Review of Lecture 2

• What is a many-to-many relationship? What is a many-to-one relationship?

• What is a weak entity set?

• How do we represent IsA relationships in tables?
Review of Lecture 2

• What are *data anomalies*?

• What is a *functional dependency*?

• When is a relation in Boyce-Codd Normal Form?
Where We Are

• Part 1: The relational data model
• Part 2: Database Systems
• Part 3: Transactions
• Part 4: Miscellaneous
Outline

• Storage and Indexes
  – Book: Ch. 8-11, and 20

• Pax paper
The Mechanics of Disk

Mechanical characteristics:
• Rotation speed (5400RPM)
• Number of platters (1-30)
• Number of tracks (<=10000)
• Number of bytes/track($10^5$)

Unit of read or write: disk block
Once in memory: page
Typically: 4k or 8k or 16k
Disk Access Characteristics

- **Disk latency**
  - Time between when command is issued and when data is in memory
  - Equals = seek time + rotational latency
- **Seek time = time for the head to reach cylinder**
  - 10ms – 40ms
- **Rotational latency = time for the sector to rotate**
  - Rotation time = 10ms
  - Average latency = 10ms/2
- **Transfer time = typically 40MB/s**

**Basic factoid:** disks always read/write an entire block at a time
RAID

Several disks that work in parallel
• Redundancy: use parity to recover from disk failure
• Speed: read from several disks at once

Various configurations (called *levels*):
• RAID 1 = mirror
• RAID 4 = n disks + 1 parity disk
• RAID 5 = n+1 disks, assign parity blocks round robin
• RAID 6 = “Hamming codes”
Storage Model

• DBMS needs spatial and temporal control over storage
  – Spatial control for performance
  – Temporal control for correctness and performance

• For spatial control, two alternatives
  – Use “raw” disk device interface directly
  – Use OS files
Spatial Control
Using “Raw” Disk Device Interface

• Overview
  – DBMS issues low-level storage requests directly to disk device

• Advantages
  – DBMS can ensure that important queries access data sequentially
  – Can provide highest performance

• Disadvantages
  – Requires devoting entire disks to the DBMS
  – Reduces portability as low-level disk interfaces are OS specific
  – Many devices are in fact “virtual disk devices”
Spatial Control
Using OS Files

• Overview
  – DBMS creates one or more very large OS files

• Advantages
  – Allocating large file on empty disk can yield good physical locality

• Disadvantages
  – OS can limit file size to a single disk
  – OS can limit the number of open file descriptors
  – But these drawbacks have mostly been overcome by modern OSs
Commercial Systems

• Most commercial systems offer both alternatives
  – Raw device interface for peak performance
  – OS files more commonly used

• In both cases, we end-up with a DBMS file abstraction implemented on top of OS files or raw device interface
File Types

The data file can be one of:

- **Heap file**
  - Set of records, partitioned into blocks
  - Unsorted

- **Sequential file**
  - Sorted according to some attribute(s) called *key*

Note: “key” here means something else than “primary key”
Buffer Management in a DBMS

Page Requests from Higher Levels

- Data must be in RAM for DBMS to operate on it!
- Table of <frame#, pageid> pairs is maintained
Buffer Manager

Needs to decide on page replacement policy

• LRU
• Clock algorithm

Both work well in OS, but not always in DB

Enables the higher levels of the DBMS to assume that the needed data is in main memory.
Arranging Pages on Disk

A disk is organized into blocks (a.k.a. pages)
• blocks on same track, followed by
• blocks on same cylinder, followed by
• blocks on adjacent cylinder

A file should (ideally) consists of sequential blocks on disk, to minimize seek and rotational delay.

For a sequential scan, pre-fetching several pages at a time is a big win!
Issues

• Managing free blocks

• File Organization

• Represent the records inside the blocks

• Represent attributes inside the records
Managing Free Blocks

• Linked list of free blocks

• Or bit map
File Organization

Linked list of pages:

Header page

Data page

Data page

Data page

Full pages

Data page

Data page

Data page

Pages with some free space
File Organization

Better: directory of pages
Page Formats

Issues to consider

• 1 page = fixed size (e.g. 8KB)

• Records:
  – Fixed length
  – Variable length

• Record id = RID
  – Typically RID = (PageID, SlotNumber)

