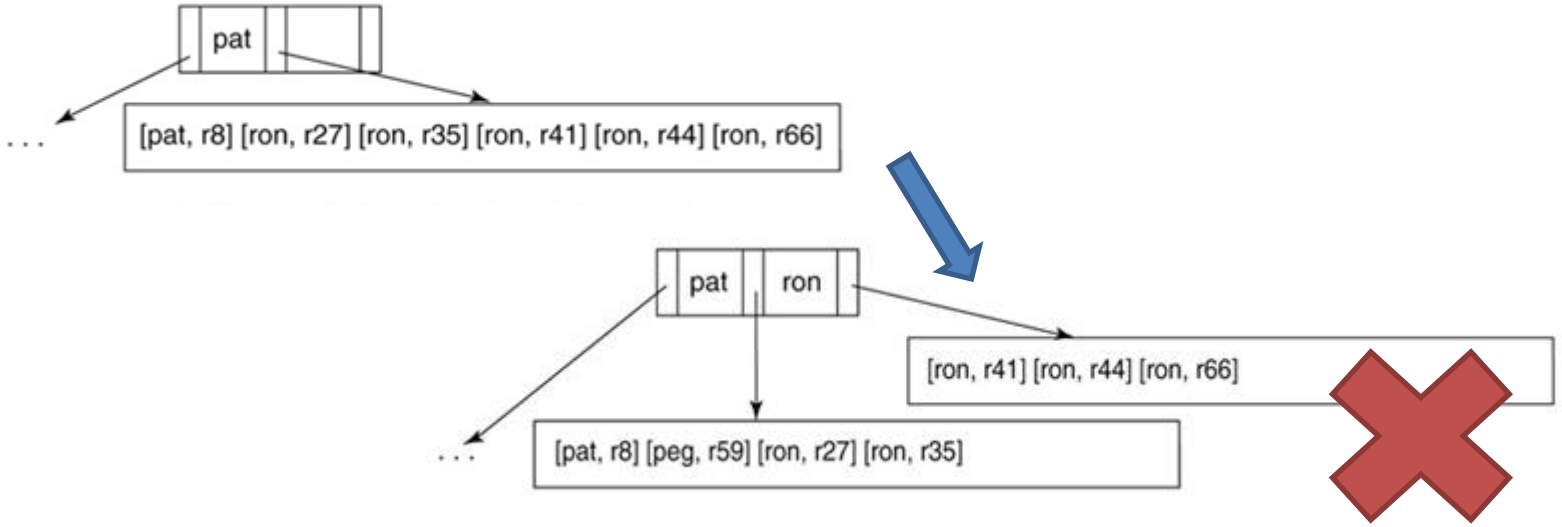
Splitting an Index Block

1. Allocate a new block in the index file
2. Move the high-valued half of the index record into this new block
3. Create a directory record for the new block
4. Insert the new directory record into the same level-0 directory block



