Indexing

Shan-Hung Wu
CS, NTHU

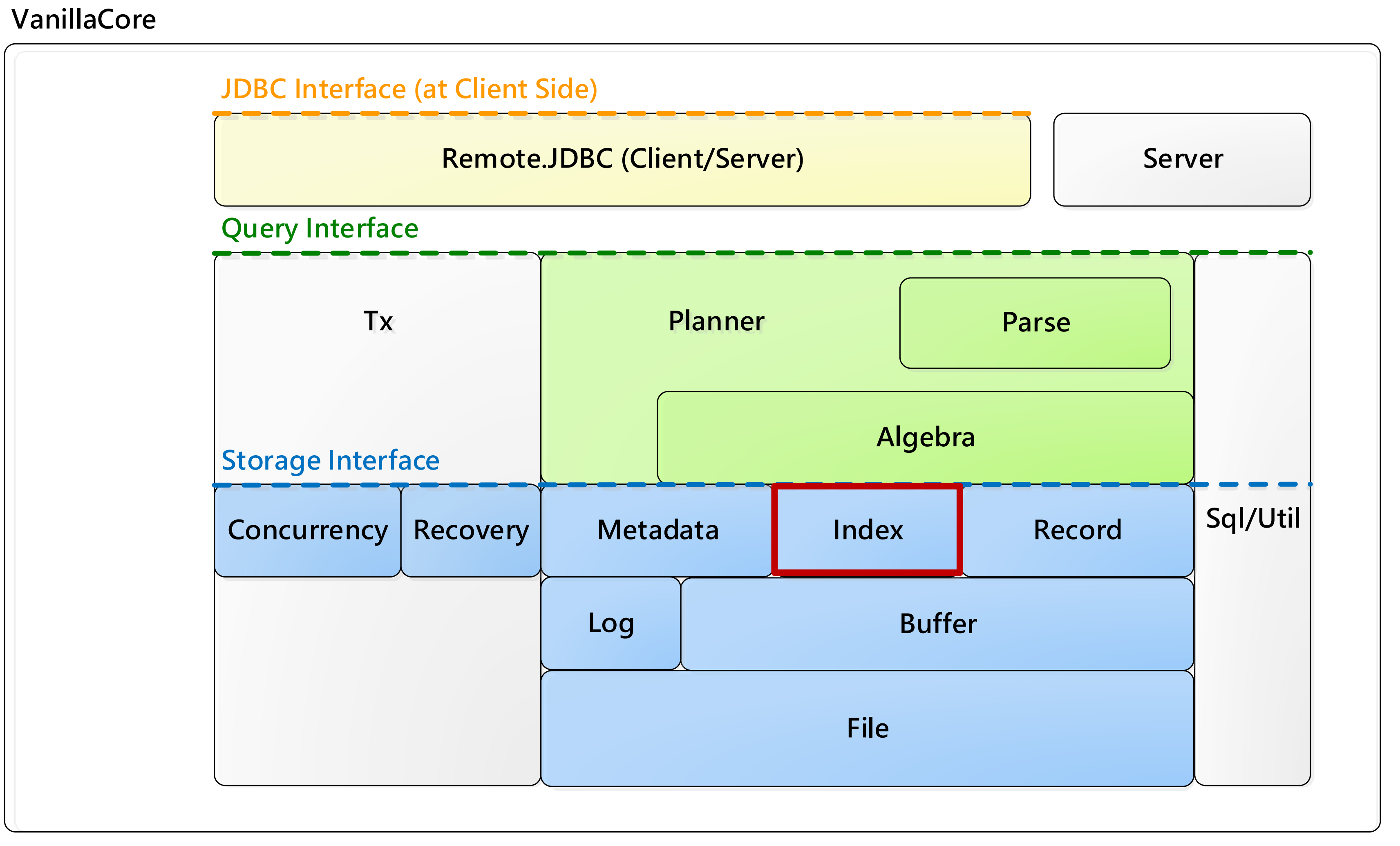
Outline

- Overview
  - API in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Query Processing
- Transaction Management

Outline

- Overview
  - API in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Query Processing
- Transaction Management

Where are we?



VanillaCore
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Tx
Planner
Parse
Algebra
Storage Interface
Concurrency
Recovery
Metadata
Index
Record
Sql/Util
Log
Buffer
File

Why Index?

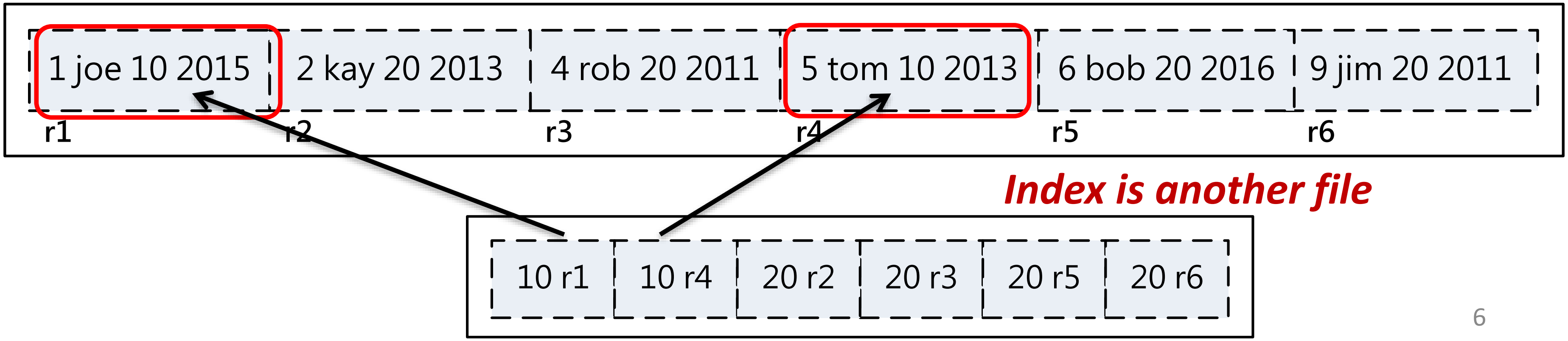
- Query:
  - `SELECT * FROM students WHERE dept = 10`
- Record file for `students`:

1 joe 10 2015	2 kay 20 2013	4 rob 20 2011	5 tom 10 2013	6 bob 20 2016	9 jim 20 2011
r1	r2	r3	r4	r5	r6

- Selectivity is usually low
- Full table scan results in poor performance

What is an Index?

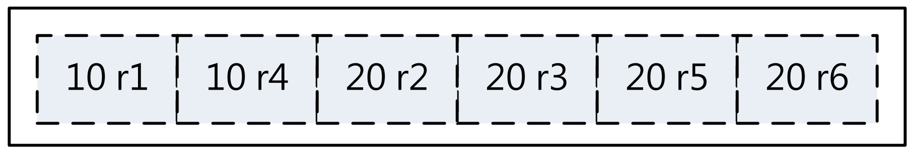
- Query:
  - `SELECT * FROM students WHERE dept = 10`
- ***Index***: a data structure (file) defined on ***fields*** that speeds up data accessing
  - Input: field values or ranges
  - Output: rids



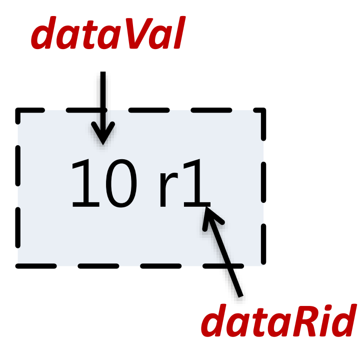
Terminology (1/2)

- Every index has an associated ***search key***
  - I.e., one or more fields

Search key: `dept`



- ***Primary index*** vs. ***secondary index***
  - If search key contains primary key or not
- Index entry/record:
  - <data value, data rid>



Terminology (2/2)

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

Outline

- Overview
  - API in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Query Processing
- Transaction Management

SQL Statements for Index Creation

- The SQL:1999 standard does not include any statement for creating or dropping indeice
- Creating index:
  - `CREATE INDEX <name> ON <table>(<fields>) USING <method>`
  - E.g., `CREATE INDEX idxdept ON students(dept) USING btree`
- In VanillaCore, an index only supports ***one*** indexed field

The `Index` Class in VanillaCore

- An abstract class in `storage.index`
  - `beforeFirst()` resets iterator and search value
  - `next()` moves to the next rid ***matching search value***