Why do we need RID’s in a relational DBMS?
Page Formats

Fixed-length records: packed representation

One page

Rec 1 | Rec 2 | Rec N

Free space N
Page Formats

Variable-length records

Free space

Slot directory
### Current Scheme: Slotted Pages

Formal name: NSM (N-ary Storage Model)

- **Records are stored sequentially**
- **Offsets to start of each record at end of page**

---

<table>
<thead>
<tr>
<th>RID</th>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1237</td>
<td>Jane</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>4322</td>
<td>John</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>1563</td>
<td>Jim</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>7658</td>
<td>Susan</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>2534</td>
<td>Leon</td>
<td>43</td>
</tr>
<tr>
<td>6</td>
<td>8791</td>
<td>Dan</td>
<td>37</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>PAGE HEADER</th>
<th>RH1</th>
<th>1237</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>30</td>
<td>RH2</td>
</tr>
<tr>
<td>John</td>
<td>4322</td>
<td>RH3</td>
</tr>
<tr>
<td>Jim</td>
<td>1563</td>
<td>RH4</td>
</tr>
<tr>
<td>Susan</td>
<td>52</td>
<td></td>
</tr>
</tbody>
</table>

*Formal name: NSM (N-ary Storage Model)*

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Predicate Evaluation using NSM

```
select name
from R
where age > 50
```

NSM pushes non-referenced data to the cache

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
## Predicate Evaluation using NSM

### Example Page

<table>
<thead>
<tr>
<th>PAGE HEADER</th>
<th>RH1</th>
<th>RH2</th>
<th>RH3</th>
<th>RH4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>30</td>
<td>4322</td>
<td>45</td>
<td>7658</td>
</tr>
<tr>
<td>RH1</td>
<td></td>
<td></td>
<td>1563</td>
<td>52</td>
</tr>
<tr>
<td>RH2</td>
<td></td>
<td></td>
<td></td>
<td>2534</td>
</tr>
<tr>
<td>RH3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RH4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### MAIN MEMORY

- Jane
- John
- Jim
- Susan
- Leon

### CACHE

- Jane
- John
- Jim
- Susan
- Leon

### SQL Query

```sql
SELECT name FROM R WHERE age > 50
```

### NSM Pushes

NSM pushes non-referenced data to the cache.

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt]
Predicate Evaluation using NSM

select name
from R
where age > 50

NSM pushes non-referenced data to the cache

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
### Predicate Evaluation using NSM

**Predicate:**
```sql
select name from R where age > 50
```

**NSM pushes non-referenced data to the cache**

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Predicate Evaluation using NSM

select name
from R
where age > 50

NSM pushes non-referenced data to the cache

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt]
Need New Data Page Layout

- Eliminates unnecessary memory accesses
- Improves inter-record locality
- Keeps a record’s fields together
- Does not affect I/O performance

and, most importantly, is…

low-implementation-cost, high-impact

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
Partition Attributes Across (PAX)

Partition data *within* the page for spatial locality

Ailamaki VLDB’01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Partition Attributes Across (PAX)

**NSM PAGE**

<table>
<thead>
<tr>
<th>PAGE HEADER</th>
<th>RH1</th>
<th>1237</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>RH2</td>
<td>4322</td>
<td></td>
</tr>
<tr>
<td>John</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>RH3</td>
<td>1563</td>
</tr>
<tr>
<td>Jim</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>RH4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7658</td>
<td>Susan</td>
<td>52</td>
</tr>
</tbody>
</table>

**PAX PAGE**

<table>
<thead>
<tr>
<th>PAGE HEADER</th>
<th>1237</th>
<th>4322</th>
</tr>
</thead>
<tbody>
<tr>
<td>1563</td>
<td>7658</td>
<td></td>
</tr>
</tbody>
</table>

Partition data *within* the page for spatial locality

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Partition Attributes Across (PAX)

Partition data *within* the page for spatial locality

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Partition Attributes Across (PAX)

**NSM PAGE**

- **PAGE HEADER**: Jane
- **RH1**: 30
- **RH2**: 4322
- **RH3**: 45
- **RH4**: 1563
- **Susan**: 7658
- **Jim**: 20
- **John**: 1237

**PAX PAGE**

- **PAGE HEADER**: 1237
- **RH2**: 4322
- **1563**: 7658
- **30**: 52
- **45**: 20
- **Jane**: John Jim Susan

Partition data *within* the page for spatial locality

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Partition Attributes Across (PAX)

