Chapter 8: Data Storage, Indexing Structures for Files
Overview of Database Design Process

1. **Miniworld**
   - **Requirements Collection and Analysis**
     - **Functional Requirements**
     - **Data Requirements**
   - **Conceptual Design**
     - **Conceptual Schema** (In a high-level data model)
   - **Logical Design** (Data Model Mapping)
     - **Logical (Conceptual) Schema** (In the data model of a specific DBMS)
   - **Physical Design**
     - Internal Schema

2. **Application Program Design**
   - **Application Program Specification** (DBMS-independent or DBMS-specific)
   - **Transaction Implementation**
     - Application Programs
## Contents

1. **Data Storage**
   - 1.1 Disk Storage Devices
   - 1.2 Files of Records
   - 1.3 Operations on Files
   - 1.4 Unordered Files & Ordered Files & Hashed Files
   - 1.5 RAID Technology

2. **Indexing Structures for Files**
   - 2.1 Types of Single-level Ordered Indexes
   - 2.2 Multilevel Indexes
   - 2.3 Dynamic Multilevel Indexes Using B-Trees and B⁺-Trees
   - 2.4 Indexes in Oracle
## Contents

1 **Data Storage**

1.1 **Disk Storage Devices**

1.2 **Files of Records**

1.3 **Operations on Files**

1.4 **Unordered Files & Ordered Files & Hashed Files**

1.5 **RAID Technology**

2 **Indexing Structures for Files**

2.1 **Types of Single-level Ordered Indexes**

2.2 **Multilevel Indexes**

2.3 **Dynamic Multilevel Indexes Using B-Trees and B+-Trees**

2.4 **Indexes in Oracle**
Disk Storage Devices

- Preferred secondary storage device for high storage capacity and low cost.
- Data stored as magnetized areas on magnetic disk surfaces.
- A *disk pack* contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular *tracks* on each disk *surface*.
  - Track capacities vary typically from 4 to 50 Kbytes.
Disk Storage Devices (cont.)

(a) Track

(b) Sector (arc of track)

Three sectors
Two sectors
One sector
Disk Storage Devices (cont.)

- Sector
- Track
- Spindle
Disk Storage Devices (cont.)

- A track is divided into smaller **blocks** or **sectors**.
  - because a track usually contains a large amount of information.

- A track is divided into **blocks**.
  - The block size $B$ is fixed for each system.
    - Typical block sizes range from $B=512$ bytes to $B=4096$ bytes.
  - Whole blocks are transferred between disk and main memory for processing.
Disk Storage Devices (cont.)

- A **read-write head** moves to the track that contains the block to be transferred.
  - Disk rotation moves the block under the read-write head for reading or writing.
- A physical disk block (hardware) address consists of:
  - a cylinder number (imaginary collection of tracks of same radius from all recorded surfaces)
  - the track number or surface number (within the cylinder)
  - and block number (within track).
- Reading or writing a disk block is time consuming because of the seek time $s$ and rotational delay (latency) $rd$.
- **Double buffering** can be used to speed up the transfer of contiguous disk blocks.
## Contents

1 Data Storage
   1.1 Disk Storage Devices
   1.2 Files of Records
   1.3 Operations on Files
   1.4 Unordered Files & Ordered Files & Hashed Files
   1.5 RAID Technology

2 Indexing Structures for Files
   2.1 Types of Single-level Ordered Indexes
   2.2 Multilevel Indexes
   2.3 Dynamic Multilevel Indexes Using B-Trees and B⁺-Trees
   2.4 Indexes in Oracle
Records

- Fixed and variable length records.
- Records contain fields which have values of a particular type.
  - E.g., amount, date, time, age.
- Fields themselves may be fixed length or variable length.
- Variable length fields can be mixed into one record:
  - Separator characters or length fields are needed so that the record can be “parsed”.
Records (cont.)

(a)

Name  Ssn  Salary  Job_code  Department  Hire_date
1  31  40  44  48  68

(b)

Name  Ssn  Salary  Job_code  Department
1  12  21  25  29

Separator Characters

CuuDuongThanCong.com  https://fb.com/tailieudientucntt
Blocking

- **Blocking**: refers to storing a number of records in one block on the disk.
- **Blocking factor (bfr)**: refers to the number of records per block.
- There may be empty space in a block if an integral number of records do not fit in one block.
- **Spanned Records**: refer to records that exceed the size of one or more blocks and hence span a number of blocks.
Blocking (cont.)

(a) Block $i$

| Record 1 | Record 2 | Record 3 |

Block $i + 1$

| Record 4 | Record 5 | Record 6 |

(b) Block $i$

| Record 1 | Record 2 | Record 3 | Record 4 | P |

Block $i + 1$

| Record 4 (rest) | Record 5 | Record 6 | Record 7 | P |
Files of Records

- A **file** is a *sequence* of records, where each record is a collection of data values (or data items).
- A **file descriptor** (or **file header**) includes information that describes the file, such as the *field names* and their *data types*, and the addresses of the file blocks on disk.
- Records are stored on disk blocks.
- The **blocking factor** $bfr$ for a file is the (average) number of file records stored in a disk block.
- A file can have **fixed-length** records or **variable-length** records.
Files of Records (cont.)