<<abstract>>
Index

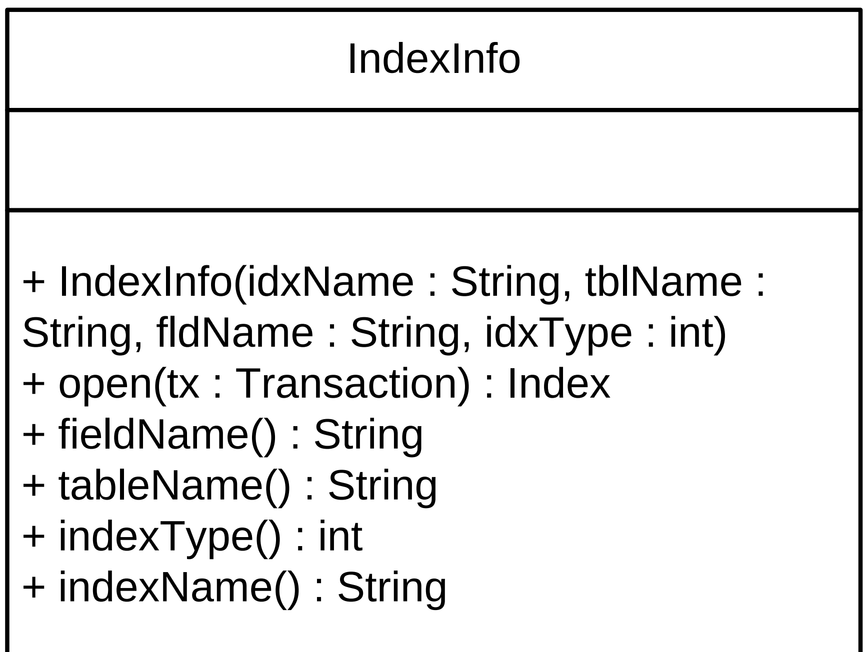
<<final>> + IDX_HASH : int
<<final>> + IDX_BTREE : int

+ searchCost(idxType : int, fldType : Type, totRecs : long, matchRecs : long) : long
+ newIntance(ii : IndexInfo, fldType : Type, tx : Transaction) : Index

<<abstract>> + beforeFirst(searchkey : ConstantRange)
<<abstract>> + next() : boolean
<<abstract>> + getDataRecordId() : RecordId
<<abstract>> + insert(key : Constant, dataRecordId : RecordId)
<<abstract>> + delete(key : Constant, dataRecordId : RecordId)
<<abstract>> + close()
<<abstract>> + preLoadToMemory()

IndexInfo

- Factory class for `Index` via `open()`
- Stores information about an index
- Similar to `TableInfo`