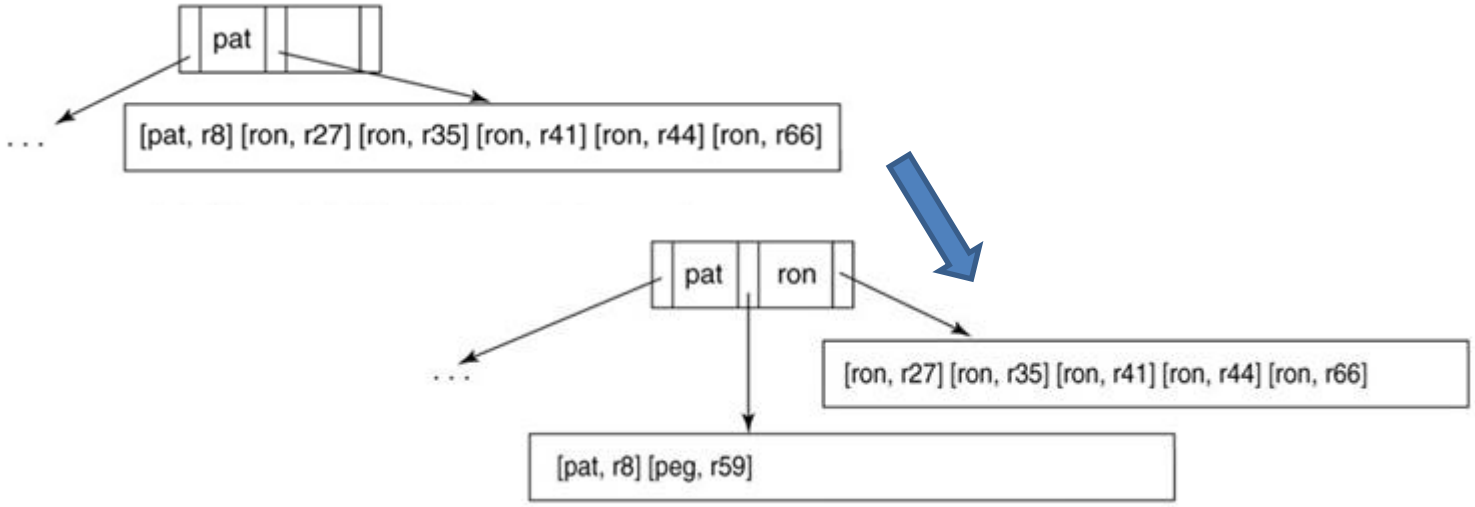
Duplicate Datavals

- What if too many index records have the same dataval?
- When splitting a block, you must place all records having the same dataval in the same block



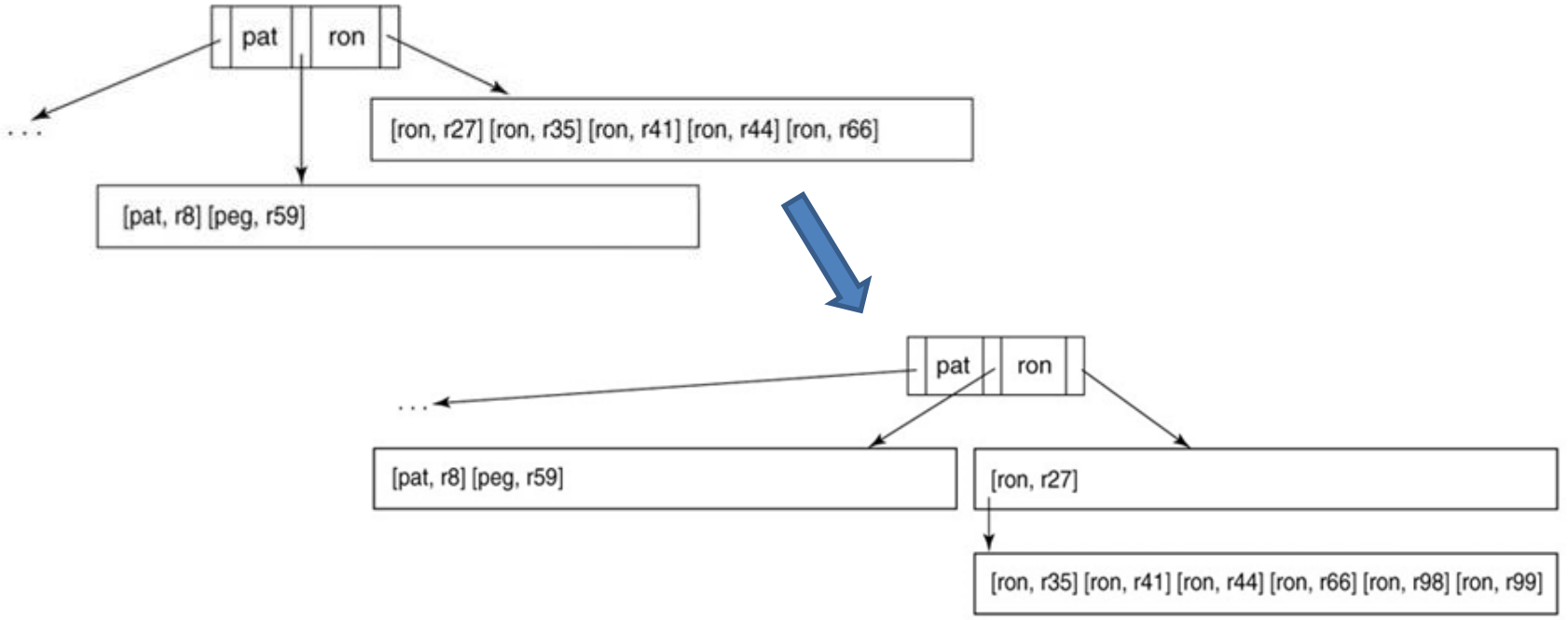
Duplicate Datavals

- What if too many index records have the same dataval?
- When splitting a block, you must place all records having the same dataval in the same block