- File records can be **unspanned** or **spanned**:
  - **Unspanned**: no record can span two blocks
  - **Spanned**: a record can be stored in more than one block

- The physical disk blocks that are allocated to hold the records of a file can be **contiguous, linked, or indexed**.

- In a file of fixed-length records, all records have the same format. Usually, unspanned blocking is used with such files.

- Files of variable-length records require additional information to be stored in each record, such as **separator characters** and **field types**.
  - Usually spanned blocking is used with such files.
Contents

1 Data Storage

1.1 Disk Storage Devices

1.2 Files of Records

1.3 Operations on Files

1.4 Unordered Files & Ordered Files & Hashed Files

1.5 RAID Technology

2 Indexing Structures for Files

2.1 Types of Single-level Ordered Indexes

2.2 Multilevel Indexes

2.3 Dynamic Multilevel Indexes Using B-Trees and B+-Trees

2.4 Indexes in Oracle
Operation on Files

Typical file operations include:

- **OPEN:** Reads the file for access, and associates a pointer that will refer to a *current* file record at each point in time.

- **FIND:** Searches for the first file record that satisfies a certain condition, and makes it the current file record.

- **FINDNEXT:** Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record.

- **READ:** Reads the current file record into a program variable.

- **INSERT:** Inserts a new record into the file, and makes it the current file record.
Operation on Files (cont.)

- **DELETE:** Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.

- **MODIFY:** Changes the values of some fields of the current file record.

- **CLOSE:** Terminates access to the file.

- **REORGANIZE:** Reorganizes the file records. For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.

- **READ_ORDERED:** Read the file blocks in order of a specific field of the file.
## Contents

1. **Data Storage**
   1.1 Disk Storage Devices
   1.2 Files of Records
   1.3 Operations on Files
   1.4 Unordered Files & Ordered Files & Hashed Files
   1.5 RAID Technology

2. **Indexing Structures for Files**
   2.1 Types of Single-level Ordered Indexes
   2.2 Multilevel Indexes
   2.3 Dynamic Multilevel Indexes Using B-Trees and B⁺-Trees
   2.4 Indexes in Oracle
Unordered Files

- Also called a **heap** or a **pile** file.
- New records are inserted at the end of the file.
- A **linear search** through the file records is necessary to search for a record.
  - This requires reading and searching half the file blocks on the average, and is hence quite expensive.
- Record insertion is quite efficient.
- Reading the records in order of a particular field requires sorting the file records.
Ordered Files

- Also called a **sequential** file.
- File records are kept sorted by the values of an **ordering field**.
- Insertion is expensive: records must be inserted in the correct order.
  - It is common to keep a separate unordered **overflow** (or **transaction**) file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A **binary search** can be used to search for a record on its **ordering field** value.
  - This requires reading and searching $\log_2$ of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.
### Ordered Files (cont.)

<table>
<thead>
<tr>
<th>Block</th>
<th>Name</th>
<th>SSN</th>
<th>Birthdate</th>
<th>Job</th>
<th>Salary</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1</td>
<td>Aaron, Ed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Abbott, Diane</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Acosta, Marc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block 2</td>
<td>Adams, John</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adams, Robin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Akers, Jan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block 3</td>
<td>Alexander, Ed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Alfred, Bob</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Allen, Sam</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block n-1</td>
<td>Wong, James</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wood, Donald</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Woods, Manny</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block n</td>
<td>Wright, Pam</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wyatt, Charles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zimmer, Byron</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CuuDuongThanCong.com

https://fb.com/tailieudientucntt
The following table shows the average access time to access a specific record for a given type of file:

<table>
<thead>
<tr>
<th>Type of Organization</th>
<th>Access/Search Method</th>
<th>Average Blocks to Access a Specific Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap (unordered)</td>
<td>Sequential scan (linear search)</td>
<td>$b/2$</td>
</tr>
<tr>
<td>Ordered</td>
<td>Sequential scan</td>
<td>$b/2$</td>
</tr>
<tr>
<td>Ordered</td>
<td>Binary search</td>
<td>$\log_2 b$</td>
</tr>
</tbody>
</table>
Hashed Files

- Hashing for disk files is called **External Hashing**.
- The file blocks are divided into $M$ equal-sized **buckets**, numbered $bucket_0$, $bucket_1$, ..., $bucket_{M-1}$.
  - Typically, a bucket corresponds to one (or a fixed number of) disk block.
- One of the file fields is designated to be the **hash key** of the file.
- The record with hash key value $K$ is stored in bucket $i$, where $i = h(K)$, and $h$ is the **hashing function**.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full.
  - An overflow file is kept for storing such records.
  - Overflow records that hash to each bucket can be linked together
Hashed Files (cont.)

Bucket Number | Block address on disk
---|---
0 | 
1 | 
2 | 
|  
|  
|  
|  
|  
|  
|  

...
Hashed Files (cont.)

There are numerous methods for collision resolution, including the following:

- **Open addressing**: Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.