Using an 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

- `INSERT INTO students (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();
```

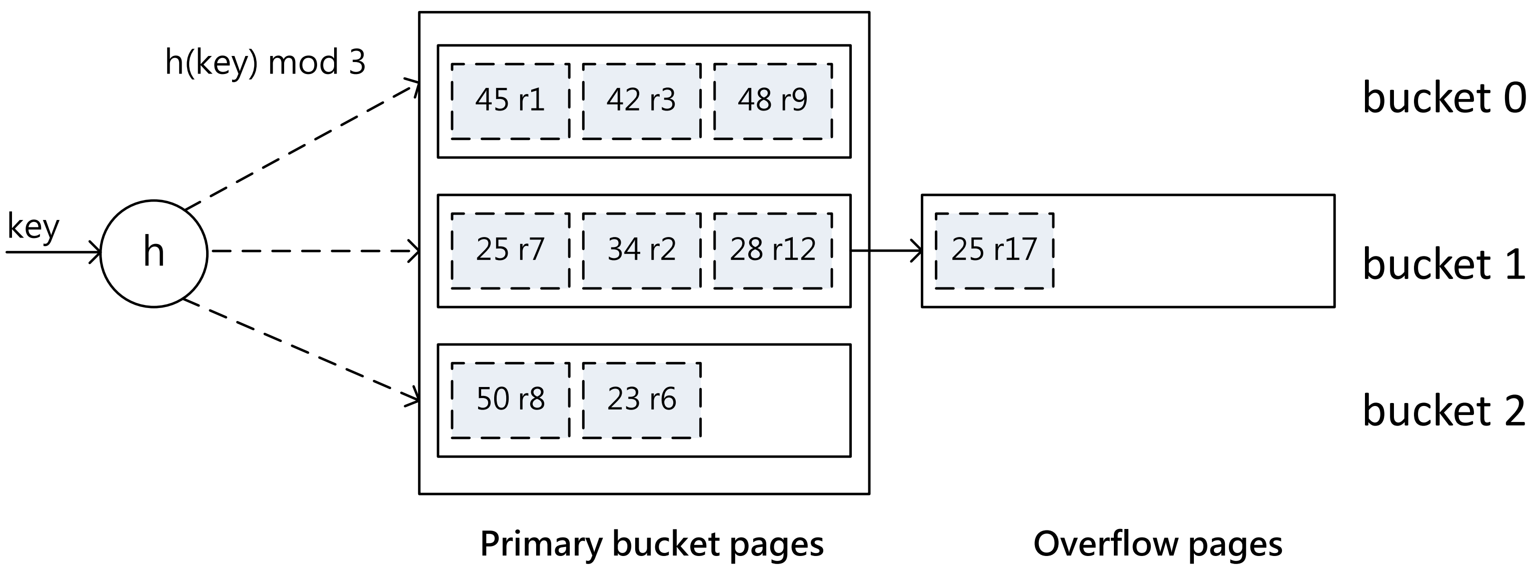
- ***Faster reads*** at the cost of ***slower writes***

Outline

- Overview
  - API in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Query Processing
- Transaction Management

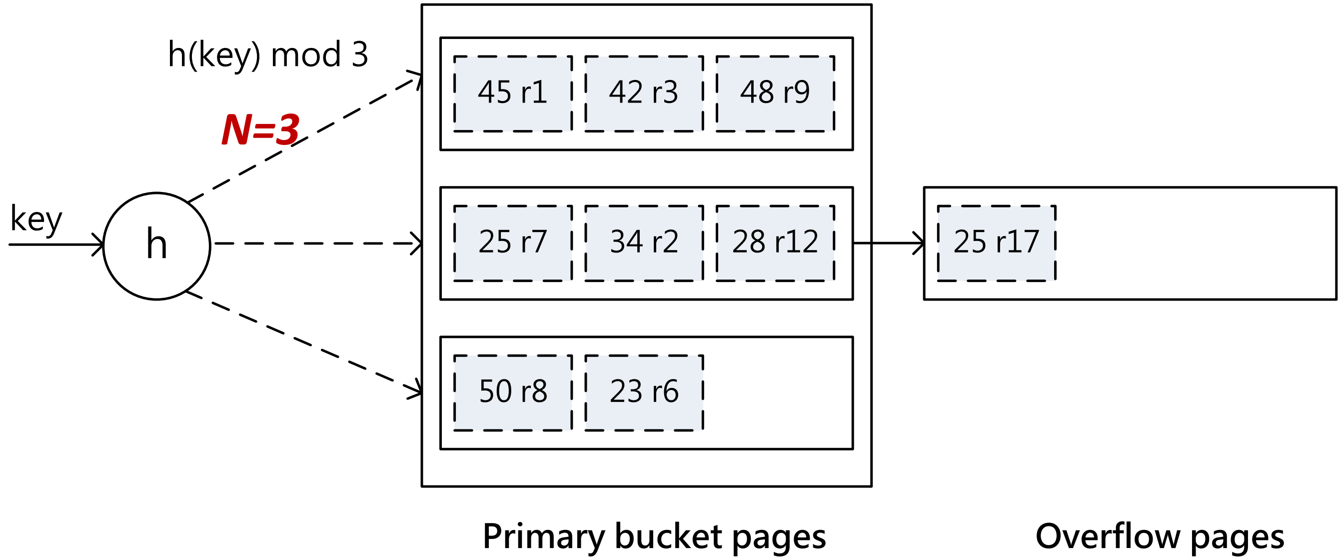
Hash-Based Indexes

- Designed for equality selections
- Uses a ***hashing function***
  - Search values → ***bucket*** numbers
- Bucket
  - Primary page plus zero or more overflow pages
- Based on static or dynamic hashing techniques



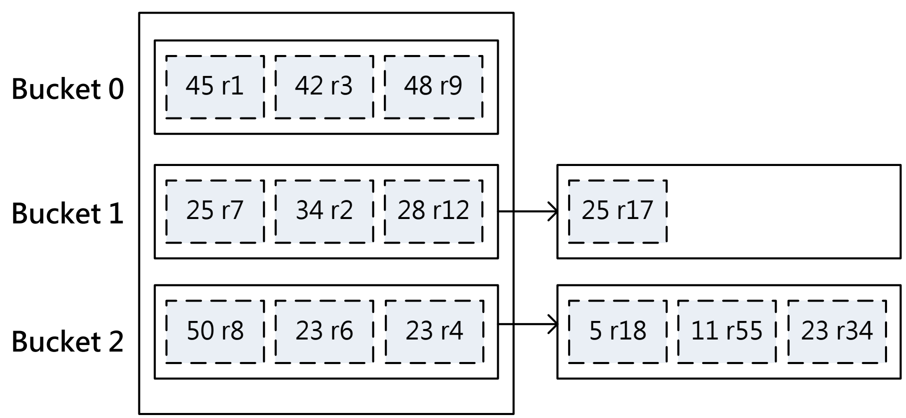
Static Hashing

- 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



Search Cost of Static Hashing

- How to compute the #block-access?
- Assume index has 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

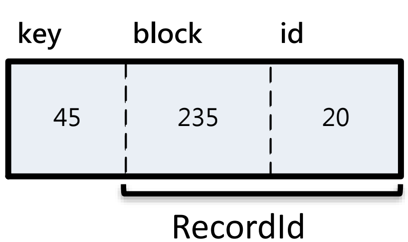
Hash Index in VanillaCore

- 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, dataRecordId : RecordId) + delete(key : Constant, dataRecordId : RecordId) + close() + preLoadToMemory()

HashIndex

- Stores each bucket in a record file
  - Name: {index-name}{bucket-num}
- `beforeFirst()`
  1. Hashes the search value, and
  2. Opens the corresponding record file
- The index record [key, blknum, id]

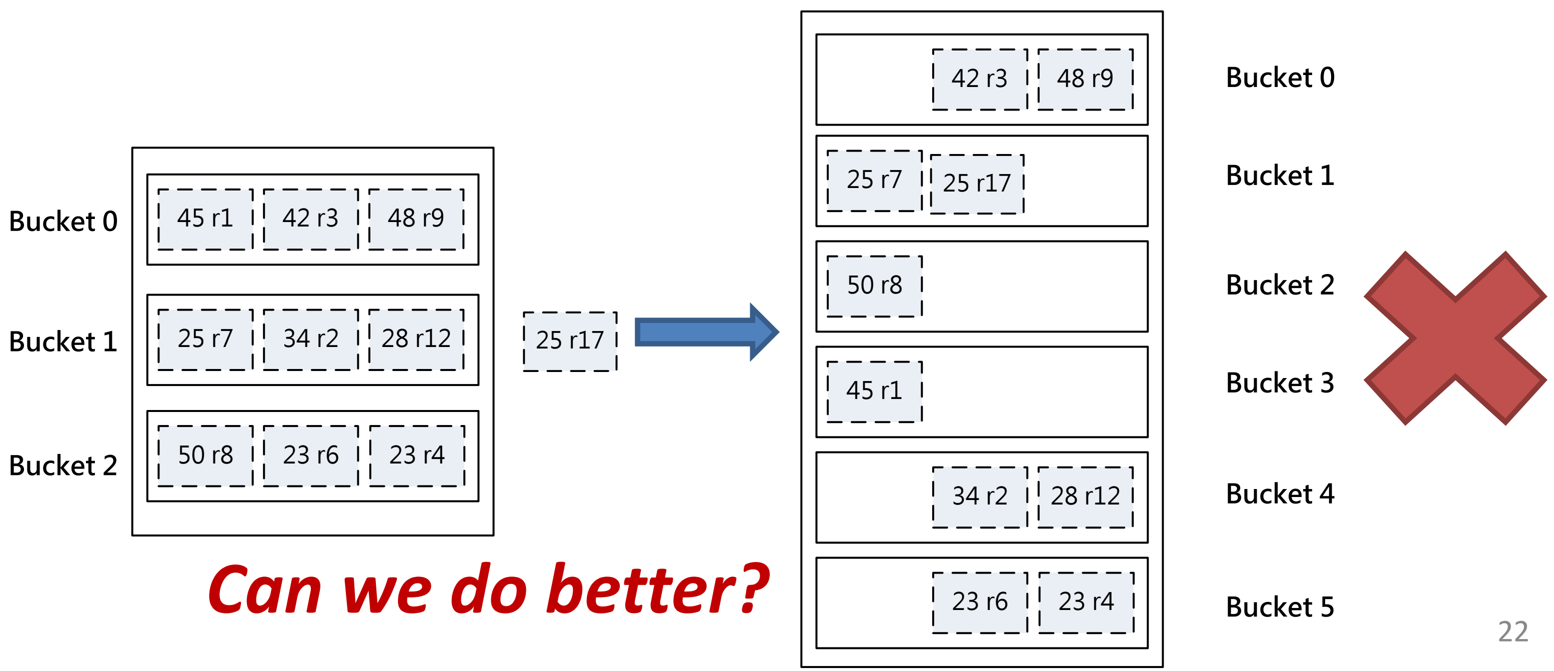


Limitations of Static Hashing (1/2)

- Search cost: B/N
- Increase efficiency → increase N (#buckets)
  - Best when 1 block per bucket
- However, a large #buckets leads to wasted space