**NSM PAGE**

<table>
<thead>
<tr>
<th>PAGE HEADER</th>
<th>RH1</th>
<th>1237</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>45</td>
<td>RH2</td>
</tr>
<tr>
<td>30</td>
<td>4322</td>
<td>John</td>
</tr>
<tr>
<td>RH3</td>
<td>1563</td>
<td>Jim</td>
</tr>
<tr>
<td>1563</td>
<td>20</td>
<td>RH4</td>
</tr>
<tr>
<td>7658</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>Susan</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PAX PAGE**

<table>
<thead>
<tr>
<th>PAGE HEADER</th>
<th>1237</th>
<th>4322</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>1563</td>
<td>7658</td>
</tr>
<tr>
<td>John</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jim</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Susan</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Partition data *within* the page for spatial locality

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Partition Attributes Across (PAX)

Partition data *within* the page for spatial locality

Ailamaki VLDB’01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
Predicate Evaluation using PAX

select name
from R
where age > 50

Fewer cache misses, low reconstruction cost

Ailamaki VLDB'01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
Predicate Evaluation using PAX

```sql
select name
from R
where age > 50
```

Fewer cache misses, low reconstruction cost

Ailamaki VLDB'01 [http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt](http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt)
A Real NSM Record

NSM: All fields of record stored together + slots

Ailamaki VLDB’01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
PAX: Detailed Design

PAX: Group fields + amortizes record headers

Ailamaki VLDB’01 http://research.cs.wisc.edu/multifacet/papers/vldb01_pax_talk.ppt
Record Formats: Fixed Length

Product(pid, name, descr, maker)

<table>
<thead>
<tr>
<th>pid</th>
<th>name</th>
<th>descr</th>
<th>maker</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>L2</td>
<td>L3</td>
<td>L4</td>
</tr>
</tbody>
</table>

Base address (B)  Address = B+L1+L2

- Information about field types same for all records in a file; stored in system catalogs.
- Finding $i^{th}$ field requires scan of record.
- Note the importance of schema information!
Need the header because:

- The schema may change for a while new+old may coexist
- Records from different relations may coexist
Variable Length Records

Place the fixed fields first: F1
Then the variable length fields: F2, F3, F4
Null values take 2 bytes only
Sometimes they take 0 bytes (when at the end)
BLOB

• Binary large objects
• Supported by modern database systems
• E.g. images, sounds, etc.
• Storage: attempt to cluster blocks together

CLOB = character large object
• Supports only restricted operations
File Organizations

- **Heap** (random order) files: Suitable when typical access is a file scan retrieving all records.
- **Sorted Files** Best if records must be retrieved in some order, or only a `range’ of records is needed.
- **Indexes** Data structures to organize records via trees or hashing.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain (“search key”) fields
  - Updates are much faster than in sorted files.
Index

• A (possibly separate) file, that allows fast access to records in the data file
• The index contains (key, value) pairs:
  – The key = an attribute value
  – The value = one of:
    • pointer to the record secondary index
    • or the record itself primary index

Note: “key” (aka “search key”) again means something else
Index Classification

• **Clustered/unclustered**
  – Clustered = records close in index are close in data
  – Unclustered = records close in index may be far in data

• **Primary/secondary**
  – Meaning 1:
    • Primary = is over attributes that include the primary key
    • Secondary = otherwise
  – Meaning 2: means the same as clustered/unclustered

• **Organization** B+ tree or Hash table
Clustered Index

- File is sorted on the index attribute
- Only one per table
Unclustered Index

- Several per table
Clustered vs. Unclustered Index

CLUSTERED

UNCLUSTERED

B+ Tree

Data entries

(Index File)

(Data file)

Data Records
Hash-Based Index

Good for point queries but not range queries

Another example of clustered/primary index

Another example of unclustered/secondary index
Alternatives for Data Entry $k^*$ in Index

Three alternatives for $k^*$:

• Data record with key value $k$

• $<k, \text{rid} \text{ of data record with key } = k>$

• $<k, \text{list of rids of data records with key } = k>$
Alternatives 1, 2, 3

<table>
<thead>
<tr>
<th>10</th>
<th>ssn</th>
<th>age</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ssn</td>
<td>age</td>
<td>...</td>
</tr>
<tr>
<td>20</td>
<td>ssn</td>
<td>age</td>
<td>...</td>
</tr>
<tr>
<td>20</td>
<td>ssn</td>
<td>age</td>
<td>...</td>
</tr>
<tr>
<td>20</td>
<td>ssn</td>
<td>age</td>
<td>...</td>
</tr>
<tr>
<td>30</td>
<td>ssn</td>
<td>age</td>
<td>...</td>
</tr>
<tr>
<td>30</td>
<td>ssn</td>
<td>age</td>
<td>...</td>
</tr>
<tr>
<td>30</td>
<td>ssn</td>
<td>age</td>
<td>...</td>
</tr>
</tbody>
</table>
B+ Trees

• Search trees

• Idea in B Trees
  – Make 1 node = 1 block
  – Keep tree balanced in height

• Idea in B+ Trees
  – Make leaves into a linked list: facilitates range queries
B+ Trees Basics

• Parameter \( d \) = the **degree**
• Each node has \( \geq d \) and \( \leq 2d \) keys (except root)

![B+ Tree Diagram](image)

• Each leaf has \( \geq d \) and \( \leq 2d \) keys:

Next leaf

40 50 60

Keys \( k < 30 \)

Keys \( 30 \leq k < 120 \)

Keys \( 120 \leq k < 240 \)

Keys \( 240 \leq k \)
B+ Tree Example

d = 2

Find the key 40
B+ Tree Example

\[ d = 2 \]

Find the key 40

40 ≤ 80
B+ Tree Example

d = 2

Find the key 40

40 ≤ 80

20 < 40 ≤ 60
B+ Tree Example

d = 2

Find the key 40
Using a B+ Tree

• Exact key values:
  – Start at the root
  – Proceed down, to the leaf

• Range queries:
  – As above
  – Then sequential traversal

Index on People(age)

SELECT name
FROM People
WHERE age = 25

SELECT name
FROM People
WHERE 20 <= age and age <= 30
Which queries can use this index?

Index on People(name, zipcode)

```
SELECT * 
FROM People 
WHERE name = 'Smith' 
and zipcode = 12345
```

```
SELECT * 
FROM People 
WHERE name = 'Smith'
```

```
SELECT * 
FROM People 
WHERE zipcode = 12345
```
Insertion in a B+ Tree

Insert \((K, P)\)
- Find leaf where \(K\) belongs, insert
- If no overflow (2\(d\) keys or less), halt
- If overflow (2\(d+1\) keys), split node, insert in parent:
  - If leaf, keep \(K3\) too in right node
  - When root splits, new root has 1 key only
Insertion in a B+ Tree

Insert K=19
Insertion in a B+ Tree

After insertion
Insertion in a B+ Tree

Now insert 25
Insertion in a B+ Tree

After insertion
Insertion in a B+ Tree

But now have to split!
Insertion in a B+ Tree

After the split
Deletion from a B+ Tree

Delete 30
Deletion from a B+ Tree

After deleting 30

May change to 40, or not
Deletion from a B+ Tree

Now delete 25
Deletion from a B+ Tree

After deleting 25
Need to rebalance

Rotate
Deletion from a B+ Tree

Now delete 40
After deleting 40
Rotation not possible
Need to *merge* nodes
Deletion from a B+ Tree

Final tree
B+ Tree Design

• How large $d$?
• Example:
  – Key size = 4 bytes
  – Pointer size = 8 bytes
  – Block size = 4096 bytes
• $2d \times 4 + (2d+1) \times 8 \leq 4096$
• $d = 170$
B+ Trees in Practice

• **Typical order:** 100. **Typical fill-factor:** 67%
  - average fanout = 133

• **Typical capacities**
  - Height 4: $133^4 = 312,900,700$ records
  - Height 3: $133^3 = 2,352,637$ records

• **Can often hold top levels in buffer pool**
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 Mbytes
Practical Aspects of B+ Trees

Key compression:

• Each node keeps only the from parent keys

• Jonathan, John, Johnsen, Johnson … →
  – Parent: Jo
  – Child: nathan, hn, hnsen, hnson, …
Practical Aspects of B+ Trees

Bulk insertion

• When a new index is created there are two options:
  – Start from empty tree, insert each key one-by-one
  – Do *bulk insertion* – what does that mean?
Concurrency control

• The root of the tree is a “hot spot”
  – Leads to lock contention during insert/delete

• Solution: do proactive split during insert, or proactive merge during delete
  – Insert/delete now require only one traversal, from the root to a leaf
  – Use the “tree locking” protocol
Summary on B+ Trees