<table>
<thead>
<tr>
<th>Insert 8</th>
<th>1</th>
<th>8</th>
<th>3</th>
<th>11</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert 15</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>Insert 13</td>
<td>13</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h(K) = K mod 7</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>11</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hashed Files (cont.)

- There are numerous methods for collision resolution, including the following:
  - **Chaining:**
    - Various overflow locations are kept: extending the array with a number of overflow positions.
    - A pointer field is added to each record location.
    - A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.
  - **Multiple hashing:**
    - The program applies a second hash function if the first results in a collision.
    - If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.
Hashed Files (cont.) - Overflow handling

Figure 17.10
Handling overflow for buckets by chaining.
Hashed Files (cont.)

- To reduce overflow records, a hash file is typically kept 70-80% full.
- The hash function $h$ should distribute the records uniformly among the buckets; otherwise, search time will be increased because many overflow records will exist.
- Main disadvantages of static external hashing:
  - Fixed number of buckets $M$ is a problem if the number of records in the file grows or shrinks.
  - Ordered access on the hash key is quite inefficient (requires sorting the records).
## Contents

1 Data Storage

1.1 Disk Storage Devices

1.2 Files of Records

1.3 Operations on Files

1.4 Unordered Files & Ordered Files & Hashed Files

1.5 RAID Technology

2 Indexing Structures for Files

2.1 Types of Single-level Ordered Indexes

2.2 Multilevel Indexes

2.3 Dynamic Multilevel Indexes Using B-Trees and B⁺-Trees

2.4 Indexes in Oracle
Parallelizing Disk Access using RAID Technology.

- Secondary storage technology must take steps to keep up in performance and reliability with processor technology.
- A major advance in secondary storage technology is represented by the development of **RAID**, which originally stood for **Redundant Arrays of Inexpensive Disks**.
- The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.
A natural solution is a large array of small independent disks acting as a single higher-performance logical disk.

A concept called **data striping** is used, which utilizes **parallelism** to improve disk performance.

Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk.
RAID Technology (cont.)

- Different raid organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information.
  - **Raid level 0** has no redundant data and hence has the best write performance.
  - **Raid level 1** uses mirrored disks.
  - **Raid level 2** uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.
- **Raid level 3** uses a single parity disk relying on the disk controller to figure out which disk has failed.
- **Raid levels 4 and 5** use block-level data striping, with level 5 distributing data and parity information across all disks.
RAID Technology (cont.)

- **Raid level 6** applies the so-called $P + Q$ redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks.
Use of RAID Technology (cont.)

- Different raid organizations are being used under different situations:
  - Raid level 1 (mirrored disks) is the easiest for rebuild of a disk from other disks
    - It is used for critical applications like logs.
  - Raid level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.
  - Raid level 3 (single parity disks relying on the disk controller to figure out which disk has failed) and level 5 (block-level data striping) are preferred for large volume storage, with level 3 giving higher transfer rates.
  - Most popular uses of the RAID technology currently are: Level 0 (with striping), Level 1 (with mirroring) and Level 5 with an extra drive for parity.
  - Design decisions for RAID include – level of RAID, number of disks, choice of parity schemes, and grouping of disks for block-level striping.
Storage Area Networks

- The demand for higher storage has risen considerably in recent times.
- Organizations have a need to move from a static fixed data center oriented operation to a more flexible and dynamic infrastructure for information processing.
- Thus they are moving to a concept of Storage Area Networks (SANs).
  - In a SAN, online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner.
- This allows storage systems to be placed at longer distances from the servers and provide different performance and connectivity options.
Storage Area Networks (contd.)

- **Advantages of SANs are:**
  - Flexible many-to-many connectivity among servers and storage devices using fiber channel hubs and switches.
  - Up to 10km separation between a server and a storage system using appropriate fiber optic cables.
  - Better isolation capabilities allowing nondisruptive addition of new peripherals and servers.

- SANs face the problem of combining storage options from multiple vendors and dealing with evolving standards of storage management software and hardware.
## Contents

1 Data Storage

1.1 Disk Storage Devices

1.2 Files of Records

1.3 Operations on Files

1.4 Unordered Files & Ordered Files & Hashed Files

1.5 RAID Technology

2 Indexing Structures for Files

2.1 Types of Single-level Ordered Indexes

2.2 Multilevel Indexes

2.3 Dynamic Multilevel Indexes Using B-Trees and B+-Trees

2.4 Indexes in Oracle
Indexes as Access Paths

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields).
- One form of an index is a file of entries `<field value, pointer to record>`, which is ordered by field value.
- The index is called an access path on the field.
Indexes as Access Paths (cont.)

- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller.
- A binary search on the index yields a pointer to the file record.
- Indexes can also be characterized as dense or sparse:
  - A **dense index** has an index entry for every search key value (and hence every record) in the data file.
  - A **sparse (or nondense) index**, on the other hand, has index entries for only some of the search values.
Types of Single-level Ordered Indexes

- Primary Indexes
- Clustering Indexes
- Secondary Indexes
Primary Index

- Defined on an **ordered data file**.
  - The data file is ordered on a **key field**.