  - Empty pages waiting the index to grow into it

Limitations of Static Hashing (2/2)

- Hard to decide N
- Why not double #buckets when a bucket is full?
  - Redistributing records is costly



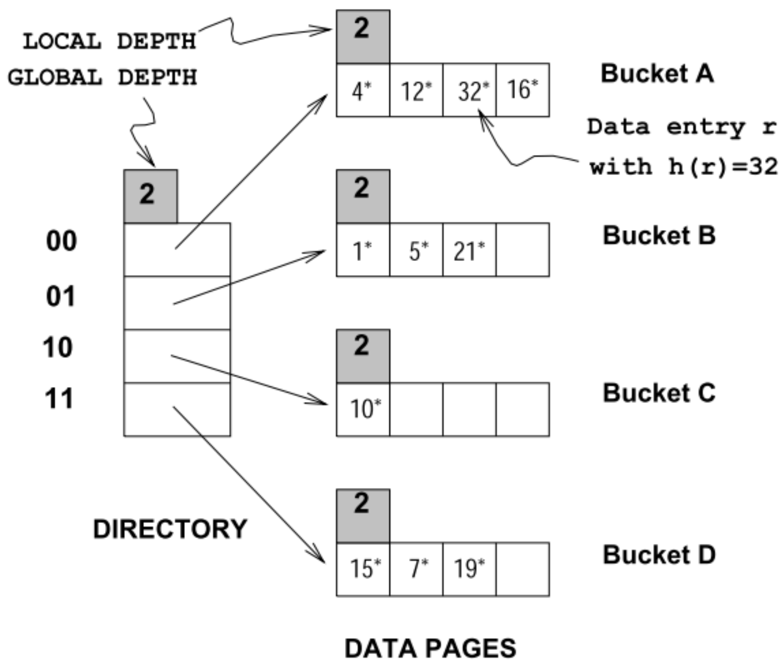
Can we do better?

Extendable Hash Indexes

- Use ***directory***: pointers to buckets
- Double #buckets by doubling the directory
- Splitting just the bucket that overflowed

Extendable Hash Indexes

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

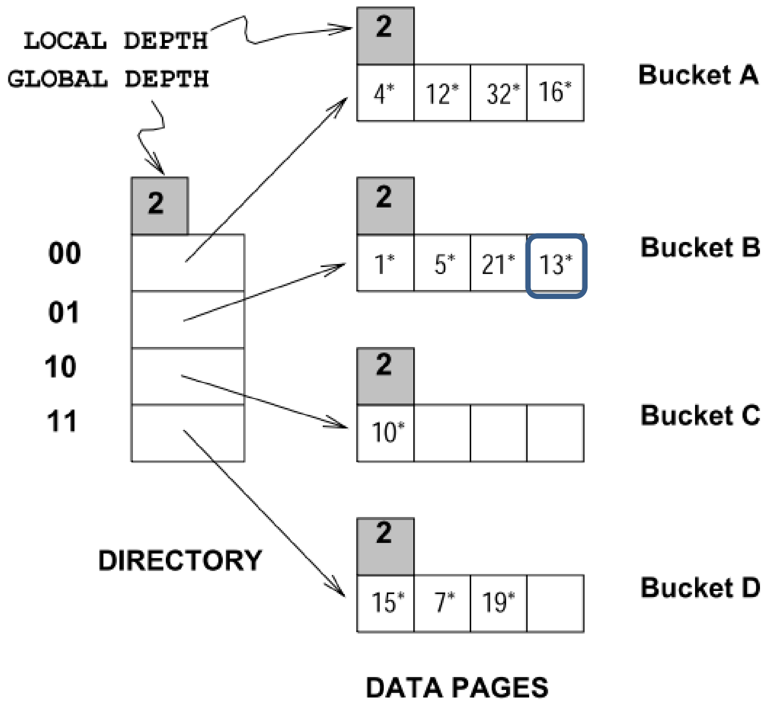


Global depth of directory:
Max #bits needed to tell
which bucket an entry belongs to

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

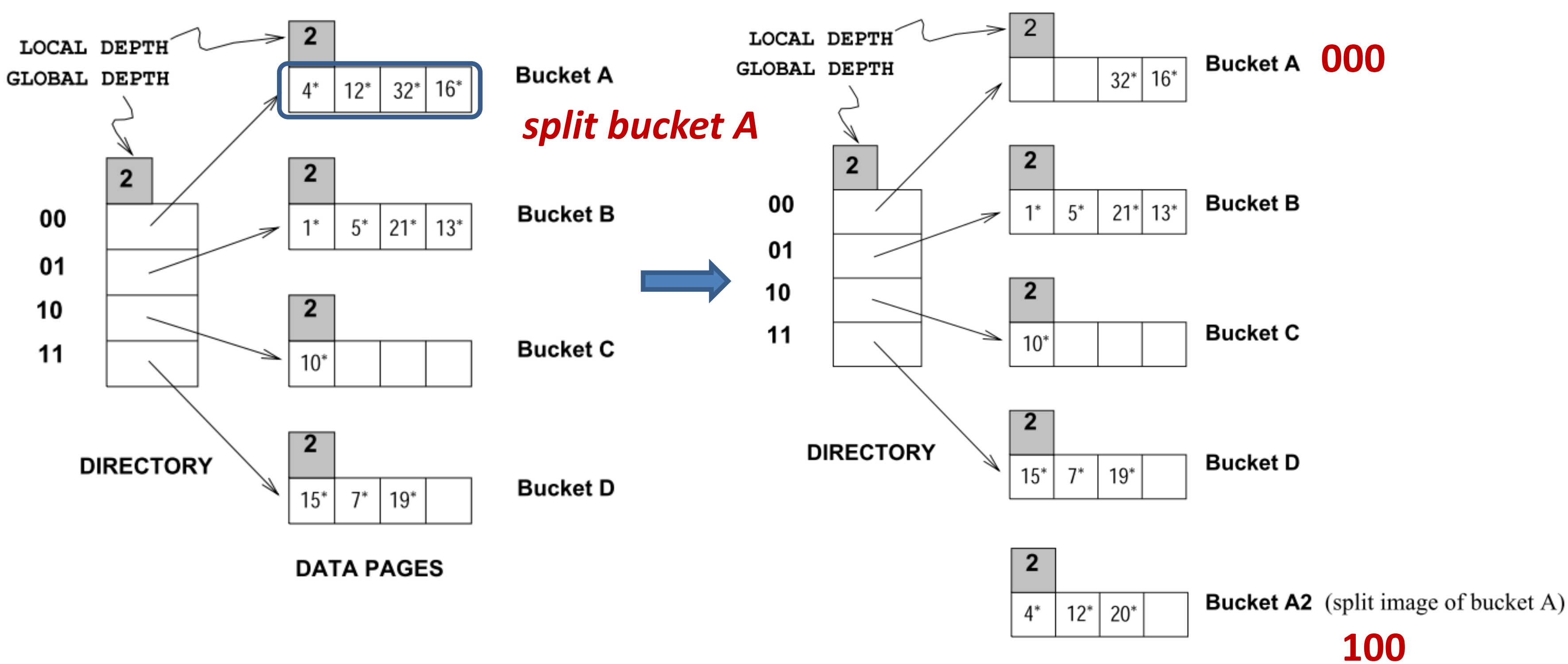
Example (1/4)

- After inserting entry r with h(r)=13
  - Binary number: 11**01**



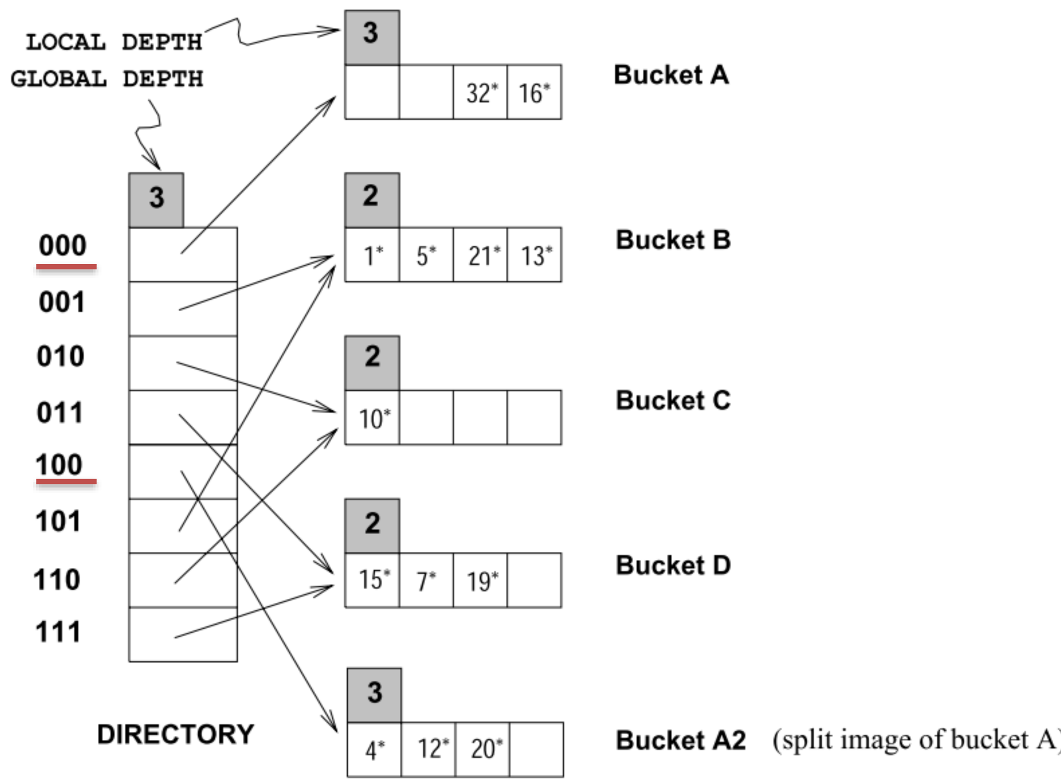
Example (2/4)

- While inserting entry r with h(r)=20
  - Binary number: 101**00**



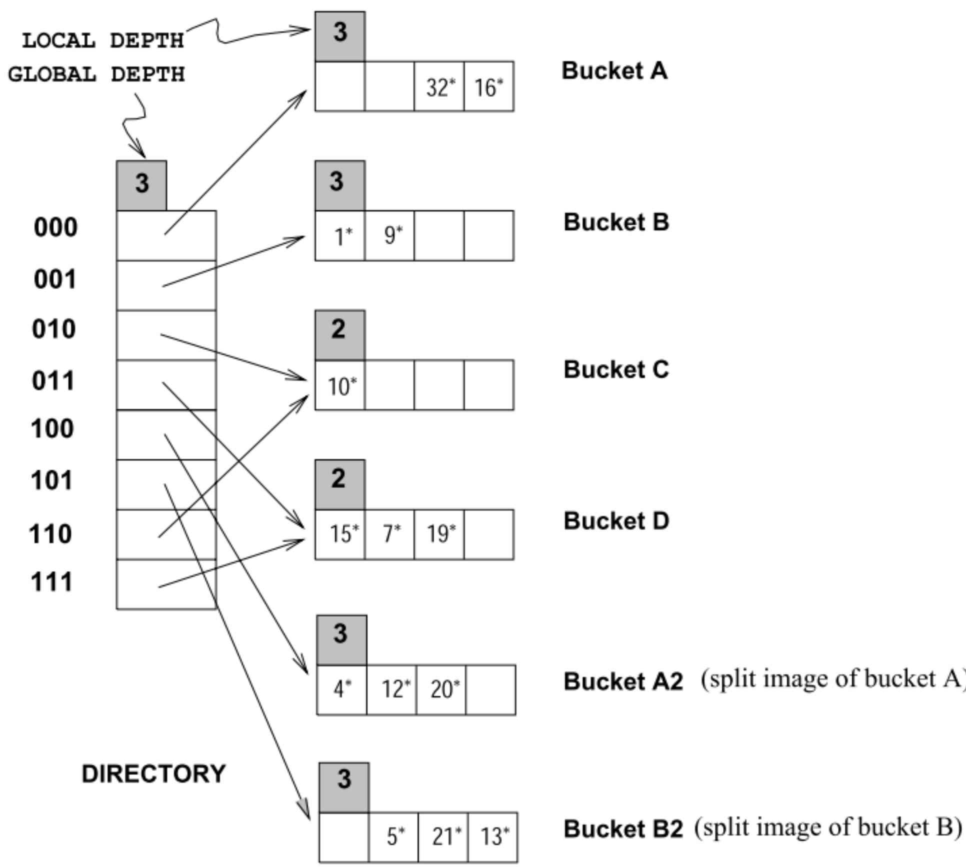
Example (3/4)

- After inserting entry r with h(r)=20
- Update the global depth
  - Some buckets will have local depth less than global depth



Example (4/4)

- After inserting entry r with h(r)=9



Remarks

- At most 1 page split for each insert
- Cheap doubling
  - When local depth of bucket = global depth
  - Only 3 page access (1 directory page, 2 data pages)

- No overflow page?
  - Still has, but only when there are a lot of records with same key value

Outline

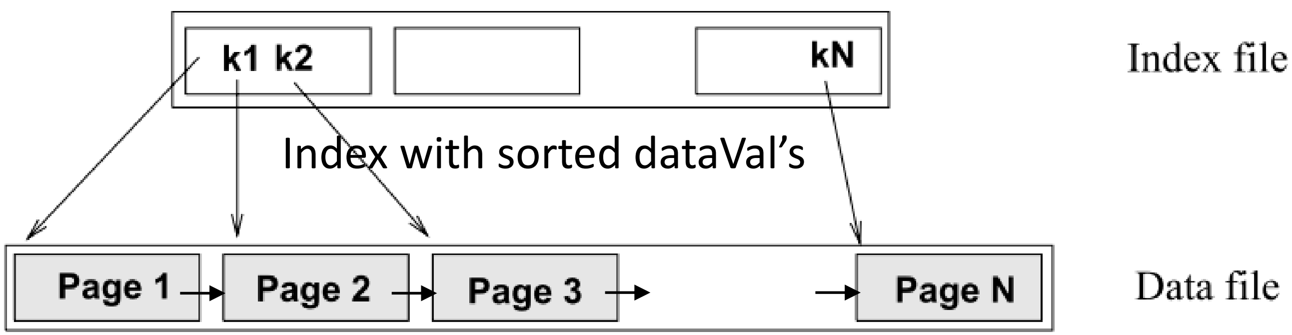
- Overview
  - API in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Query Processing
- Transaction Management