• Default index structure on most DBMS
• Very effective at answering ‘point’ queries:
  productName = ‘gizmo’
• Effective for range queries:
  50 < price AND price < 100
• Less effective for multirange:
  50 < price < 100 AND 2 < quant < 20
Hash Tables

• Secondary storage hash tables are much like main memory ones
• Recall basics:
  – There are $n$ buckets
  – A hash function $f(k)$ maps a key $k$ to $\{0, 1, \ldots, n-1\}$
  – Store in bucket $f(k)$ a pointer to record with key $k$
• Secondary storage: bucket = block, use overflow blocks when needed
### Hash Table Example

- Assume 1 bucket (block) stores 2 keys + pointers
- $h(e) = 0$
- $h(b) = h(f) = 1$
- $h(g) = 2$
- $h(a) = h(c) = 3$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>e</td>
<td>b</td>
<td>f</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>
Searching in a Hash Table

- Search for a:
- Compute $h(a)=3$
- Read bucket 3
- 1 disk access

```
   c
  3
  2   g
  1   f
  0   b
   e
```

CSEP544 - Winter, 2014
Insertion in Hash Table

- Place in right bucket, if space
- E.g. $h(d) = 2$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>g</td>
<td></td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td></td>
<td>c</td>
</tr>
</tbody>
</table>
## Insertion in Hash Table

- Create overflow block, if no space
- E.g. $h(k)=1$

### Insertion Example

<table>
<thead>
<tr>
<th>Index</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e</td>
</tr>
<tr>
<td>1</td>
<td>b, f</td>
</tr>
<tr>
<td>2</td>
<td>g, d</td>
</tr>
<tr>
<td>3</td>
<td>a, c</td>
</tr>
</tbody>
</table>

More overflow blocks may be needed.
Hash Table Performance

• Excellent, if no overflow blocks
• Degrades considerably when number of keys exceeds the number of buckets (i.e. many overflow blocks).
Extensible Hash Table

• Allows hash table to grow, to avoid performance degradation
• Assume a hash function \( h \) that returns numbers in \( \{0, \ldots, 2^k - 1\} \)
• Start with \( n = 2^i \ll 2^k \), only look at \( i \) least significant bits
Extensible Hash Table

- E.g. \( i=1, \ n=2^i=2, \ k=4 \)

- Keys:
  - 4 (\(=0100\))
  - 7 (\(=0111\))

- Note: we only look at the last bit (0 or 1)
Insertion in Extensible Hash Table

• Insert 13 (=1101)
Insertion in Extensible Hash Table

• Now insert 0101

• Need to extend table, split blocks
• i becomes 2
Insertion in Extensible Hash Table

i=1

(010)0

(011)1
(110)1, (010)1

i=2

(010)0

(11)01
(01)01

(01)11

1

2
Insertion in Extensible Hash Table

• Now insert 0000, 1110

• Need to split block
Insertion in Extensible Hash Table

• After splitting the block

1 became 2
Extensible Hash Table

• How many buckets (blocks) do we need to touch after an insertion?

• How many entries in the hash table do we need to touch after an insertion?
Performance Extensible Hash Table

• No overflow blocks: access always one read

• BUT:
  – Extensions can be costly and disruptive
  – After an extension table may no longer fit in memory
Linear Hash Table

- Idea: extend only one entry at a time
- Problem: \( n \) is no longer a power of 2
- Let \( i \) be such that \( 2^i \leq n < 2^{i+1} \)
- After computing \( h(k) \), use last \( i \) bits:
  - If last \( i \) bits represent a number > \( n \), change msb from 1 to 0 (get a number \( \leq n \)
Linear Hash Table Example

• $n=3$

```
  00 01 10
  \--\--\--
    i=2

  (01)00
  (11)00

(01)11 BIT FLIP
(10)10
```
Linear Hash Table Example

• Insert 1000: overflow blocks…
Linear Hash Tables

• Extension: independent on overflow blocks

• Extend $n := n + 1$ when average number of records per block exceeds (say) 80%
Linear Hash Table Extension

- From $n=3$ to $n=4$

- Only need to touch one block (which one?)
Linear Hash Table Extension

• From n=3 to n=4 finished

• Extension from n=4 to n=5 (new bit)

• Need to touch every single block (why?)
Indexes in Postgres

CREATE TABLE V(M int, N varchar(20), P int);

CREATE INDEX V1_N ON V(N);

CREATE INDEX V2 ON V(P, M);

CREATE INDEX VVV ON V(M, N);

CLUSTER V USING V2

Makes V2 clustered
Index Selection Problem 1

V(M, N, P);

Your workload is this

100000 queries:

SELECT * FROM V
WHERE N=?