- One index entry **for each block** in the data file
  - *First record* in the block, which is called the **block anchor**

- A similar scheme can use the **last record** in a block.
### Primary key field

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>DoB</th>
<th>Salary</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Index file

(<K(i), P(i)> entries)

<table>
<thead>
<tr>
<th>Primary key value</th>
<th>Block pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Primary Index

- Number of index entries?
  - Number of blocks in data file.

- Dense or Nondense?
  - Nondense

- Search/ Insert/ Update/ Delete?
Clustering Index

- Defined on an **ordered data file**.
  - The data file is ordered on a **non-key field**.

- One index entry **each distinct value** of the field.
  - The index entry points to the **first data block** that contains records with that field value
### Index file
(<K(i), P(i)> entries)

<table>
<thead>
<tr>
<th>Clustering field value</th>
<th>Block pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

### Clustering field

<table>
<thead>
<tr>
<th>Dept_No</th>
<th>Name</th>
<th>DoB</th>
<th>Salary</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clustering field value</td>
<td>Block pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Index file**

\(<K(i), P(i)> \text{ entries}\)

<table>
<thead>
<tr>
<th>Dept_No</th>
<th>Name</th>
<th>DoB</th>
<th>Salary</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Clustering Index

- Number of index entries?
  - Number of distinct indexing field values in data file.

- Dense or Nondense?
  - Nondense

- Search/ Insert/ Update/ Delete?

- At most one primary index or one clustering index but not both.
Secondary index

- A secondary index provides a secondary means of accessing a file.
  - The data file is unordered on indexing field.

- Indexing field:
  - secondary key (unique value)
  - nonkey (duplicate values)

- The index is an ordered file with two fields.
  - The first field: *indexing field*.
  - The second field: *block* pointer or *record* pointer.

- There can be *many* secondary indexes for the same file.
Index file
(<K(i), P(i)> entries)

<table>
<thead>
<tr>
<th>Index field value</th>
<th>Block pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>

Secondary key field

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
</tr>
</tbody>
</table>
Secondary index on key field

- Number of index entries?
  - Number of record in data file

- Dense or Nondense?
  - Dense

- Search/ Insert/ Update/ Delete?
Secondary index on non-key field

- **Discussion**: Structure of Secondary index on non-key field?
- Option 1: include **duplicate index entries** with the same $K(i)$ value - one for each record.
- Option 2: keep a **list of pointers** $<P(i, 1), ..., P(i, k)>$ in the index entry for $K(i)$.
- Option 3:
  - more commonly used.
  - one entry for each *distinct index field value* + an *extra level of indirection* to handle the multiple pointers.
Secondary Index on non-key field: option 3
Secondary index on nonkey field

- Number of index entries?
  - Number of records in data file
  - Number of distinct index field values

- Dense or Nondense?
  - Dense/ nondense

- Search/ Insert/ Update/ Delete?
Summary of Single-level indexes

- Ordered file on indexing field?
  - Primary index
  - Clustering index

- Indexing field is Key?
  - Primary index
  - Secondary index

- Indexing field is not Key?
  - Clustering index
  - Secondary index
Summary of Single-level indexes

- Dense index?
  - Secondary index

- Nondense index?
  - Primary index
  - Clustering index
  - Secondary index
# Summary of Single-level indexes

## Table 18.2  Properties of Index Types

<table>
<thead>
<tr>
<th>Type of Index</th>
<th>Number of (First-level) Index Entries</th>
<th>Dense or Nondense (Sparse)</th>
<th>Block Anchoring on the Data File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Number of blocks in data file</td>
<td>Nondense</td>
<td>Yes</td>
</tr>
<tr>
<td>Clustering</td>
<td>Number of distinct index field values</td>
<td>Nondense</td>
<td>Yes/no&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>Secondary (key)</td>
<td>Number of records in data file</td>
<td>Dense</td>
<td>No</td>
</tr>
<tr>
<td>Secondary (nonkey)</td>
<td>Number of records&lt;sup&gt;b&lt;/sup&gt; or number of distinct index field values&lt;sup&gt;c&lt;/sup&gt;</td>
<td>Dense or Nondense</td>
<td>No</td>
</tr>
</tbody>
</table>

<sup>a</sup>Yes if every distinct value of the ordering field starts a new block; no otherwise.

<sup>b</sup>For option 1.

<sup>c</sup>For options 2 and 3.
# Contents

## 1 Data Storage

1.1 Disk Storage Devices

1.2 Files of Records

1.3 Operations on Files

1.4 Unordered Files & Ordered Files & Hashed Files

1.5 RAID Technology

## 2 Indexing Structures for Files

2.1 Types of Single-level Ordered Indexes

2.2 Multilevel Indexes

2.3 Dynamic Multilevel Indexes Using B-Trees and B⁺-Trees

2.4 Indexes in Oracle
Multi-Level Indexes

- Because a single-level index is an ordered file, we can **create a primary index to the index itself**.
  - The original index file is called the *first-level index* and the index to the index is called the *second-level index*.

- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in *one disk block*.

- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block.
A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.
Multi-Level Indexes

- Such a multi-level index is a form of search tree.
- However, insertion and deletion of new index entries is a severe problem because every level of the index is an ordered file.
A Node in a Search Tree with Pointers to Subtrees below It
A search tree of order $p = 3$
Contents

1 Data Storage
1.1 Disk Storage Devices
1.2 Files of Records
1.3 Operations on Files
1.4 Unordered Files & Ordered Files & Hashed Files
1.5 RAID Technology

2 Indexing Structures for Files
2.1 Types of Single-level Ordered Indexes
2.2 Multilevel Indexes
2.3 Dynamic Multilevel Indexes Using B-Trees and B+-Trees
2.4 Indexes in Oracle
Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem.
- This leaves space in each tree node (disk block) to allow for new index entries.

These data structures are variations of search trees that allow efficient insertion and deletion of new search values.

In B-Tree and B+-Tree data structures, each node corresponds to a disk block.
- Each node is kept between half-full and completely full.
Dynamic Multilevel Indexes Using B-Trees and B+-Trees (cont.)

- An insertion into a node that is not full is quite efficient.
  - If a node is full, the insertion causes a split into two nodes.
- Splitting may propagate to other tree levels.
- A deletion is quite efficient if a node does not become less than half full.
- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes.
Difference between B-tree and B+-tree

- In a B-Tree, pointers to data records exist at all levels of the tree.
- In a B+-Tree, all pointers to data records exist at the leaf-level nodes.
- A B+-Tree can have less levels (or higher capacity of search values) than the corresponding B-tree.
Figure 18.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.
The Nodes of a B⁺-Tree

(a)

X ≤ K₁

K₁ ≤ X ≤ Kᵢ

Kᵢ ≤ X ≤ Kᵢ₋₁

Kᵢ₋₁ < X

(b)

K₁ Pr₁

K₂ Pr₂

... Kᵢ Prᵢ

... Kᵢ₋₁ Prᵢ₋₁

P_{next}

Data pointer

Data pointer

Data pointer

Figure 18.11
The nodes of a B⁺-tree. (a) Internal node of a B⁺-tree with q − 1 search values. (b) Leaf node of a B⁺-tree with q − 1 search values and q − 1 data pointers.
Example of insertion in $B^+$-Tree

- Tree node pointer
- Data pointer
- Null tree pointer

$p = 3$ and $p_{\text{leaf}} = 2$

Insertion Sequence: $8, 5, 1, 7, 3, 12, 9, 6$
Example of insertion in $B^+$-Tree (cont.)

Insertion Sequence: 8, 5, 1, 7, 3, 12, 9, 6
Example of insertion in $B^+$-Tree (cont.)

Insertion Sequence: $8, 5, 1, 7, 3, 12, 9, 6$
Example of insertion in $B^+$-Tree (cont.)

Insertion Sequence: 8, 5, 1, 7, 3, 12, 9, 6
Example of insertion in $B^+$-Tree (cont.)

Insertion Sequence: 8, 5, 1, 7, 3, 12, 9, 6
Example of insertion in $B^+$-Tree (cont.)

Insertion Sequence: 8, 5, 1, 7, 3, 12, 9, 6
Example of insertion in $B^+$-Tree (cont.)

Insertion Sequence: 8, 5, 1, 7, 3, 12, 9, 6
B⁺-Tree: Delete entry

- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If L has fewer entries than it should,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.
Example of deletion from $B^+$-Tree

$p = 3$ and $p_{\text{leaf}} = 2$.

Deletion sequence: 5, 12, 9
Example of deletion from B⁺-Tree (cont.)

\[ p = 3 \text{ and } p_{\text{leaf}} = 2. \]

Deletion sequence: 5, 12, 9

Delete 12: underflow (redistribute)
Example of deletion from $B^+$-Tree (cont.)

$p = 3$ and $p_{\text{leaf}} = 2$.

Deletion sequence: 5, 12, 9

Delete 9:
Underflow (merge with left, redistribute)
Example of deletion from $B^+$-Tree (cont.)

$p = 3$ and $p_{\text{leaf}} = 2$.

Deletion sequence: 5, 12, 9
Contents

1 Data Storage
  1.1 Disk Storage Devices
  1.2 Files of Records
  1.3 Operations on Files
  1.4 Unordered Files & Ordered Files & Hashed Files
  1.5 RAID Technology

2 Indexing Structures for Files
  2.1 Types of Single-level Ordered Indexes
  2.2 Multilevel Indexes
  2.3 Dynamic Multilevel Indexes Using B-Trees and B+-Trees
  2.4 Indexes in Oracle
Types of Indexes

- B-tree indexes: standard index type
  - Index-organized tables: the data is itself the index.
  - Reverse key indexes: the bytes of the index key are reversed. For example, 103 is stored as 301. The reversal of bytes spreads out inserts into the index over many blocks.
  - Descending indexes: This type of index stores data on a particular column or columns in descending order.
  - B-tree cluster indexes: is used to index a table cluster key. Instead of pointing to a row, the key points to the block that contains rows related to the cluster key.
Types of Indexes (cont.)