Is Hash-Based Index Good Enough?

- Hash-based indexes are good for equality selections
- However, cannot support ***range searches***
  - E.g., `... WHERE dept>100`
- We now consider an index structured as a ***search tree***
  - Speeds up search by ***sorting*** values
  - Supports ***both*** range and equality searches

Power of Sorting

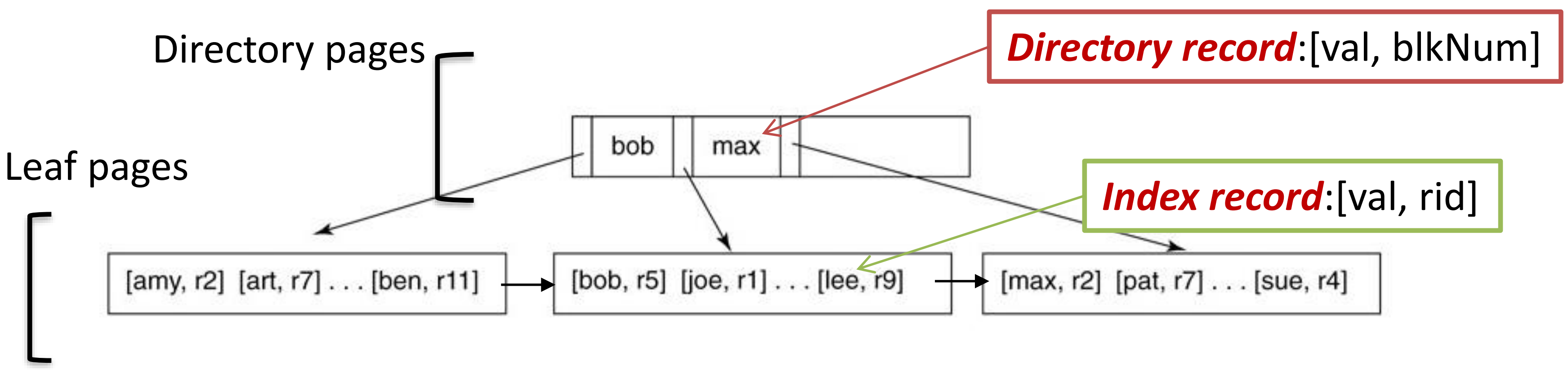
- Create an "index" file
  - where `dataVal`'s are sorted
- Query: "Find all students with dept > 100"
  - Do ***binary search*** to find first such student, then scan the index till end to find others



- However, slow update: O(#data-records)

B-Tree Index

- The most widely used index
- Index records are sorted on dataVal in each page
- M-way balanced search tree:
  - $O(\log_M(\#\text{data-records}))$ for equality search & update
  - $O(\#\text{data-records})$ for range search

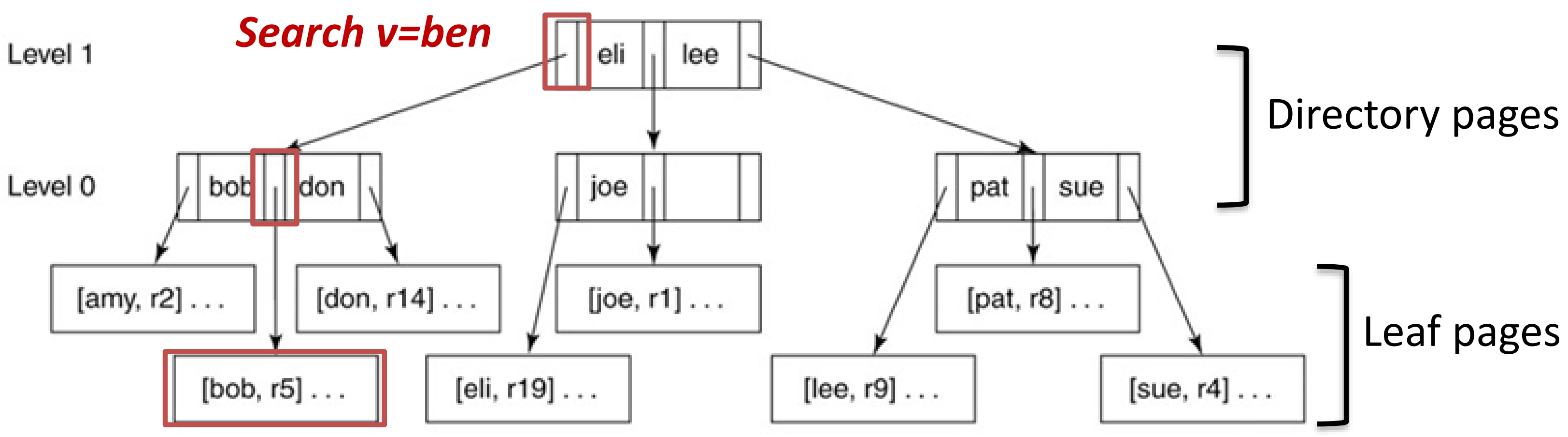


Searching

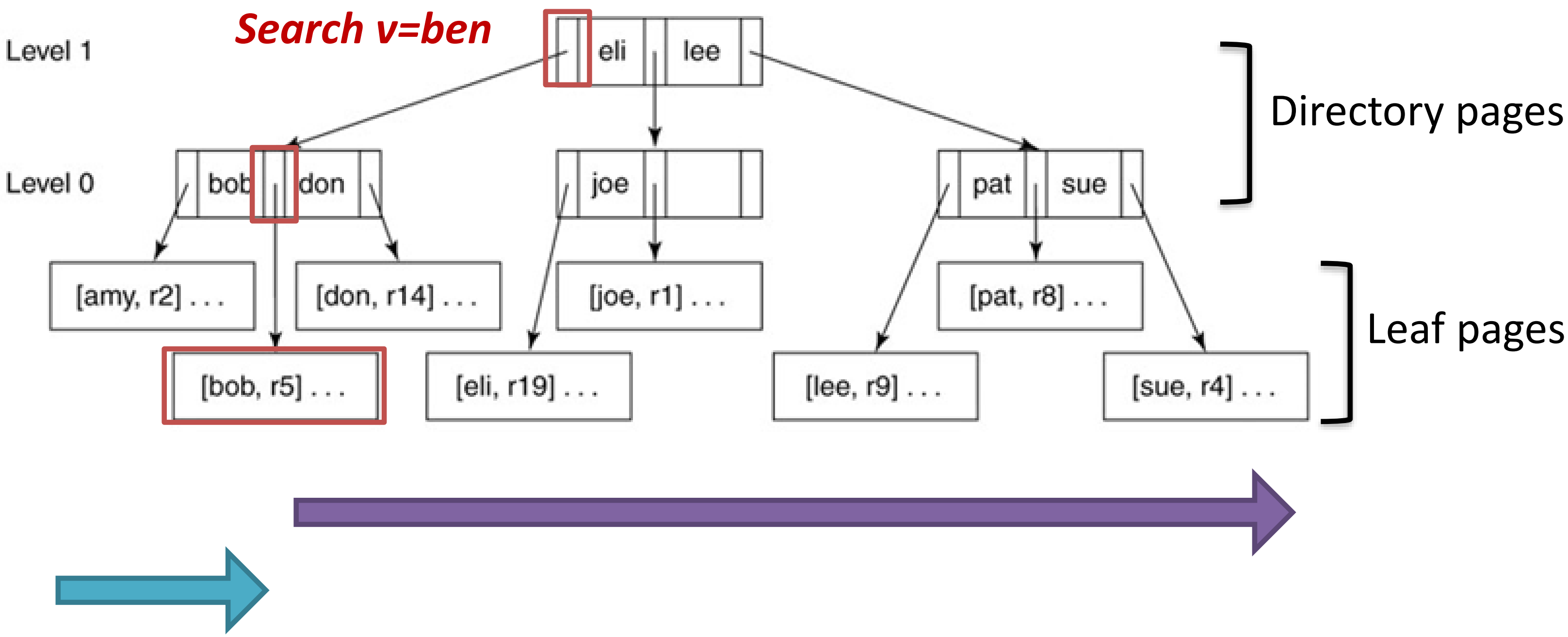
- "Finding all index records having a specified dataVal v"

1. Search begins at root
2. Fetches child block pointed by parent until leaf

- Search cost: O(tree height), usually < 5



Range Searching

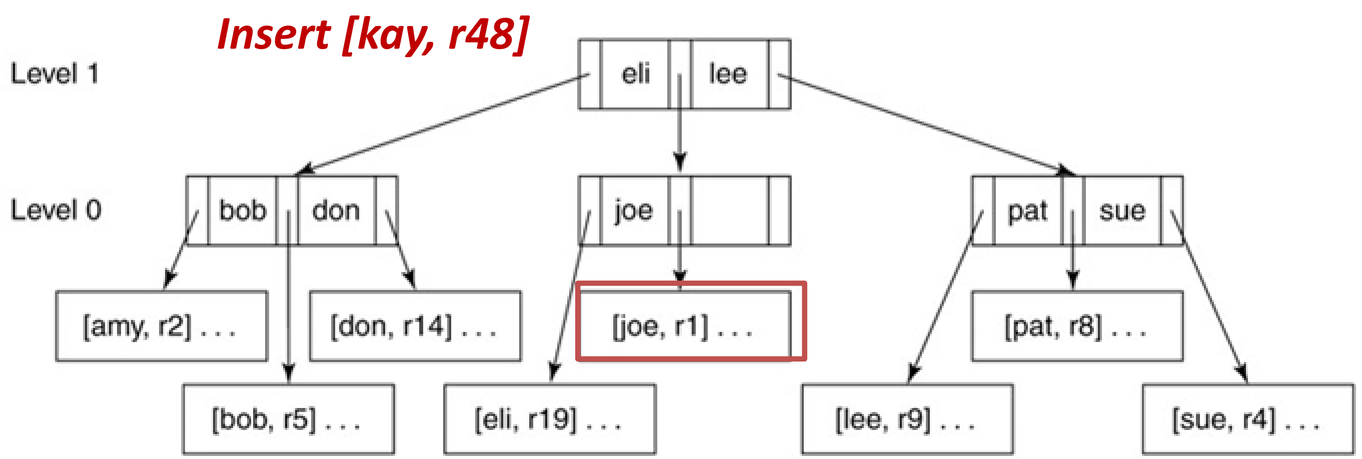


- **> v**: traverse leaf nodes from v to end
- **< v**: traverse leaf nodes from start to v

Insertion

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

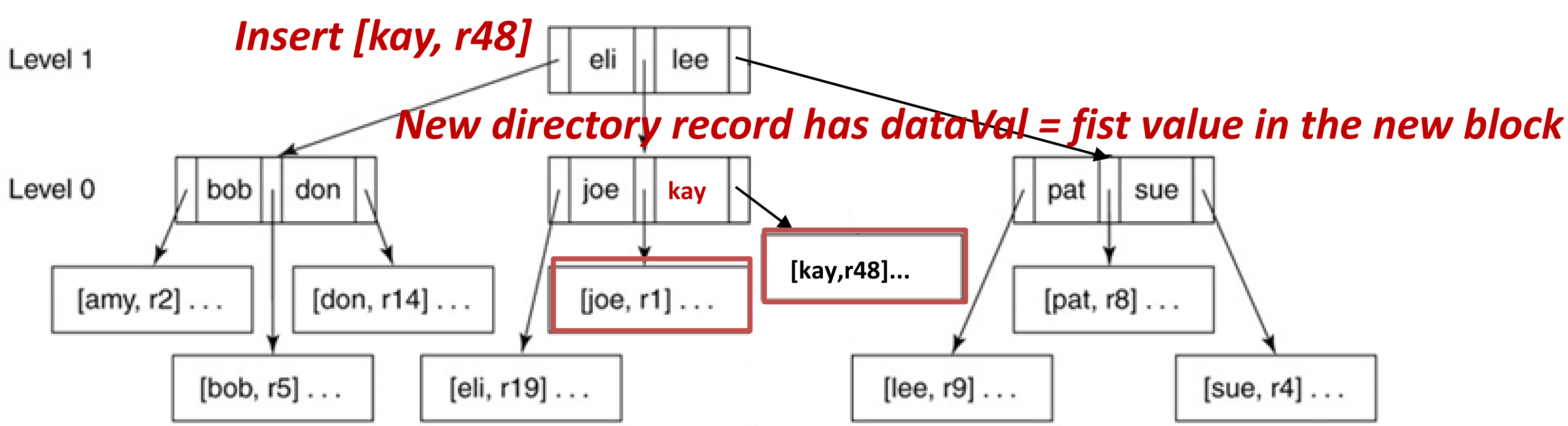
- What if the block has no more room?
  – Remember extendable hashing? ***Spilt it!***



Splitting

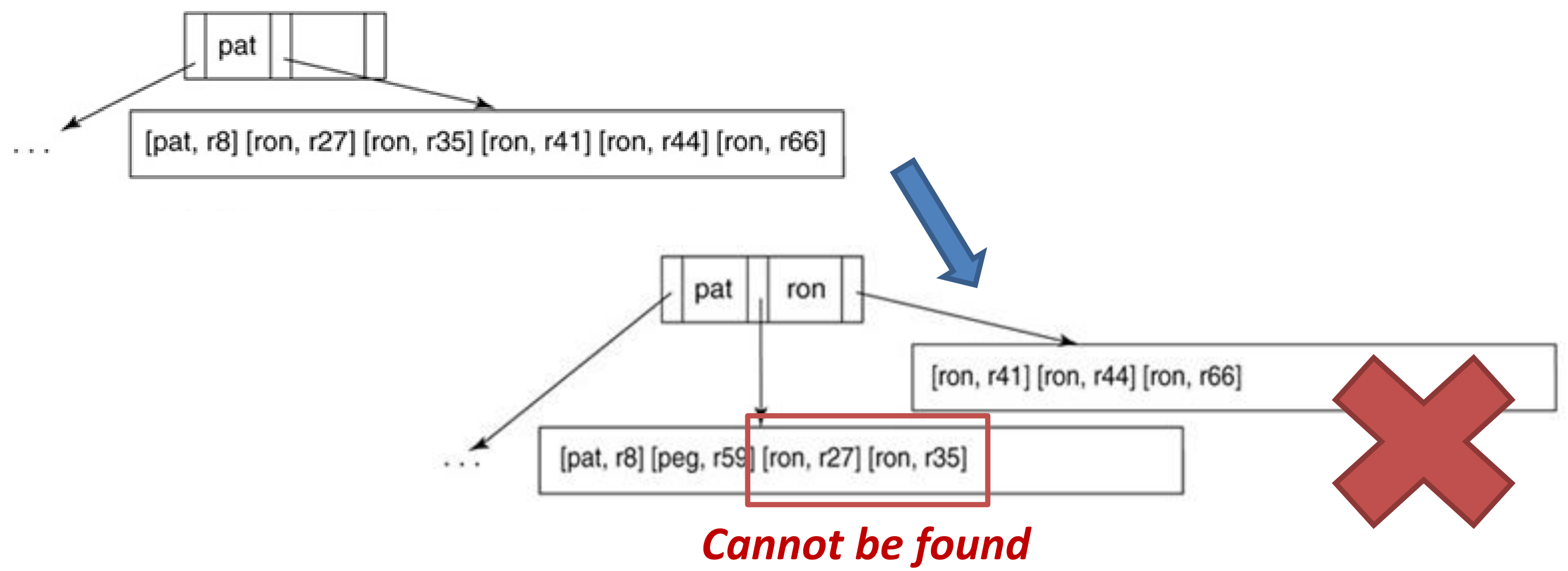
1. Leaf node: Redistribute entries evenly; ***copy up*** middle dataVal