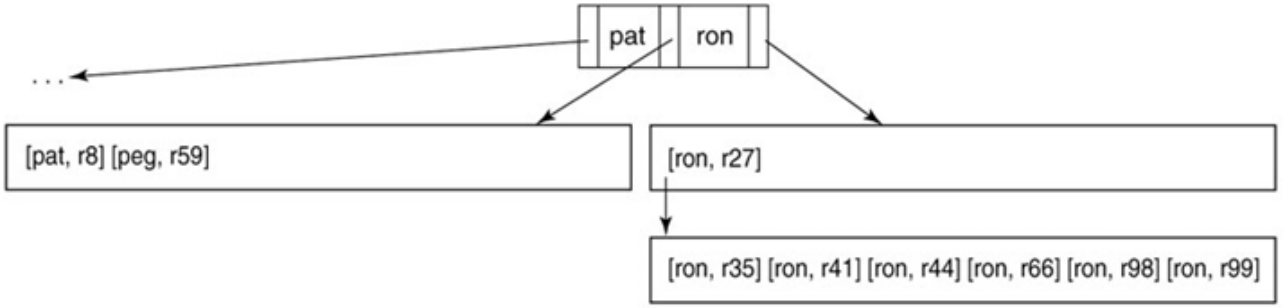
Duplicate Datavals

- Insert another index record [ron, r27]
  - The original block is full again
- Use the ***overflow block***



Duplicate Datavals

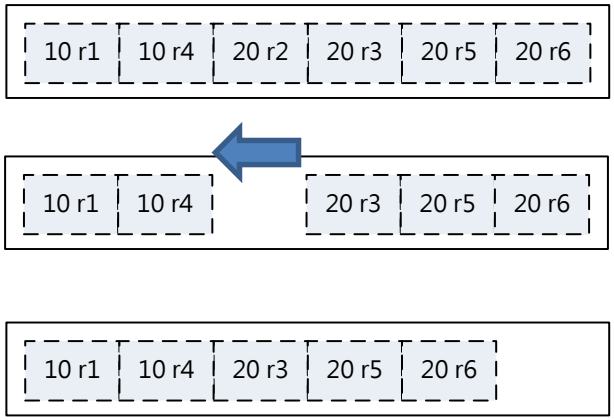
- We need to make sure the first record in the primary leaf block always having the same dataval as the records in overflow block
- When insert a index record [ray, r11]
  - Spilt the overflow block further



Deleting an Index Record

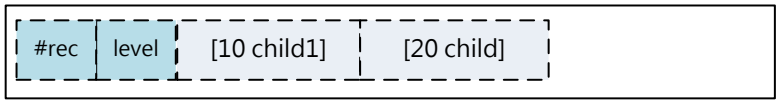
1. Search the index with the deleted dataval and datarid
2. Delete the index record in the target leaf block
3. Shift the index records

- Result in lot of record modification
- Merge the block if the # of record in a block is less than a predefined number

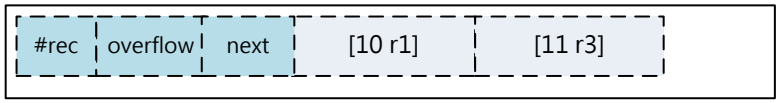


B-tree Index in VanillaCore

- Related package
  - `storage.index.btree`
- B-tree page
  - Directory pages



  - Leaf pages



Outline

- Overview
- The API of Index in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Related Relational Algebra and Update Planner
- Transaction management revisited

Related Relational Algebra

- Related package: `query.algebra.index`
- `IndexSelectPlan`
- `IndexJoinPlan`

Update Planner

- Related package: `query.planner.index`
- `IndexUpdatePlanner`

Outline

- Overview
- The API of Index in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Related Relational Algebra and Update Planner
- Transaction management revisited

Index Locking

- Why, given that we have S2PL already?
  - Can we just lock data objects (after index search)?
- No! You need to lock indices
- To ensure the consistency of the index structures
- To prevent phantom due to modification

Maintaining Structure Consistency

- How?
- Naïve: simply s-/x-lock on an index
- But an index is one of the most frequently accessed meta-structures in a DBMS
- Can you improve the performance?
- Idea: early lock release

Specialized Locking Protocols

- Data access with a static hash index:
  - S-/X-lock on the bucket file
  - Perform index lookup/insert/delete
  - ***Release the index locks***
  - S-/X-lock on data object
  - Perform data access insert/delete
  - Hold the data locks following S2PL

Specialized Locking Protocols

- Data access with a B-tree index:
  - ***Crab-locking*** along the B-tree
  - Perform index lookup/insert/delete
  - ***Release the leaf locks***
  - S-/X-lock on data object
  - Perform data access insert/delete
  - Hold the data locks following S2PL
- Deadlock free

How about Phantom due to Updates?

- Idea: hold the lock of B-tree leave until tx end
- Limitation: only prevents phantoms due to single-table updates
- Be careful about deadlock!
  - This protocol is no longer deadlock free
  - A better deadlock handling is required

Recovery

- Since locks are released early, logical logging and recovery is required

You Have Assignment!

Assignment: Preventing Update Phantoms

- Modify index locking protocol to prevent phantoms due to updates
- Hint: revisit lock mode and data access path
  - No update phantom in SERIALIZED isolation mode
  - Other isolation modes need to be compatible with SERIALIZED mode