- Bitmap and bitmap join indexes: an index entry uses a bitmap to point to multiple rows. A bitmap join index is a bitmap index for the join of two or more tables.

- Function-based indexes:
  - Includes columns that are either transformed by a function, such as the UPPER function, or included in an expression.
  - B-tree or bitmap indexes can be function-based.

- Application domain indexes: customized index specific to an application.
Creating Indexes

Simple create index syntax:

```
CREATE [ UNIQUE | BITMAP ] INDEX
    [schema.] <index_name>
    ON [schema.] <table_name> (column [ ASC | DESC ] [ , column [ ASC | DESC ] ] ...) [REVERSE];
```
Example of creating indexes

- **CREATE INDEX** ord_customer_ix ON ORDERS (customer_id);

- **CREATE INDEX** emp_name_dpt_ix ON HR.EMPLOYEES(last_name ASC, department_id DESC);

- **CREATE BITMAP INDEX** emp_gender_idx ON EMPLOYEES (Sex);

- **CREATE BITMAP INDEX** emp_bm_idx ON EMPLOYEES (JOBS.job_title) FROM EMPLOYEES, JOBS WHERE EMPLOYEES.job_id = JOBS.job_id;
Example of creating indexes (cont.)

Function-Based Indexes:

- **CREATE INDEX** `emp_fname_uppercase_idx`  
  ON `EMPLOYEES` ( `UPPER(first_name)` );

- **SELECT** `First_name, Lname`  
  FROM `Employee` WHERE `UPPER(Lname)` = "SMITH";

- **CREATE INDEX** `emp_total_sal_idx`  
  ON `EMPLOYEES` ( `salary + (salary * commission_pct)` );

- **SELECT** `First_name, Lname`  
  FROM `Employee` WHERE `((Salary*Commission_pct) + Salary)` > 15000;
Guidelines for creating indexes

- Primary and unique keys automatically have indexes, but you might want to create an index on a foreign key.
- Create an index on any column that the query uses to join tables.
- Create an index on any column from which you search for particular values on a regular basis.
- Create an index on columns that are commonly used in ORDER BY clauses.
- Ensure that the disk and update maintenance overhead an index introduces will not be too high.
Dynamic and Extendible Hashing Techniques

- Hashing techniques are adapted to allow the dynamic growth and shrinking of the number of file records.
- These techniques include the following: *dynamic hashing*, *extendible hashing*, and *linear hashing*.
- Both dynamic and extendible hashing use the *binary representation* of the hash value $h(K)$ in order to access a *directory*. In dynamic hashing the directory is a binary tree. In extendible hashing the directory is an array of size $2^d$ where $d$ is called the *global depth*. 
Dynamic And Extendible Hashing (cont.)

- The directories can be stored on disk, and they expand or shrink dynamically. Directory entries point to the disk blocks that contain the stored records.
- An insertion in a disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks. The directory is updated appropriately.
- Dynamic and extendible hashing do not require an overflow area.
- Linear hashing does require an overflow area but does not use a directory. Blocks are split in linear order as the file expands.
Extendible Hashing

DATA FILE BUCKETS

- $d' = 3$
  - bucket for records whose hash values start with 000

- $d' = 3$
  - bucket for records whose hash values start with 001

- $d' = 2$
  - bucket for records whose hash values start with 01

- $d' = 2$
  - bucket for records whose hash values start with 10

- $d' = 3$
  - bucket for records whose hash values start with 110

- $d' = 3$
  - bucket for records whose hash values start with 111
## Extendible Hashing – Example

<table>
<thead>
<tr>
<th>Record</th>
<th>K</th>
<th>h(K) = K % 32</th>
<th>h(K)B</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>2657</td>
<td>1</td>
<td>00001</td>
</tr>
<tr>
<td>r2</td>
<td>3760</td>
<td>16</td>
<td>10000</td>
</tr>
<tr>
<td>r3</td>
<td>4692</td>
<td>20</td>
<td>10100</td>
</tr>
<tr>
<td>r4</td>
<td>4871</td>
<td>7</td>
<td>00111</td>
</tr>
<tr>
<td>r5</td>
<td>5659</td>
<td>27</td>
<td>11011</td>
</tr>
<tr>
<td>r6</td>
<td>1821</td>
<td>29</td>
<td>11101</td>
</tr>
<tr>
<td>r7</td>
<td>1074</td>
<td>18</td>
<td>10010</td>
</tr>
<tr>
<td>r8</td>
<td>2123</td>
<td>11</td>
<td>01011</td>
</tr>
<tr>
<td>r9</td>
<td>1620</td>
<td>20</td>
<td>10100</td>
</tr>
<tr>
<td>r10</td>
<td>2428</td>
<td>28</td>
<td>11100</td>
</tr>
<tr>
<td>r11</td>
<td>3943</td>
<td>7</td>
<td>00111</td>
</tr>
<tr>
<td>r12</td>
<td>4750</td>
<td>14</td>
<td>01110</td>
</tr>
<tr>
<td>r13</td>
<td>6975</td>
<td>31</td>
<td>11111</td>
</tr>
<tr>
<td>r14</td>
<td>4981</td>
<td>21</td>
<td>10101</td>
</tr>
<tr>
<td>r15</td>
<td>9208</td>
<td>24</td>
<td>11000</td>
</tr>
</tbody>
</table>

$d'$ = local depth  
$d$ = global depth

<table>
<thead>
<tr>
<th>r1 (00001)</th>
<th>d' = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2 (10000)</td>
<td>d = 0</td>
</tr>
</tbody>
</table>

Each bucket has maximum 2 records
Extendible Hashing – Example (cont.)