2. Directory node (recursive): Redistribute entries evenly; ***push up*** middle dataVal

- Update cost: O(tree height)



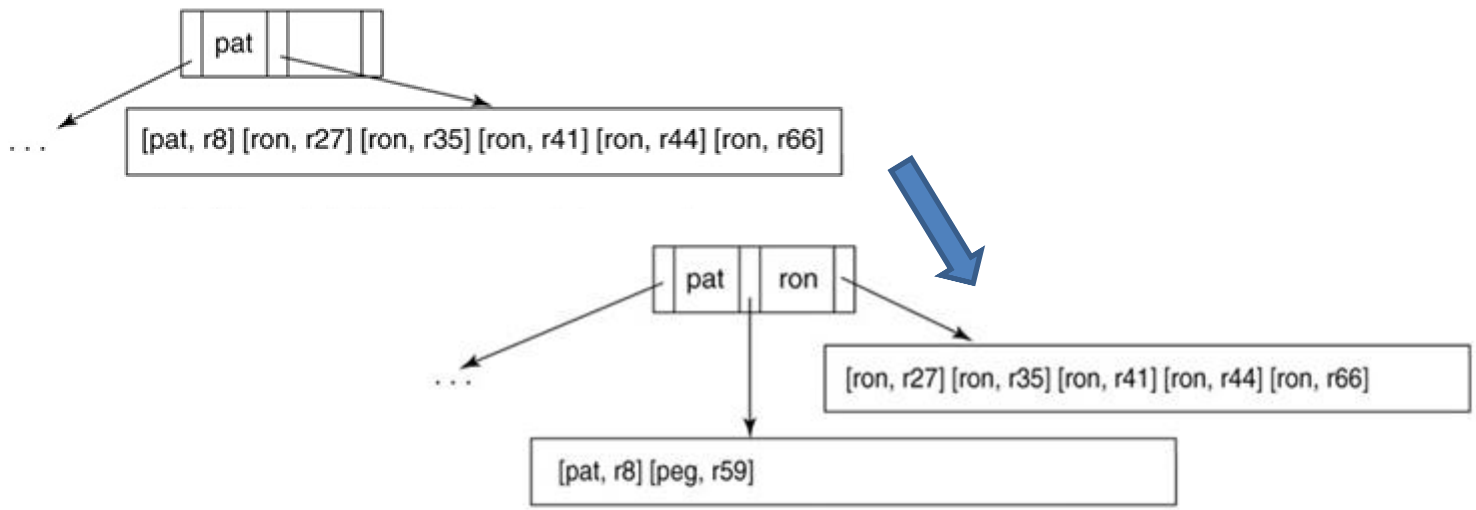
Duplicate DataVals (1/2)

- When splitting a leaf block, we must place all records with same dataVal in same block



Duplicate DataVals (2/2)

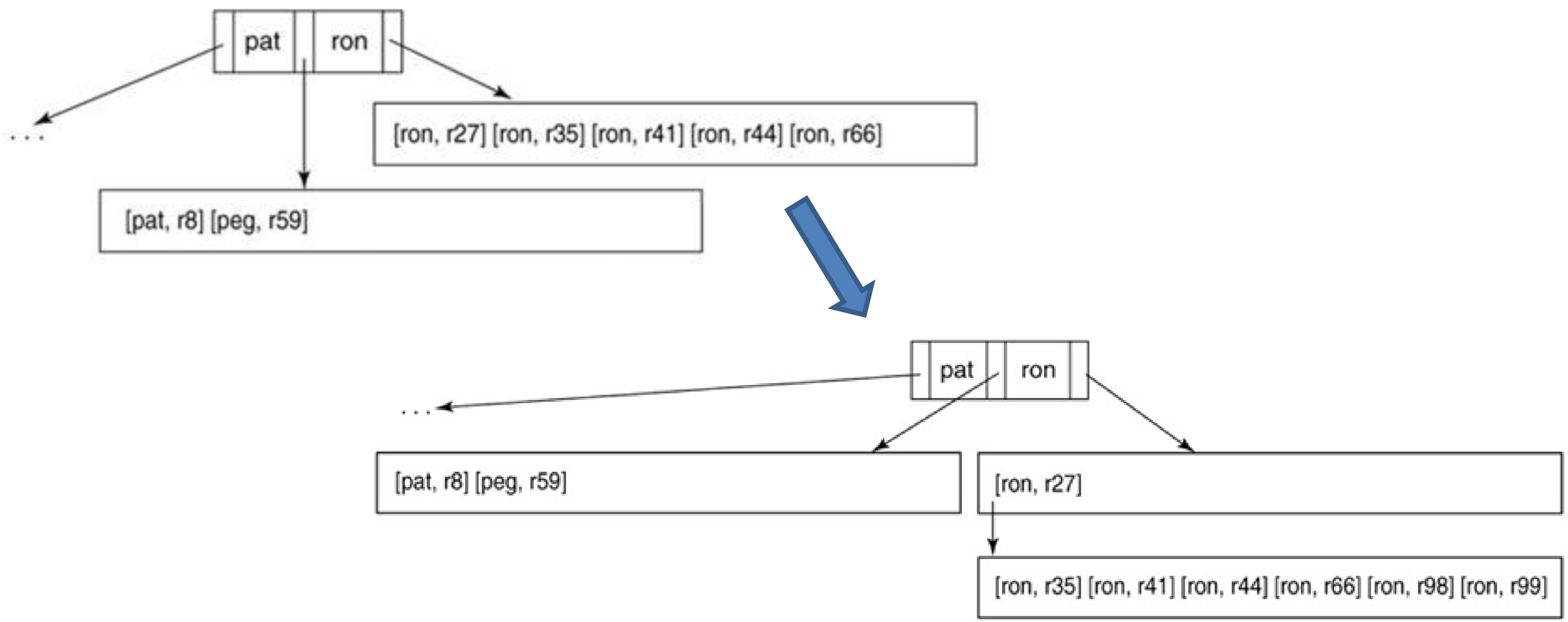
- E.g., insert [ron, r27]



- What if there are too many records with same dataVal?

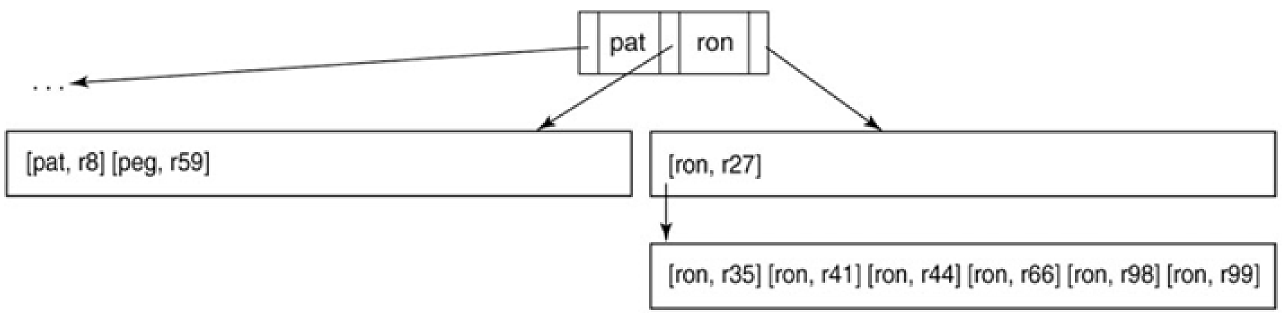
Overflow Blocks (1/2)

- Keep records of the same dataVal
- Chained by primary blocks



Overflow Blocks (2/2)

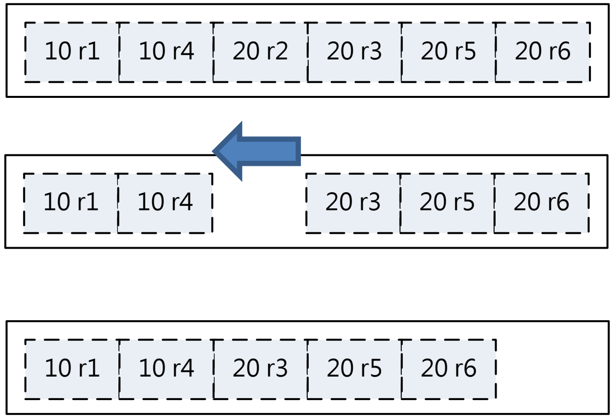
- First dataVal in primary leaf block = dataVal in overflow block



- After deleting [peg, r59], should the two leaf nodes merge?
  - ***No***

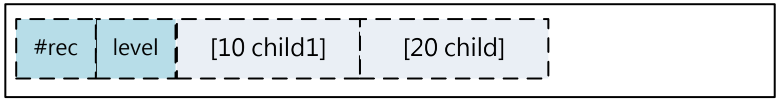
Deletion

1. Search the index with the target dataVal
2. Delete the index record in a leaf block
3. Move the next records one-slot ahead
4. Merge blocks if #records is less than a threshold
5. Recursive delete on parents

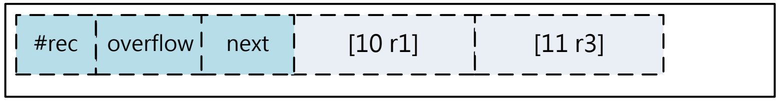


B-tree Index in VanillaCore

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



  - Leaf pages



- Supports node-splitting for insert ops
- But ***not*** merging for delete ops
  - Only records in leaf nodes are deleted, leaving directory unchanged

Outline

- Overview
  - API in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Query Processing
- Transaction Management

Related Relational Algebra

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

Update Planner