100 queries:

SELECT * FROM V
WHERE P=?

Which indexes should we create?
Index Selection Problem 1

V(M, N, P);

Your workload is this

100000 queries:

SELECT *
FROM V
WHERE N=?

100 queries:

SELECT *
FROM V
WHERE P=?

A: V(N) and V(P) (hash tables or B-trees)
Index Selection Problem 2

V(M, N, P);

Your workload is this

100000 queries: 100 queries: 100000 queries:

SELECT * FROM V WHERE N>? and N<?

SELECT * FROM V WHERE P=?

INSERT INTO V VALUES (?, ?, ?)

Which indexes should we create?
Index Selection Problem 2

V(M, N, P);

Your workload is this
100000 queries: 100 queries: 100000 queries:

SELECT * FROM V WHERE N>? and N<?

SELECT * FROM V WHERE P=?

INSERT INTO V VALUES (?, ?, ?)

A: definitely V(N) (must B-tree); unsure about V(P)
Index Selection Problem 3

V(M, N, P);

Your workload is this
100000 queries: 1000000 queries: 100000 queries:

SELECT * FROM V WHERE N=?
SELECT * FROM V WHERE N=? and P>?
INSERT INTO V VALUES (?, ?, ?)

Which indexes should we create?
Index Selection Problem 3

V(M, N, P);

Your workload is this

100000 queries: 1000000 queries: 100000 queries:

SELECT * FROM V WHERE N = ?
SELECT * FROM V WHERE N = ? and P > ?
INSERT INTO V VALUES (?, ?, ?)

A: V(N, P)
Index Selection Problem 4

\[ V(M, N, P); \]

Your workload is this

1000 queries:

\[
\begin{align*}
&\text{SELECT } * \\
&\text{FROM } V \\
&\text{WHERE } N>? \text{ and } N<? \\
\end{align*}
\]

100000 queries:

\[
\begin{align*}
&\text{SELECT } * \\
&\text{FROM } V \\
&\text{WHERE } P>? \text{ and } P<? \\
\end{align*}
\]

Which indexes should we create?
Index Selection Problem 4

V(M, N, P);

Your workload is this
1000 queries:
SELECT * FROM V WHERE N>? and N<?

100000 queries:
SELECT * FROM V WHERE P>? and P<?

A: V(N) secondary, V(P) primary index
The Index Selection Problem

- **SQL Server**
  - Automatically, thanks to *AutoAdmin* project
  - Much acclaimed successful research project from mid 90’s, similar ideas adopted by the other major vendors

- **PostgreSQL**
  - You will do it manually, part of homework 5
  - But tuning wizards also exist
Index Selection: Multi-attribute Keys

Consider creating a multi-attribute key on K1, K2, … if

• WHERE clause has matches on K1, K2, …
  – But also consider separate indexes

• SELECT clause contains only K1, K2, ..
  – A covering index is one that can be used exclusively to answer a query, e.g. index

SELECT K2 FROM R WHERE K1=55
To Cluster or Not

- Range queries benefit mostly from clustering
- Covering indexes do not need to be clustered: they work equally well unclustered
SELECT *
FROM R
WHERE K>? and K<?
Hash Table v.s. B+ tree

• Rule 1: always use a B+ tree 😊

• Rule 2: use a Hash table on K when:
  – There is a very important selection query on equality (WHERE K=?), and no range queries
  – You know that the optimizer uses a nested loop join where K is the join attribute of the inner relation (you will understand that in a few lectures)
Balance Queries v.s. Updates

• Indexes speed up queries
  – SELECT FROM WHERE

• But they usually slow down updates:
  – INSERT, DELECTE, UPDATE
  – However some updates benefit from indexes

```sql
UPDATE R
SET A = 7
WHERE K=55
```
Tools for Index Selection

• SQL Server 2000 Index Tuning Wizard
• DB2 Index Advisor

• How they work:
  – They walk through a large number of configurations, compute their costs, and choose the configuration with minimum cost