- r1 (00001) => overflow => splitting
  - d' = 0
- r2 (10000)
  - d = 0
  - Insert r3 (10100) => overflow => splitting
  - d = 1
  - r1 (00001) d' = 1
  - r2 (10000)
  - r3 (10100)
  - Insert r4 (00111)
Extendible Hashing – Example (cont.)

Insert r5 (11011) => overflow => splitting

```
0 1
   d = 1
```

```
r1 (00001) | d' = 1
r4 (00111)

r2 (10000) | d' = 1
r3 (10100)
```
Extendible Hashing – Example (cont.)

\[
\begin{align*}
00 & \rightarrow r1 (00001) \quad d' = 1 \\
01 & \rightarrow r4 (00111) \\
10 & \rightarrow \text{ } \\
11 & \rightarrow r2 (10000) \quad d' = 2 \\
& \rightarrow r3 (10100) \\
& \rightarrow r5 (11011) \quad d' = 2 \\
& \rightarrow \text{ (11111)} \\
\end{align*}
\]

Insert r6 (11101)
Extendible Hashing – Example (cont.)

Insert r7 (10010) =>
overflow => splitting

00
01
10
11

d = 2

r1 (00001)  d' = 1
r4 (00111)

r2 (10000)  d' = 2
r3 (10100)

r5 (11011)  d' = 2
r6 (11101)

Insert r7 (10010) =>
overflow => splitting
Extendible Hashing – Example (cont.)

Insert r8 (01011) =>
overflow => splitting

d = 3

r1 (00001)  \( d' = 1 \)
r4 (00111)  

r2 (10000)  \( d' = 3 \)
r7 (10010)  

r3 (10100)  \( d' = 3 \)

r5 (11011)  \( d' = 2 \)
r6 (11101)  

d = 3
Extendible Hashing – Example (cont.)

Insert $r_9 (10100)$

<table>
<thead>
<tr>
<th>Level</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$r_1 (00001)$, $d' = 2$</td>
</tr>
<tr>
<td>1</td>
<td>$r_4 (00111)$</td>
</tr>
<tr>
<td>2</td>
<td>$r_8 (01011)$, $d' = 2$</td>
</tr>
<tr>
<td>3</td>
<td>$r_2 (10000)$, $d' = 3$</td>
</tr>
<tr>
<td>4</td>
<td>$r_7 (10010)$</td>
</tr>
<tr>
<td>5</td>
<td>$r_3 (10100)$, $d' = 3$</td>
</tr>
<tr>
<td>6</td>
<td>$r_5 (11011)$, $d' = 2$</td>
</tr>
<tr>
<td>7</td>
<td>$r_6 (11101)$</td>
</tr>
</tbody>
</table>

$d = 3$
Extendible Hashing – Example (cont.)

```
010 011
100 101
110 111
000 001 010 011 100 101 110 111
d = 3
```

- r1 (00001) \( d' = 2 \)
- r4 (00111)
- r8 (01011) \( d' = 2 \)
- r2 (10000) \( d' = 3 \)
- r7 (10010)
- r3 (10100) \( d' = 3 \)
- r9 (10100)
- r5 (11011) \( d' = 2 \)
- r6 (11101)

Insert r10 (11100) => overflow => splitting
Extendible Hashing – Example (cont.)

Insert r11 (00111) => overflow => splitting

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
<th>d'</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>r1</td>
<td>2</td>
</tr>
<tr>
<td>001</td>
<td>r4</td>
<td>2</td>
</tr>
<tr>
<td>010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>r8</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>r2</td>
<td>3</td>
</tr>
<tr>
<td>101</td>
<td>r7</td>
<td>2</td>
</tr>
<tr>
<td>110</td>
<td>r3</td>
<td>3</td>
</tr>
<tr>
<td>111</td>
<td>r5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>r6</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>r10</td>
<td>3</td>
</tr>
</tbody>
</table>
Extendible Hashing – Example (cont.)

```
000  r1 (00001)  d' = 3
     r8 (01011)  d' = 2
001  r4 (00111)  d' = 3
     r11 (00111)
010  r2 (10000)  d' = 3
     r7 (10010)
011
100  r3 (10100)  d' = 3
     r9 (10100)
101
110  r5 (11011)  d' = 3
     r6 (11101)  d' = 3
111  r10 (11100)
```

Insert r12 (01110)

---

CuuDuongThanCong.com
https://fb.com/tailieudientucntt
Extendible Hashing – Example (cont.)