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

Outline

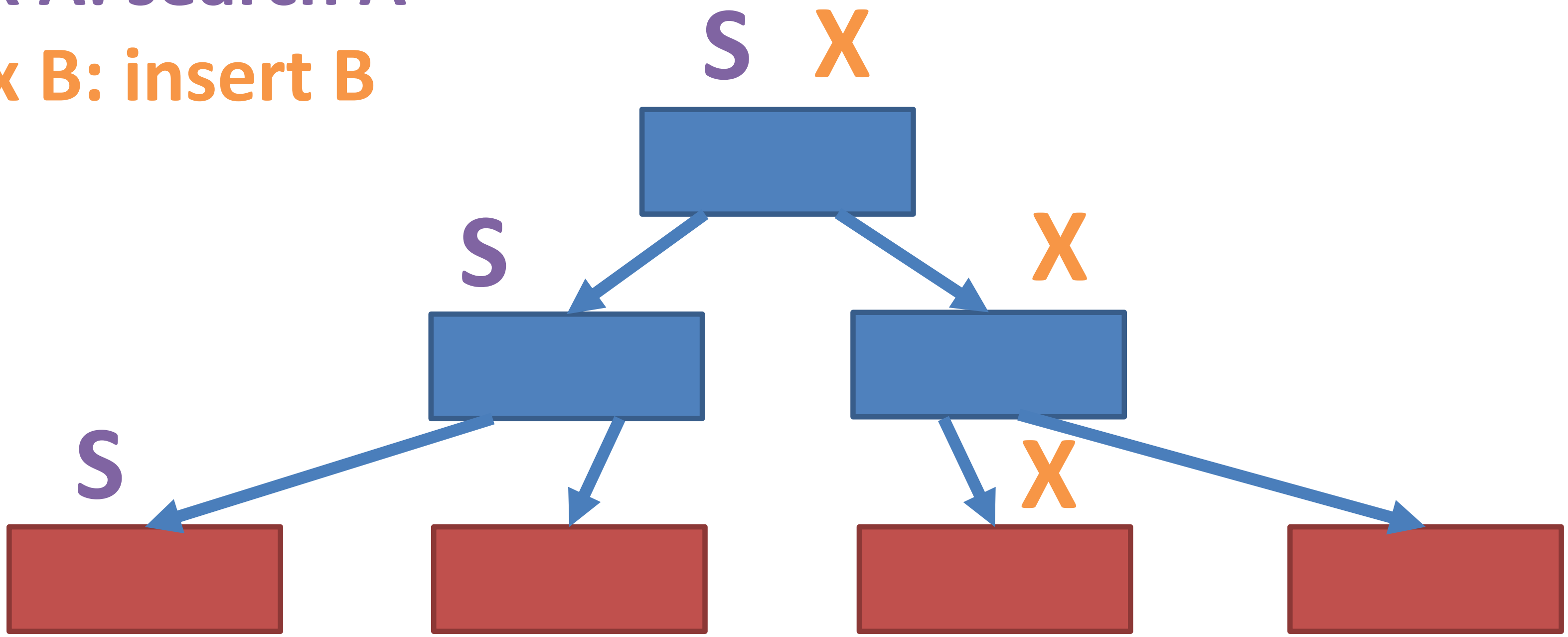
- Overview
  - API in VanillaCore
- Hash-Based Indexes
- B-Tree Indexes
- Query Processing
- Transaction Management

Index Locking

- Why?
  - To ensure I
  - Avoid phantom problems
- S2PL?
  - Index/block/record level
- Poor performance!

Block-Level S2PL

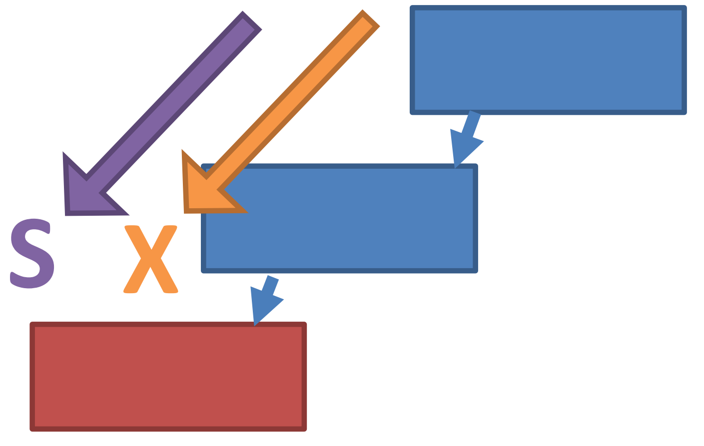
- Tx A: search A
- Tx B: insert B



- Root node becomes the bottleneck
- Better locking protocol?

Observations

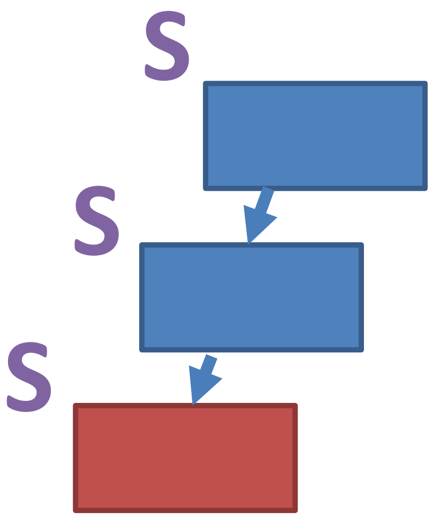
- Every tx traverse the tree from root to leaf
  - A tx can ***release "ancestor" locks early*** while still being able to prevent conflicting access



- For inserts, a split can only propagate up along "full" nodes

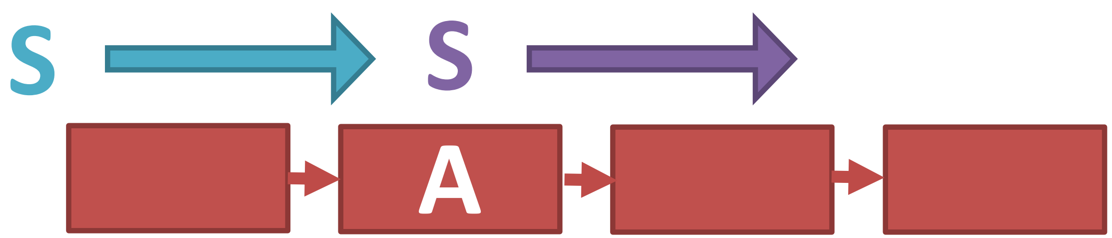
Lock Crabbing Protocol (1/2)

- Search:
  - Start at root and go down
  - S-lock child ***then unlock parent***
- Insert/delete:
  - Start at root and go down
  - X-lock child
  - ***Unlock all ancestors if child is safe***
- Safe: "not full" / "not half empty"



Lock Crabbing Protocol (2/2)

- Range searches:
  - > A: expanding locks from A to end
  - < A: expanding locks from start to A



- Locks not released early are held ***until tx ends***

Phantoms

- T_1: `SELECT * FROM users WHERE age=10;`
- T_2: `INSERT INTO users VALUES (3, 'Bob', 10); COMMIT;` ***Phantom due to insert***
- T_3: `UPDATE users SET age=10 WHERE id=7; COMMIT;` ***Phantom due to update***
- T_1: `SELECT * FROM users WHERE age=10;`

- If index on `age` is available, T2 and T3 will be blocked