Insert r13 (11111) =>
overflow => splitting

d = 3

r1 (00001)  d' = 3
r4 (00111)  d' = 3
r11 (00111)

r8 (01011)  d' = 2
r12 (01110)

r3 (10100)  d' = 3
r9 (10100)

r5 (11011)  d' = 3

r6 (11101)  d' = 3
r10 (11100)
Extendible Hashing – Example (cont.)

r1 (00001)  \( d' = 3 \)

r4 (00111)  \( d' = 3 \)

r11 (00111)

r8 (01011)  \( d' = 2 \)

r2 (10000)  \( d' = 3 \)

r7 (10010)

r12 (01110)

r3 (10100)  \( d' = 3 \)

r5 (11011)  \( d' = 3 \)

r9 (10100)

r6 (11101)  \( d' = 4 \)

r10 (11100)

r1 (11111)  \( d' = 4 \)

d = 4

Insert r14 (10101) => overflow => splitting

CuuDuongThanCong.com

https://fb.com/tailieudientucntt
Extendible Hashing – Example (cont.)

- Insert r15 (11000)
- d = 4
- r1 (00001) d' = 3
- r4 (00111) d' = 3
- r11 (00111)
- r8 (01011) d' = 2
- r12 (01110)
- r2 (10000) d' = 3
- r7 (10010)
- r3 (10100) d' = 4
- r9 (10100)
- r14 (10101) d' = 4
- r15 (11000)
- Insert r15 (11000)
- r6 (11101) d' = 4
- r10 (11100)
- r5 (11011) d' = 3
- r1 (11111) d' = 4
- r1 (11111)
Extendible Hashing – Example (cont.)

\[
\begin{array}{ccc}
0000 & \rightarrow & r1 (00001) \quad d' = 3 \\
0001 & & \\
0010 & \rightarrow & r8 (01011) \quad d' = 2 \\
0011 & & \\
0100 & \rightarrow & r12 (01110) \\
0101 & & \\
0110 & \rightarrow & r3 (10100) \quad d' = 4 \\
0111 & & \\
1000 & \rightarrow & r9 (10100) \\
1001 & & \\
1010 & \rightarrow & r14 (10101) \quad d' = 4 \\
1011 & & \\
1100 & \rightarrow & r6 (11101) \quad d' = 4 \\
1101 & & \\
1110 & \rightarrow & r10 (11100) \\
1111 & & \\
\end{array}
\]

- \( d = 4 \)
- \( r4 (00111) \quad d' = 3 \)
- \( r11 (00111) \)
- \( r2 (10000) \quad d' = 3 \)
- \( r7 (10010) \)
- \( d' = 4 \)
- \( r5 (11011) \quad d' = 3 \)
- \( r15 (11000) \)
- \( r1 (11111) \quad d' = 4 \)
Linear Hashing – Example

- $M=4$, $h_0(K) = K \mod M$, each bucket has 3 records.
- Initialization:

  \[
  h_1(K) = K \mod 2 \times M
  \]

  Insert 17
  (17 mod 4 = 1)

  Split bucket 0
  Bucket 1: overflow
Linear Hashing – Example (cont.)

Split pointer

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>:</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>:</td>
</tr>
<tr>
<td>6</td>
<td>:</td>
<td>7</td>
<td>11</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>4</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>17</td>
<td>:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4: bucket (4 mod 2*4 =) 4
8: bucket (8 mod 2*4 =) 0
17: overflow records
Linear Hashing – Example (cont.)

**Split pointer**

```
0 1 2 3 4
8 : : 5 : 9 : 13 6 : : 7 : 11 : 4 : :
```

Insert 15
(15 mod 4 = 3)
Linear Hashing – Example (cont.)

Bucket 3: overflow
Split bucket 1.
=> Overflow records: Redistributed
Linear Hashing – Example (cont.)

5: bucket \((5 \mod 2 \times 4 = )\) 5
9: bucket \((9 \mod 2 \times 4 = )\) 1
13: bucket \((13 \mod 2 \times 4 = )\) 5
17: bucket \((17 \mod 2 \times 4 = )\) 1

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>:</td>
<td>9</td>
<td>17</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>6</td>
<td>:</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td>:</td>
</tr>
<tr>
<td>4</td>
<td>:</td>
<td>5</td>
<td>:</td>
<td>:</td>
<td>13</td>
</tr>
</tbody>
</table>

*Split pointer*
Bucket 3: overflow.

Split bucket 2.

Insert 23
(23 mod 4 = 3)
Bucket 3: overflow

Split bucket 3

=> Overflow records: Redistributed

\[\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}\]

Split pointer

insert 31
(31 mod 4 = 3)

3: 23:
Linear Hashing – Example (cont.)

7: bucket (7 mod 2*4 = ) 7
11: bucket (11 mod 2*4 = ) 3
15: bucket (15 mod 2*4 = ) 3
3: bucket (3 mod 2*4 = ) 3
23: bucket (23 mod 2*4 = ) 7
31: bucket (31 mod 2*4 = ) = 7

\[ h_1(K) = K \mod 8 \]

\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}

Split pointer