- Index locking prevents phantoms due to ***both*** inserts & updates
  - A special case of ***predicate locking***

Isolation Levels (1/2)

Prevent phantoms due to inserts & updates

	Read rec	Modify/delete rec	Insert rec
`SERIALIZABLE`	S lock on index IS lock on file IS lock on block S lock on record	IX lock on file and block X lock on record X lock on index	X lock on file and block X lock on record X lock on index
`REPEATABLE READ` ***Read committed and avoid cascading abort***	S lock on index; release immediately IS lock on file and block; release immediately S lock on record	IX lock on file and block X lock on record X lock on index	IX lock on file and block X lock on record X lock in index

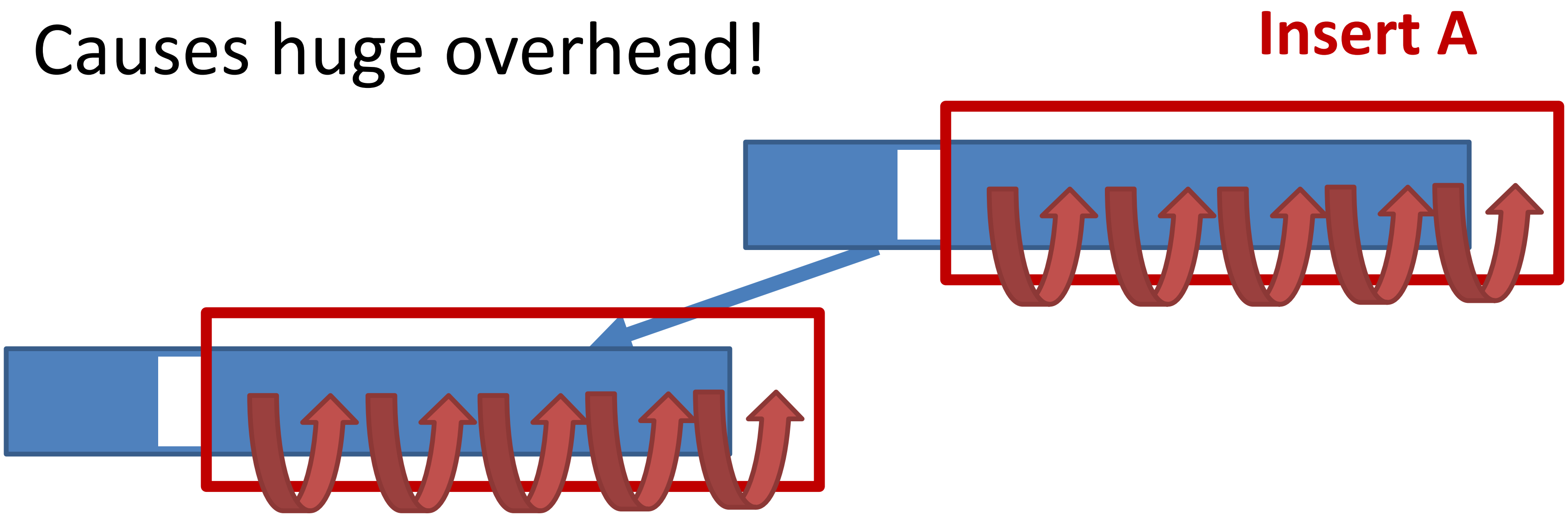
Isolation Levels (2/2)

	Read rec	Modify/delete rec	Insert rec
`READ COMMITTED`	S lock on index; release immediately	IX lock on file and block	**IX** lock on file and block
	IS lock on file and block; release immediately	X lock on record	X lock on record
	S lock on record; release upon end statement	X lock on index	X lock on index

Allow non-repeatable reads

Recovery

- Naïve: value-level, physical logging
- Causes huge overhead!



- Block-level, ***physiological*** logging
  - E.g., to log "insert at slot X"

Index Locking/Logging in VanillaDB

- Hash index: no special design
  - Rely on locking/logging mechanism implemented in `RecordFile` for each bucket
  - Locks on `FileHeaderPage` are released early; parallel inserts/deletes
- B-tree index:
  - Lock crabbing
  - Phantom prevention if index available
  - Physiological logs for block ops